Universidad de Buenos Aires

Facultad de Ingeniería

Trabajo Práctico I:

File Transfer

Redes (TA048)

1° Cuatrimestre 2025

Alumno	**Padrón**	**E-mail**
Agustín Barbalase	109071	abarbalase@fi.uba.ar
Felipe D'alto	110000	fedalto@fi.uba.ar
Nicolás Sanchez	98792	nrsanchez@fi.uba.ar
Santiago Sevitz	107520	ssevitz@fi.uba.ar
Máximo Utrera	109651	mutrera@fi.uba.ar

Fecha de Presentación: 20 de mayo de 2025

Índice

1 Introducción

Este proyecto consta en la investigación y la resolución práctica sobre cómo diseñar un protocolo de aplicación, con respecto a la transmisión de archivos, con la premisa de conseguir un sistema de transferencia de datos **confiable** utilizando como protocolo de transporte UDP (siendo el mismo *no confiable*).

Se ha realizado una emulación sobre el caso de pérdida de paquetes utilizando la herramienta **mininet**, la cual crea una red virtual local en el ordenador, y permite simular un porcentaje de pérdida de paquetes, para luego poder verificar el correcto funcionamiento del protocolo.

Como arquitectura principal se realizó una transferencia entre un servidor y posibles múltiples clientes, cuyos clientes tienen la posibilidad de descargar y cargar (upload/download) archivos, y se puede realizar transacciones simultáneamente.

2 Hipótesis y suposiciones realizadas

Para el desarrollo de este trabajo, se tomaron las siguientes suposiciones:

- Todo paquete que llega al protocolo de aplicación estará completo e integro.
- Los paquetes UDP podrían no llegar a destino.
- Los paquetes UDP podrían llegar fuera de orden.
- El protocolo a implementar, debe suportar mínimamente una perdida de paquetes del 10 %.
- Los tiempos de transferencia deben ser razonables (e.g. menor a $2min$ para un archivo de $5MB$ en una red con 10 % de perdida).
- El protocolo a implementar, debe ser capaz de transferir archivos binarios.
- Se considera que la transmisión es confiable si a pesar de tener pérdida de paquetes con información, éstos lleguen fuera de orden o incluso lleguen repetidos, estos eventos no afecten a la correcta transferencia de ellos.

3 Implementación

En esta sección se demuestra el desarrollo de un protocolo de aplicación de transferencia confiable de archivos, desde la confección de su encabezado hasta la implementación de dos mecanismos de recuperación (Stop & Wait y Go-Back-N), entre otros detalles.

3.1. Protocolo

3.1.1. Header

Primeramente se decidió en lineas generales como seria el encabezado (o *header*) del protocolo a implementar. Para esto se tuvo en cuenta que algunos de los datos necesarios para el funcionamiento del mismo son el protocolo de recuperación, el modo de operación (cargar/descargar archivos), el largo de la sección de datos, un numero de *acknoledge* y otro de *sequence* (cuya explicación se encuentra en la sección de *mecanismo de recuperación*), y finalmente tres $flags$ adicionales para facilitar la comunicación.

Al unir dichos datos en un encabezado, y asignar a cada uno un tamaño especifico en función de su necesidad, se obtuvo el siguiente *header* de tamaño fijo ($6B$).

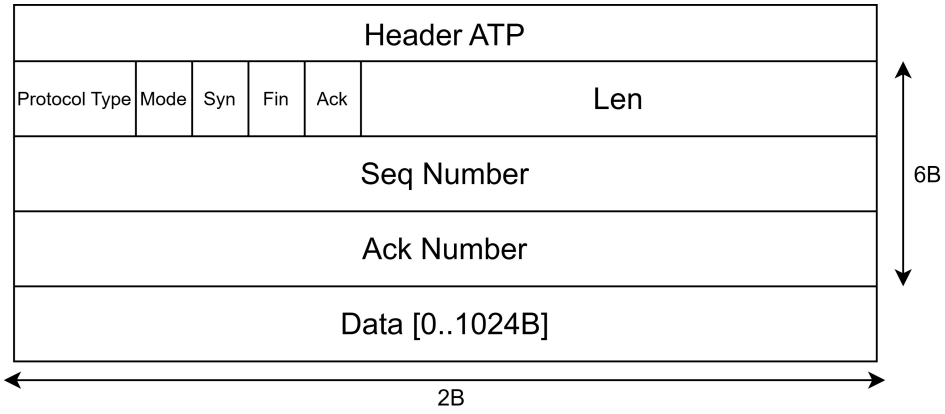


Figura 1: Diagrama del *header* confeccionado

- Para el *Protocol type* se asignaron dos bits, de los cuales solo se utiliza uno, sin embargo se decidió agregar ese bit extra para dar espacio a la posible implementación de dos nuevos mecanismos de recuperación.
- En el caso de los $flags$, se utilizo solamente un bit para cada uno, y esto es suficiente para denotar la presencia o ausencia de una determinada opción.
- Para el largo del paquete se decidió utilizar un total de 10 bits, lo que permite un largo máximo de $l_{max} = 2^{10}B = 1024B$. Para llegar a esta decisión se hizo una investigación previa, sobre el MTU de un enlace promedio, y se encontró que el mismo es de aproximadamente $1500B$, lo que significa que los paquetes que excedan ese largo tendrán que ser fragmentados o descartados.
- Finalmente para los números de secuencia y de *acknoledge*, se utilizaron $2B$ para cada uno, esto permite (en un contexto de secuencias incrementales) un total de 2^{16} paquetes enviados antes de que ocurra un 'wrap-around' y se vuelva a la secuencia 0.

3.2. Flujo de mensajes

En esta sección se demuestran y explican los mensajes que utiliza el protocolo desarrollado para establecer conexiones, y posteriormente realizar transferencias de archivos.

3.2.1. Handshake

Se ha asentado un intercambio de mensajes iniciales, para entablar una conexión entre el cliente y el servidor. Para esto el cliente sera quien envíe el primer mensaje (uno con la opción de **syn** encendida), en respuesta a este, el servidor hace la configuración interna para poder administrar el nuevo cliente y si no hay problemas le enviara al mismo un mensaje de aceptación (con las opciones **syn** y **ack** encendidas) concluyendo así la inicializacion de la conexión.

Algunos manejos de situaciones de error son:

- En caso de que un cliente utilice un mecanismo de recuperación diferente al que utiliza el servidor, la conexión sera rechazada con un mensaje **fin**.
- Considerando que puede haber pérdida de paquetes durante esta etapa, se tiene una serie de reintentos y un timeout si no se logra establecer una conexión entre las partes.

3.2.2. Establecimiento de modo de operación y nombre de archivo

Una vez establecida la conexión, el cliente debe enviar su consulta (o *request*) al servidor conteniendo en la sección de datos, el nombre del archivo sobre el cual se quiere realizar la operación. A su vez en este mensaje se establece la opción **mode** en alguna de las dos operaciones disponibles (carga/descarga). El servidor responde al mismo con un mensaje **ack** y procede a realizar la operación pedida.

En caso de perdida de paquetes al igual que anteriormente, se llevan a cabo una serie de reintentos hasta que se retome la comunicación o se de por cerrada.

3.2.3. Transferencia del archivo

Luego de establecida la conexión, y hecha la consulta de transferencia de un archivo dado, se entra en la etapa de transferencia. En esta etapa entra en juego el mecanismo de recuperación elegido, cuyos detalles se verán en la siguiente sección.

Al finalizar la transferencia el lado emisor, envía un mensaje final con la opción **fin**, a lo que el receptor le responde con un mensaje **fin-ack** y se da por cerrada la conexión.

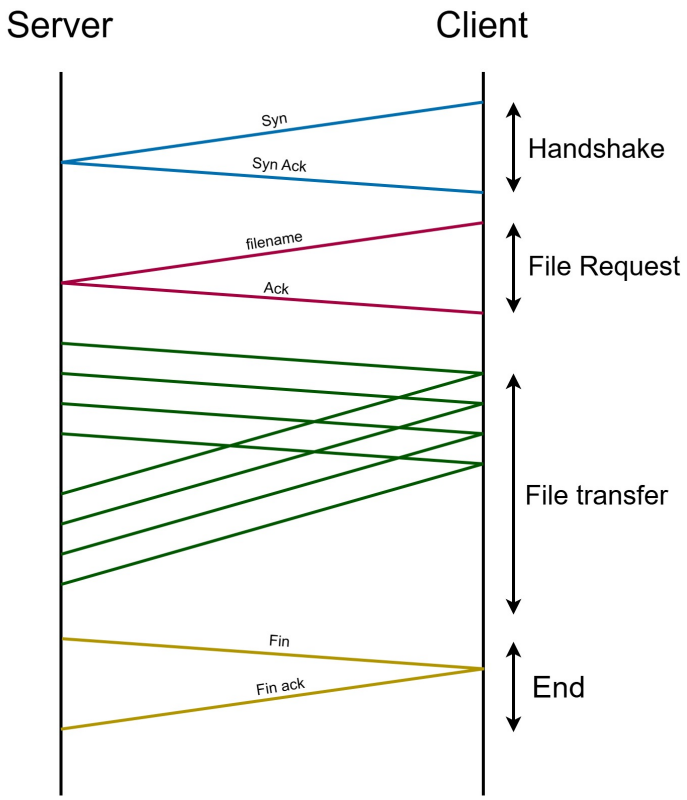


Figura 2: Diagrama del flujo de mensajes con un cliente

3.3. Mecanismo de recuperación

3.3.1. Stop & Wait

El protocolo Stop and wait(SW) es simple: un host envía un paquete y espera su confirmación (ACK). Si no recibe el ACK en un tiempo determinado, lo reenvía. Esto se logra mediante un temporizador.

Un problema potencial ocurre cuando el retardo de la red provoca que el paquete duplicado y el original lleguen casi al mismo tiempo. Para solucionarlo, se emplea una secuencia en módulo 2, alternando los valores entre 0 y 1 (0, 1, 0, 1, ...). El emisor espera un ACK específico y sólo envía el siguiente paquete al recibirlo. De esta forma el protocolo se asegura que lo haya recibido y no avanza hasta confirmarlo, y lo escribe una sola vez.

El protocolo asegura el orden porque el siguiente paquete solo se envía tras la confirmación del anterior. La pérdida se maneja mediante reenvío. Sin embargo, su simplicidad lo hace poco eficiente, ya que entre cada envío debe transcurrir al menos un RTT (tiempo de ida y vuelta).

Tiempo estimado de transferencia sin pérdidas:

$$\text{Tiempo} = \frac{\text{longitud del archivo} \times RTT}{\text{tamaño máximo del paquete}}$$

3.3.2. Go-Back-N

El protocolo GBN utiliza una ventana de paquetes en vuelo y si ha enviado toda la cantidad máxima de paquetes permitidos por la ventana, no enviará más hasta que algún paquete haya sido confirmado por su *ACK*. Por ejemplo, con una ventana de tamaño 3, se envían los paquetes 0, 1 y 2 simultáneamente, y luego espera a recibir confirmación de llegada. Al recibir el *ACK* del paquete 0, se envía el 3, y así sucesivamente.

El receptor espera los paquetes en orden. Si recibe uno fuera de secuencia, lo descarta, y sólo envía el *ACK* si es el paquete con número de secuencia que estaba esperando. Por parte del host emisor, cada vez que se mueve la ventana inicia un temporizador, y si éste expira, se reenvía toda la ventana desde el paquete que no ha sido confirmado. Esta es la desventaja que presenta este protocolo, ya que puede provocar reenvíos innecesarios y aumentar el tráfico de paquetes considerablemente si hay muchos clientes utilizando la misma red.

El orden y la llegada completa se garantizan mediante reenvíos, a costa de la deficiencia en presencia de pérdidas o desorden en la llegada de los paquetes al receptor. Si bien GBN es más veloz que SW al permitir múltiples envíos simultáneos, maneja un flujo mayor de paquetes que pueden llegar a ser descartados.

3.4. Sockets

En esta sección se explica el diseño realizado en este trabajo con respecto a los sockets utilizados.

3.4.1. UDP socket

Para facilitar la interfaz asincronica utilizando la librería estándar de sockets en python, se confecciono un 'wrapper' llamado UDP Socket. Este objeto es utilizado por los sockets de aceptación y de conexión, que se detallan a continuación.

3.4.2. Acceptor socket

Se creó un objeto llamado acceptor socket cuya función principal es el proceso de aceptación de nuevas conexiones entrantes, para las cuales, este socket realiza el *handshake* y crea un socket de conexión, que luego sera utilizado por el protocolo de transferencia de archivos.

Para mantener las conexiones abiertas, se utiliza un mecanismo llamado 'Flow Manager' (detallado en la siguiente sección) el cual se encarga de despachar los mensajes entrantes, que no corresponden a un inicio de conexión (o etapa de *handshake*).

3.4.3. Connection socket

Este socket es el utilizado una vez el cliente ya paso por la etapa de *handshake* y necesita comenzar una transacción, pasando por las etapas de 'file request' y 'file transfer' anteriormente mencionadas.

Cuando el usuario del socket ejecuta la función de recepción, este tendrá dos posibles comportamientos dependiendo de como fue inicializado. En caso de ser el socket de conexión del lado de un cliente, este recibe utilizando el socket udp interno. Alternativamente, si fue inicializado en el servidor, este tendrá una cola de paquetes asociada al 'Flow manager' de la cual leerá los mismos (dicha cola, se menciona en la siguiente sección).

3.5. Conexiones

Para lograr una transferencia de archivos confiable, se considero necesaria, la implementación de conexiones entre los clientes y el servidor. Esto quiere decir que se mantiene un estado de las transacciones de cada cliente en el servidor.

Para lograr el establecimiento de conexiones utilizando un servicio sin conexión, como lo es UDP, se propusieron dos alternativas. Por un lado la creación de sockets de conexión en puertos efimeros, y por otro lado un administrador de flujos exponiendo solo un puerto.

3.5.1. Flow manager

El administrador de flujos (o *flow manager*), es un intermediario entre el socket de aceptación donde esta escuchando el servidor y los 'handlers' de cada cliente. Para las nuevas conexiones este crea una cola (o canal) de paquetes en la cual guardar los futuros paquetes $post - handshake$, y de la cual va a leer el socket de conexión. Cuando se finaliza una conexión este elimina la cola asociada a el flujo terminado. Entonces este administrador actúa como un demultiplexor de paquetes entrantes al puerto del servidor (7532 para este protocolo), para que luego, los 'handlers' (en este caso tareas asincronicas) reciban el paquete desde su respectiva cola. Este enfoque permite la comunicación con múltiples clientes con un solo puerto expuesto.

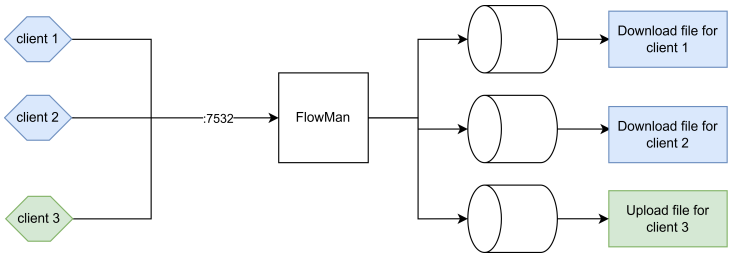


Figura 3: Diagrama de conexiones con flow manager

3.5.2. Puertos efímeros

La segunda opción propuesta, fue abrir puertos efimeros para cada conexión con un cliente, y durante el *handshake*, avisarle al cliente cual es el nuevo puerto de comunicación con el servidor. Esto permitiría evitar el cuello de botella de la anterior solución, sin embargo agregan nuevas complicaciones, una de ellas siendo que a nivel servidor, se deben mantener varios puertos abiertos, uno para cada cliente conectado. Otra de las complicaciones, aunque menos importante, es la capacidad reducida de depuración de este método, ya que hace mas difícil hacer un seguimiento de los mensajes correspondientes a cada conexión.

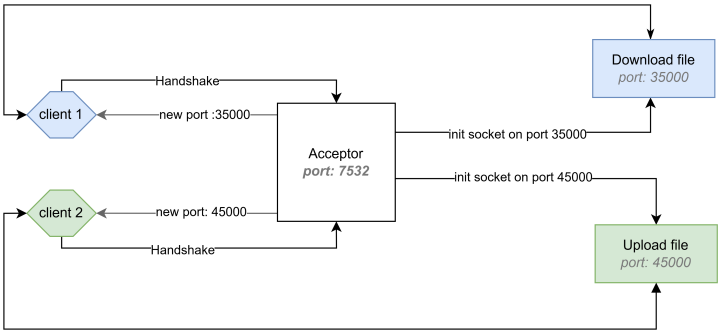


Figura 4: Diagrama del conexiones con puertos efimeros

Al compararlos meticulosamente, se decidió tomar la primera opción propuesta, es decir, el administrador de flujos para el manejo de las conexiones.

3.6. Arquitectura

3.6.1. Modelo de concurrencia

Para presentar un funcionamiento concurrente al momento de recibir múltiples peticiones simultáneas sobre el servicio de transferencia de datos, se ha utilizado un modelo de concurrencia basado en asincronismo, específicamente utilizando la librería asyncio. Esta permite correr tareas asincronicas y hacer 'polling' entre ellas. Un ejemplo del uso de este modelo, es la ejecución de la funcionalidad aceptadora de nuevos clientes y la funcionalidad de transferencias en curso concurrentemente.

3.6.2. Servidor

El servidor se inicia indicando en su directiva de ejecución el tipo de protocolo a utilizar para la transferencia de archivos, si algún cliente intenta conectarse con un protocolo diferente, el servidor le avisa y termina la conexión del cliente, y continúa esperando nuevas conexiones. También por opción dentro de la directiva de ejecución se establece el directorio donde se almacenan los datos cargados por los clientes, como también los archivos disponibles para ser descargados por ellos. También como directiva se establece la dirección IP y el puerto en el que el server queda escuchando la conexión a distintos clientes que quieran conectarse.

3.6.3. Cliente

El cliente puede ejecutarse de dos formas, en modalidad **upload** y **download**. A su vez puede seleccionarse el protocolo mediante una opción en la directiva de ejecución, el nombre de archivo que desea descargar y el directorio donde desea descargar dicho archivo (si no existe se crea), o el archivo a cargar y el directorio donde se encuentra (si no existe el archivo, finaliza en error). A su vez por directiva también se especifica la IP y puerto del servidor al que se ha de conectarse para comenzar la transferencia.

4 Pruebas

Se llevaron a cabo una serie de pruebas automatizadas en mininet, las mismas crean una topología en base a un archivo de configuración en formato 'json', y ejecutan un escenario de transferencias con una cantidad predefinida de clientes y de perdida de paquetes (dado por la configuración).

Al finalizar todas las transferencias, se registran los tiempos que demoraron, y se ejecuta un script que corrobora que los archivos transferidos sean idénticos, utilizando 'SHA' hashing.

4.1. Pruebas aumentando la probabilidad de perdida de paquetes

Para el siguiente experimento, se lanzaron varios clientes concurrentes (1, 3 y 5), y se midieron sus tiempos de transferencia, luego se aumento el 'packet loss' al doble del anterior. Luego se utilizaron las mediciones para calcular su media, mediana y desviación estándar.

protocolo	perdida [%]	operación	media [s]	std [s]	mediana [s]
GBN	10	download	18.81	7.88	17.28
GBN	10	upload	6.99	2.52	6.18
GBN	20	download	18.00	7.07	18.00
GBN	20	upload	9.00	1.41	9.00
GBN	40	download	62.83	20.50	70.50
GBN	40	upload	23.12	7.99	22.00
SW	10	download	34.13	14.39	28.00
SW	10	upload	14.16	7.68	12.02
SW	20	download	52.50	36.06	52.50
SW	20	upload	21.50	2.78	22.00
SW	40	download	152.00	1.63	152.00
SW	40	upload	37.80	11.05	40.00

4.2. Pruebas de tiempos en base a la cantidad de clientes

Para el siguiente conjunto de métricas, al igual que en la anterior prueba, se utilizaron clientes incrementales (1, 3 y finalmente 5), y también se aumento la perdida gradualmente. Sin embargo, en este caso se demuestra el desempeño de ambos protocolos en base al **aumento de clientes** y el aumento de perdida.

protocolo	perdida [%]	clientes	operación	mediciones	media [s]	std [s]
GBN	10	1	upload	4.00	6.53	1.18
GBN	40	1	upload	2.00	17.50	7.78
GBN	10	3	download	3.00	8.85	3.60
GBN	40	3	download	2.00	50.00	39.60
GBN	10	3	upload	5.00	6.38	2.09
GBN	40	3	upload	3.00	31.00	6.24
GBN	10	5	download	10.00	21.79	6.08
GBN	40	5	download	4.00	69.25	3.59
GBN	10	5	upload	7.00	7.68	3.35
GBN	40	5	upload	3.00	19.00	2.00
SW	10	1	upload	6.00	15.04	12.04
SW	40	1	upload	2.00	32.00	16.97
SW	10	3	download	4.00	21.21	7.89
SW	10	3	upload	7.00	15.80	7.18
SW	10	5	download	10.00	39.30	13.21
SW	40	5	download	4.00	152.00	1.63
SW	10	5	upload	7.00	11.78	1.99
SW	40	5	upload	3.00	41.67	6.66

5 Preguntas a responder

5.1. Arquitectura Cliente-Servidor

La arquitectura del modelo cliente-servidor es una estructura de aplicación distribuida centralizada, que reparte tareas/cargas entre los proveedores de un recurso o servicio (servidores) y los solicitantes de servicios, en este caso clientes. En la arquitectura cliente-servidor, cuando un cliente solicita un servicio o información mediante la red a un servidor, este último acepta la solicitud y devuelve los paquetes de datos solicitados al cliente.

Los clientes no comparten sus recursos sino que los recursos se procesan y la información se trabaja en el servidor, y los cambios que sean requeridos en actualizar la información (si fuera del estilo el funcionamiento), es informada a través del servidor a cada uno de los clientes que estén conectados.

El modelo cliente-servidor centraliza los recursos en servidores, lo que mejora el control y la seguridad, ofrece flexibilidad a los usuarios y depende de una red sólida para garantizar escalabilidad y rendimiento. Aunque conlleva ciertos gastos, que puede ser redistribuido tal carga utilizando múltiples servidores, este esquema sigue siendo de los más utilizados.

5.2. Protocolos de capa de aplicación

Un protocolo de capa de aplicación se define cómo los procesos de una aplicación, ejecutándose en diferentes sistemas finales e intercambiando mensajes. Entre todas las funcionalidades, presenta principalmente:

- Tipos de mensajes: Por ejemplo, mensajes de solicitud (request) y respuesta (response).
- Sintaxis de los mensajes: La estructura de los campos presentes en los mensajes y cómo se delimitan.
- Semántica de los campos: El significado de la información contenida en cada campo.
- Reglas de comunicación: Cuándo y cómo un proceso envía mensajes y cómo responde a ellos, siguiendo ciertos criterios establecidos.

Además, algunos protocolos son públicos (como HTTP, estandarizado en RFCs), mientras que otros son privados.

Un protocolo de aplicación (ej. HTTP, DASH) es solo una parte de una aplicación de red (ej. navegador web, Netflix), encargándose de la comunicación estructurada entre cliente y servidor.

Estos protocolos estandarizan la interacción entre aplicaciones, garantizando compatibilidad y eficiencia en la comunicación.

5.3. Protocolo de aplicación desarrollado

El protocolo de aplicación desarrollado, es un protocolo de transferencia confiable de archivos utilizando como transporte el protocolo UDP. Este cuenta con dos algoritmos de recuperación ante problemas inherentes a las redes de computadoras, como lo es la perdida de paquetes (en especial si se utiliza un protocolo de transporte no confiable). Dicho protocolo se encuentra detallado en la sección de implementación del presente informe.

5.4. TCP vs UDP

La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuando es apropiado utilizar cada uno?

TCP (Transmission Control Protocol)

El protocolo TCP provee los siguientes servicios:

- **Orientado a conexión (connection-oriented):** Establece una conexión lógica entre el emisor y el receptor antes de enviar datos.
- **Entrega confiable:** Garantiza que los datos lleguen sin errores, en orden y sin pérdidas mediante mecanismos de:
  - **ACK (acknowledgments):** Confirmación de recepción.
  - **Retransmisión:** Si no se recibe ACK, el paquete se reenvía.
  - **Control de secuencia:** Ordena los segmentos recibidos.
- **Control de flujo (flow control):** Evita saturar al receptor ajustando la tasa de envío.
- **Control de congestión:** Reduce la tasa de transmisión si detecta congestión en la red.

Características principales

- **Overhead alto:** Debido a los mecanismos de confiabilidad.
- **Latencia mayor:** Por el establecimiento de conexión y retransmisiones.
- **Full-duplex:** Permite comunicación bidireccional simultánea.

Casos de uso apropiados

- Aplicaciones que requieren fiabilidad y integridad de datos, como:
  - Navegación web (HTTP/HTTPS).
  - Transferencia de archivos (FTP).
  - Correo electrónico (SMTP, IMAP).
  - Conexiones remotas (SSH).

Protocolo UDP (User Datagram Protocol)

Servicios provistos

- **Sin conexión (connectionless):** No establece una conexión previa.
- **No confiable:** No garantiza entrega, orden ni duplicación de paquetes.
- **Sin control de flujo ni congestión:** El emisor envía datos a la máxima velocidad posible.

Características principales

- **Overhead bajo:** No tiene ACK, retransmisiones ni control de flujo.
- **Baja latencia:** Ideal para aplicaciones en tiempo real.
- **Mensajes independientes:** Cada datagrama se maneja por separado.

Casos de uso apropiados

- Aplicaciones que priorizan velocidad y baja latencia sobre la fiabilidad, como:
  - Streaming de video/audio (VoIP, Zoom, Netflix).
  - Juegos en línea (latencia crítica).
  - DNS (consultas rápidas).
  - Broadcast/multicast (envío a múltiples receptores).

Característica	**TCP**	**UDP**
Orientación	Conexión	Sin conexión
Fiabilidad	Alta	Baja
Control de flujo	Sí	No
Control de congestión	Sí	No
Overhead	Alto	Bajo
Latencia	Mayor	Menor

Cuadro 1: Resumen comparativo TCP vs UDP

6 Conclusión

Se ha llegado a la conclusión que es posible realizar protocolos de aplicación para transferencia de datos **confiable**, que puede incluso ser más estricto a nivel de recuperación, ajustando ciertos parámetros como el timeout al no detectar mensajes de **ACK** o el número de intentos en reenviar paquetes, en caso de que se presente un porcentaje de pérdida de datos elevado.

Por otro lado, en este proyecto se realizó dos tipos de protocolos, con distintas características. Por un lado el protocolo **Stop and Wait**, si bien como fue explicado en el informe, es un proceso mas lento, ya que requiere la confirmación de recepción por cada paquete, utiliza menos recursos que el otro protocolo, y además si un número elevado de clientes están realizando múltiples transacciones simultáneamente, el servidor es capaz de soportar más conexiones a la vez y a su vez genera menor congestión de paquetes en la red.

Sin embargo el protocolo **Go Back N**, en términos de velocidad es ampliamente superior, más aún si no hay un porcentaje elevado de pérdida de paquetes. Pero en el caso de tener perdida de paquetes o éstos llegan de forma desordenada, se puede llegar a enviar una cantidad de paquetes considerable, que puede llegar a afectar el funcionamiento del servidor y también en la congestión de la red y en nuestro caso en las colas que maneja el Flow Manager, si son muchos clientes utilizando el servicio.

7 Referencias

1. James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed., Pearson, 2021.
2. Mininet. (n.d.). Mininet walkthrough. https://mininet.org/walkthrough/
3. die.net. (n.d.). iperf(1) - Linux man page. https://linux.die.net/man/1/iperf

8 Anexo

8.1. Fragmentación IPv4

Para este inciso se confeccionó una topología diferente a la del enunciado, ya que se descubrió que no era posible realizar el análisis con tres switches intermedios entre dos hosts, debido a que los mismos no poseen la capacidad de fragmentar cuando el largo de un paquete es mayor al MTU del enlace.

Por esta razón se llegó a la conclusión de que para que dicha topología funcione correctamente para el análisis, está debe contener al menos un router intermedio. Esto último se logró, ajustando la configuración del switch del medio, para que sea capaz de hacer 'IP-forwarding', y reduciendo el MTU de uno de sus enlaces (en este caso, el enlace con dirección a $h2$).

A su vez, se debió configurar ambos hosts, de manera que los paquetes enviados tengan la *flag* de 'Do not Fragment' (DF) en 0. Y para el caso de prueba con flujo TCP, fue necesario deshabilitar la opción, de 'MTU Probing', ya que esta fuerza que dicho protocolo pruebe y ajuste dinámicamente el tamaño de los paquetes para no tener fragmentación.

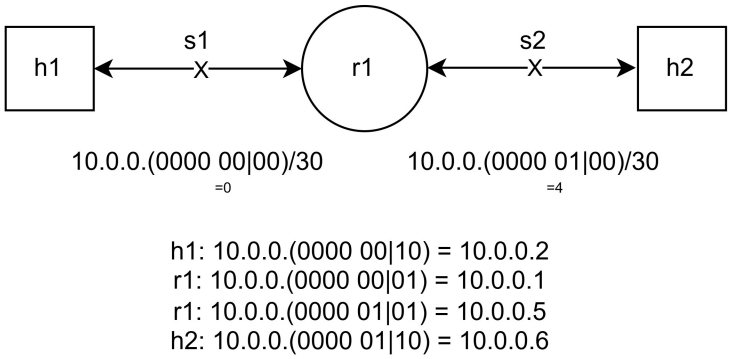


Figura 5: Diagrama de la topología implementada para fragmentación

Se ha establecido una asignación de IPs justas y necesarias para el proyecto, utilizando subnetting ($\#IPs = 1$ host + 1 router + 1 IP red + IP broadcast para ambos lados). Ademas, se definió el MTU de la interfaz entre $r1$ y $s2$ con un valor de $600B$, mientras que el resto de interfaces se dejaron con el valor por defecto, que es de $1500B$.

De esta manera, si se toma al segundo host ($h2$) como servidor y al primero ($h1$) como cliente, se puede generar utilizando la herramienta **'iperf'** un flujo de mensajes, dándole un largo especifico a cada uno (en este caso se utilizo $1400B$ que es menor al MTU de la interfaz de entrada al router, y mayor al MTU de la interfaz de salida del mismo), y observar alguna de las interfaces previo a entrar al router y otra, luego de salir del router. Es en esta ultima etapa donde se podrá observar la fragmentación hecha por el router.

A continuación se anexaron imágenes donde se analiza el flujo de los paquetes a través de esta topología, utilizando como herramienta a **Wireshark**.

8.1.1. Escenario de fragmentación con protocolo TCP

En esta sección se llevara a cabo un análisis del fenómeno de fragmentación en IPv4, y como se comporta TCP al momento de la perdida de un fragmento. Para esto se utilizaran dos capturadores de Wireshark en interfaces $s1 - eth1$ (llegando al router) y $s2 - eth1$ (saliendo del router).

Captura de paquetes entrantes al router

En la siguiente captura, se puede apreciar que $h1$ envía el paquete numero 7, (que corresponde al primer paquete con información luego del handshake), con número de secuencia 61. Se observa el header TCP (destacado en azul) y la data subsiguiente.

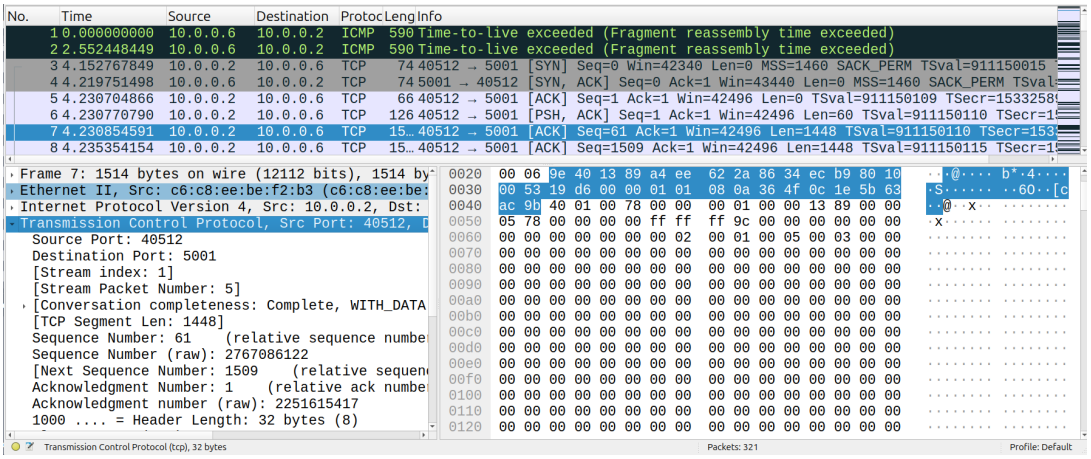


Figura 6: Envió de paquete 7 con TCP (Emisor)

Es importante notar que el largo de dicho paquete #7, es $1514B$ cuando el MTU de las interfaces por las que paso hasta ahora, fueron todas de $1500B$, sin embargo estos $14B$ excedentes pertenecen al header del protocolo de capa de enlace utilizado (en este caso, Ethernet II). También, la razón por la que sin contar dichos $14B$ el total es $1500B$ en lugar de los $1400B$ pre-definidos en la opción de 'iperf', es porque a este se le agregan $32B$ del header TCP (ya que cuenta con $12B$ de opciones), $20B$ del header IPv4 y $48B$ de padding que agrega TCP al segmento para llegar exactamente al MTU.

En síntesis, el largo final en capturado es:

- $largo_{encable} = largo_{data} + largo_{padding} + largo_{TCPHeader} + largo_{IPv4Header} + largo_{Eth2Header}$
- $1514B = 1400B + 48B + 32B + 20B + 14B$

Una ultima observación antes de analizar lo ocurrido luego de pasar por el router, es que hay algunos paquetes con largo $2962B$, este fenómeno se puede adjudicar a 'segment agregation', es decir, acumulación de dos o mas segmentos en uno, esto puede ocurrir en el contexto de 'TCP segmentation offload' que es una optimización de rendimiento, utilizada en sistemas operativos modernos y tarjetas de interfaz de red (NICs).

Captura de paquetes salientes del router

En la siguiente captura los paquetes recibidos número 6, 7 y 8 son los fragmentos correspondientes al paquete 7 enviado en la captura anterior, los contenidos de data en ambas capturas coinciden, y además su número de secuencia 61. Se puede ver que Wireshark marca con puntos a la izquierda del número de paquete, los fragmentos re-ensamblados (los fragmentos son #6, #7 y #8, y los divide en dos paquetes de $610B$ y uno de $362B$).

Como observación y aclaración adicional, por ejemplo en esta transmisión de paquetes mediante TCP, donde el emisor envía $1514B$. Se observa que luego de fragmentarse, a pesar de haber configurado el MTU de la interfaz con un valor de $600B$, en las capturas del receptor se dividen en 3 fragmentos, donde dos paquetes son de tamaño $610B$ y un paquete de $362B$.

Esto se debe a que el largo máximo de data se puede calcular como $l_{max} = \lfloor \frac{MTU - l_{IPv4Header}}{8} \rfloor \times 8 = 576B$ y a esto se le agrega largo del header IPv4 resultando en un total de $596B$, tamaño que no supera el MTU de dicha interfaz, pero se agregan en la capa de enlace luego de fragmentar cada paquete, $14B$ de header Ethernet II (resultando en $610B$ para los primeros dos fragmentos).

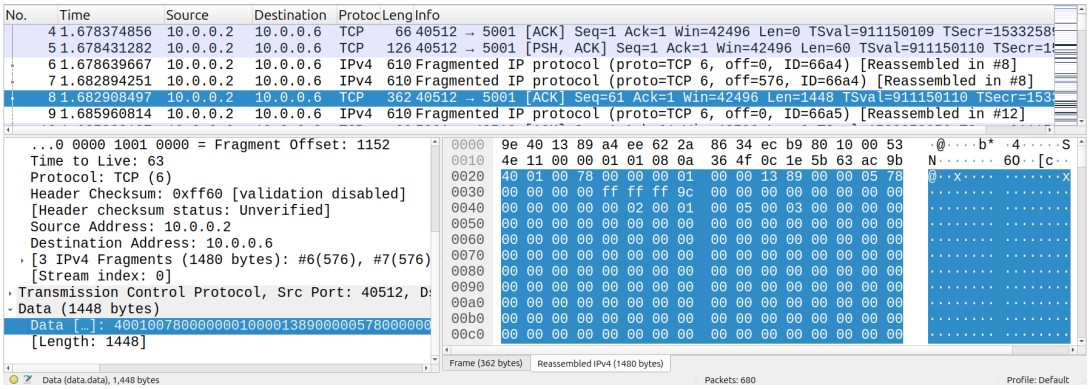


Figura 7: Recepción de fragmentos 6, 7 y 8 con TCP (Receptor)

8.1.2. Escenario de retransmisión con TCP

En esta sección se llevara a cabo un análisis del funcionamiento de retransmisión de paquetes, debido a que un fragmento se pierde en el receptor, por lo tanto el protocolo TCP pide su recuperación al emisor, y éste realiza una retransmisión del paquete completo. Se utilizaran dos capturadores de Wireshark en interfaces $s1 - eth1$ (llegando al router) y $s2 - eth1$ (saliendo del router).

Captura de paquetes entrantes del router

En las siguientes capturas puede observarse que el paquete 12 se pierde, o al menos uno de sus fragmentos, y se retransmite de nuevo como el paquete número 29 *(notar que tienen el mismo numero de secuencia, 1509)*. A su vez se corroboro que ambos paquetes tengan exactamente el mismo contenido en la sección de data.

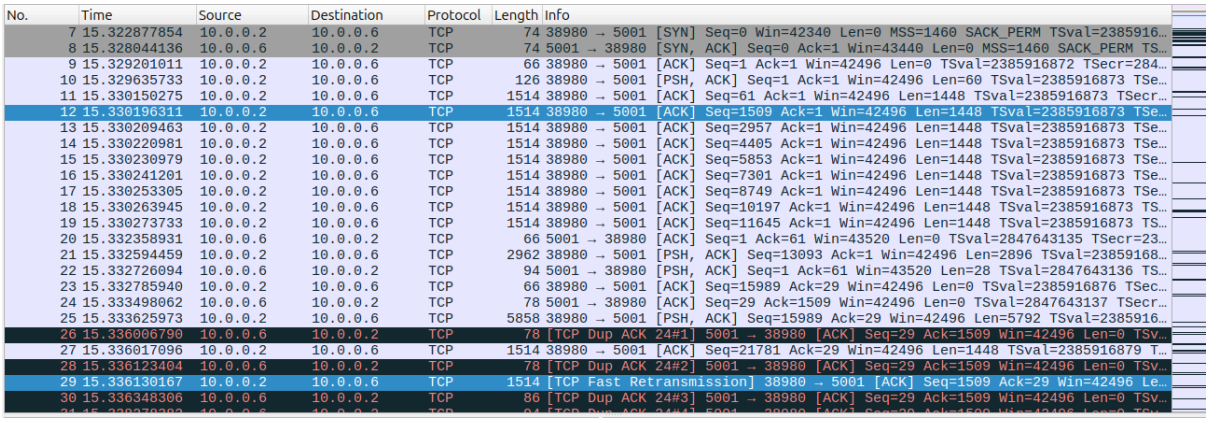


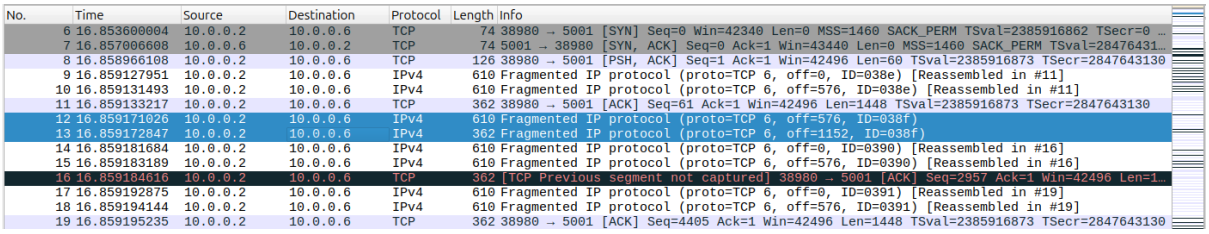
Figura 8: Perdida y retransmisión de paquete 12 con TCP (Emisor)

Captura de paquetes salientes del router

En las siguientes capturas de la interfaz saliente del router, representando al receptor, puede observarse que se pierden fragmentos y se vuelven a capturar en su retransmisión.

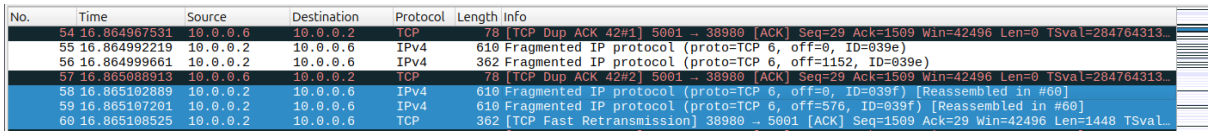
En la primer captura los paquetes número 12 y 13 no tienen información sobre re-ensamblado en su recepción, por lo tanto eso quiere decir que el primer o segundo fragmento se ha perdido (el paquete 13 tiene tamaño 362 bytes que es el último de los tres fragmentos).

En la segunda captura se observa un tercer ACK al numero de secuencia 1509 seguido de un 'TCP FAST RETRANSMIT', y se presentan los paquetes 58, 59 y 60. Estos fragmentos corresponden al paquete retransmitido ya que su número de secuencia es 1509, el observado en el paquete enviado por el emisor y posteriormente retransmitido.



No.	Time	Source	Destination	Protocol	Length	Info
6	16.853600004	10.0.0.2	10.0.0.6	TCP	74	38980 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=2385916862 TSecr=0 …
7	16.857006608	10.0.0.6	10.0.0.2	TCP	74	5001 → 38980 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=28476431…
8	16.858966108	10.0.0.2	10.0.0.6	TCP	126	38980 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=60 TSval=2385916873 TSecr=2847643130
9	16.859127951	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=038e) [Reassembled in #11]
10	16.859131493	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=038e) [Reassembled in #11]
11	16.859133217	10.0.0.2	10.0.0.6	TCP	362	38980 → 5001 [ACK] Seq=61 Ack=1 Win=42496 Len=1448 TSval=2385916873 TSecr=2847643130
12	16.859171026	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=038f)
13	16.859172847	10.0.0.2	10.0.0.6	IPv4	362	Fragmented IP protocol (proto=TCP 6, off=1152, ID=038f)
14	16.859181684	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=0390) [Reassembled in #16]
15	16.859183189	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=0390) [Reassembled in #16]
16	16.859184616	10.0.0.2	10.0.0.6	TCP	362	[TCP Previous segment not captured] 38980 → 5001 [ACK] Seq=2957 Ack=1 Win=42496 Len=1…
17	16.859192875	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=0391) [Reassembled in #19]
18	16.859194144	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=0391) [Reassembled in #19]
19	16.859195235	10.0.0.2	10.0.0.6	TCP	362	38980 → 5001 [ACK] Seq=4405 Ack=1 Win=42496 Len=1448 TSval=2385916873 TSecr=2847643130

Figura 9: Perdida fragmento del paquete *seq=1509* con TCP (Receptor)



No.	Time	Source	Destination	Protocol	Length	Info
54	16.864967531	10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 42#1] 5001 → 38980 [ACK] Seq=29 Ack=1509 Win=42496 Len=0 TSval=284764313…
55	16.864992219	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=039e)
56	16.864999661	10.0.0.2	10.0.0.6	IPv4	362	Fragmented IP protocol (proto=TCP 6, off=1152, ID=039e)
57	16.865088913	10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 42#2] 5001 → 38980 [ACK] Seq=29 Ack=1509 Win=42496 Len=0 TSval=284764313…
58	16.865102889	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=039f) [Reassembled in #60]
59	16.865107201	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=039f) [Reassembled in #60]
60	16.865108525	10.0.0.2	10.0.0.6	TCP	362	[TCP Fast Retransmission] 38980 → 5001 [ACK] Seq=1509 Ack=29 Win=42496 Len=1448 TSval…

Figura 10: Retransmisión del paquete *seq=1509* con TCP (Receptor)

8.1.3. Escenario de fragmentación con UDP

En esta sección se llevara a cabo un análisis sobre la fragmentación de paquetes utilizando el protocolo UDP. A diferencia del protocolo anterior, este protocolo no cuenta con la retransmisión de paquetes por pérdida de fragmentos, por lo tanto no se analizará la retransmisión. Se utilizaran dos capturadores de Wireshark en interfaces $s1 - eth1$ (llegando al router) y $s2 - eth1$ (saliendo del router).

Captura de paquetes entrantes del router

En la siguiente captura del lado del emisor se puede observar el paquete 2, y su sección de data correspondiente, la cual se encuentra seleccionada para luego poder realizar una comparación con el paquete recibido del otro lado.

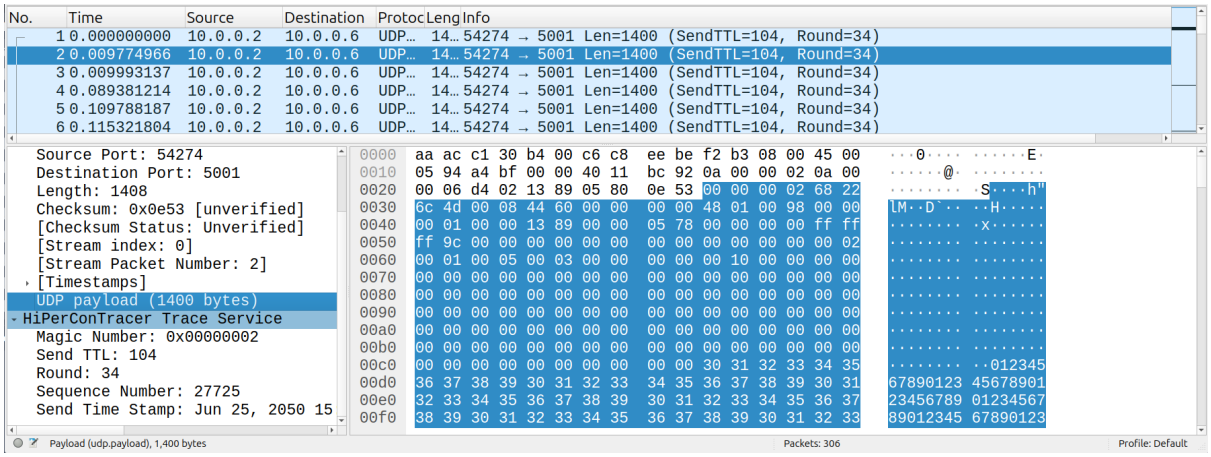


Figura 11: Envío de paquete 2 con UDP (Emisor)

Captura de paquetes salientes del router

Se observa en la siguiente captura de la interfaz saliente del router, los paquetes 3, 4 y 5 (donde Wireshark muestra tres puntos a la izquierda de cada fragmento indicando que corresponden al mismo paquete) son los fragmentos del paquete 2 enviado por h1. Esto se puede corroborar viendo que coinciden exactamente los bytes de data, que se muestran seleccionados en ambas capturas. A su vez vale la pena mencionar, que todo fragmento intermedio tiene la *flag* 'More Fragments' encendida, no así, el ultimo fragmento para completar el paquete.

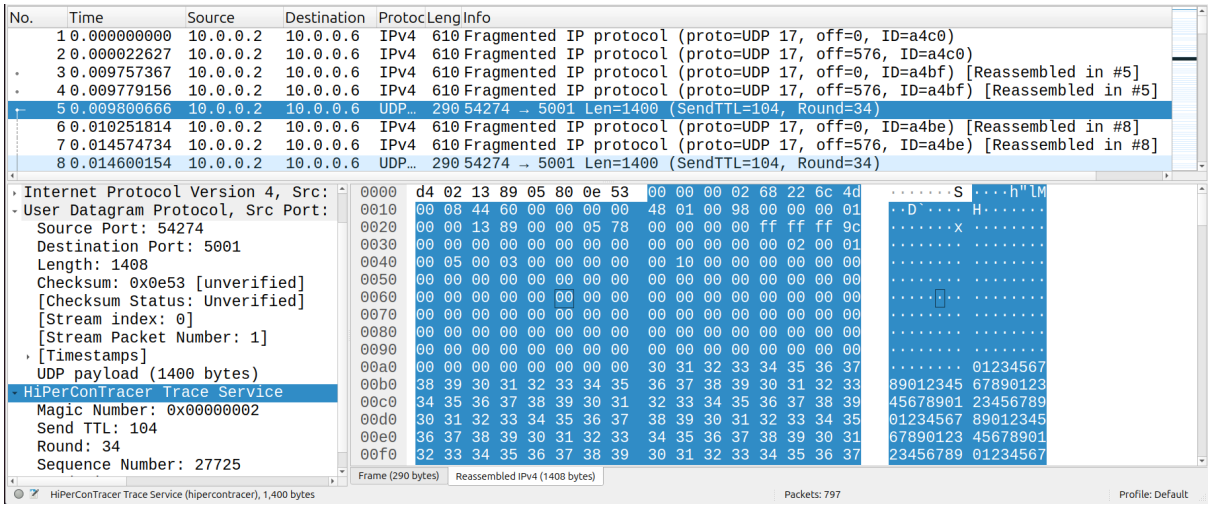


Figura 12: Recepción de fragmentos 3, 4 y 5 con UDP (Receptor)

Como análisis adicional, utilizando la misma captura, el paquete 1 enviado desde h1, luego de ser enviado y pasado por el router debería ser fragmentado en tres paquetes debido a su tamaño, sin embargo se observan solo los dos primeros fragmentos, lo que indica que el tercer y ultimo fragmento se perdió en la transmisión, por lo que dicho paquete 1, no pudo ser re-ensamblado (entonces Wireshark no marca a los fragmentos como [**Reassembled in ...**]), ni tampoco sera retransmitido debido a la naturaleza del protocolo.

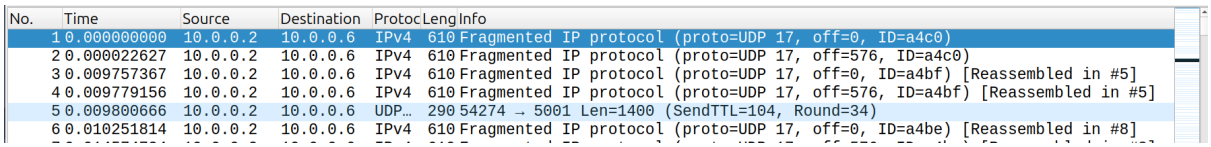


Figura 13: Descarte de paquete, por pérdida de fragmento con UDP