Trabajo Práctico N°1

Introduccion a los Sistemas Distribuidos [TA049]

Integrantes:

- 108091, Martin Morilla
- 107552, Iñaki Llorens
- 108313, Rafael Ortegano

Fecha de entrega: 06 de Mayo de 2025

Índice

1. Introducción

El presente trabajo práctico tiene como objetivo el desarrollo de una aplicación de red basada en el modelo *cliente-servidor*, que permita la transferencia de archivos binarios entre procesos remotos. Para ello, se abordarán conceptos fundamentales sobre la comunicación entre procesos en red y el modelo de servicios que la capa de transporte brinda a la capa de aplicación.

La propuesta implica el diseño e implementación de un protocolo de aplicación que soporte las operaciones de:

- `UPLOAD`: El cliente sube archivos al servidor.
- `DOWNLOAD`: El cliente descarga archivos que se encuentran en el servidor.

Para esto se utilizara el protocolo *user datagram protcol (UDP)* como medio de transporte. Dado que UDP no garantiza por sí solo una entrega confiable, se requerirá implementar mecanismos de *reliable data transfer (RDT)*, utilizando tanto el protocolo *Stop and Wait (SAW)* como el protocolo *Go-Back-N (GBN)*.

Con el objetivo de validar la robustez de las soluciones implementadas, se introducirán condiciones de red adversas, tales como la pérdida de paquetes. Para tal fin, se empleará la herramienta *Mininet*, la cual permite simular estos entornos de prueba.

2. Hipótesis y suposiciones realizadas

- Los paquetes no vienen corruptos ya que UDP provee dicho servicio de validación.
- Se supone que la red no posee amenazas de seguridad.
- Se asume que los datos transmitidos no requieren cifrado ni compresión
- La red subyacente puede perder, duplicar o reordenar paquetes.
- Los paquetes tienen un tamaño máximo de 4096 bytes.

3. Implementación

3.1. Arquitectura general

Basandonos en el libro *Computer Networking: A Top-Down Approach: 7th Edition* podemos visualizar las siguientes imagenes:

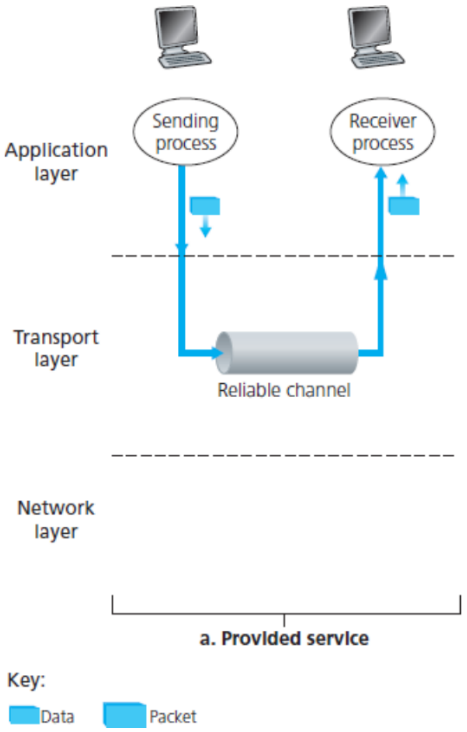


Figura 1: Reliable data transfer

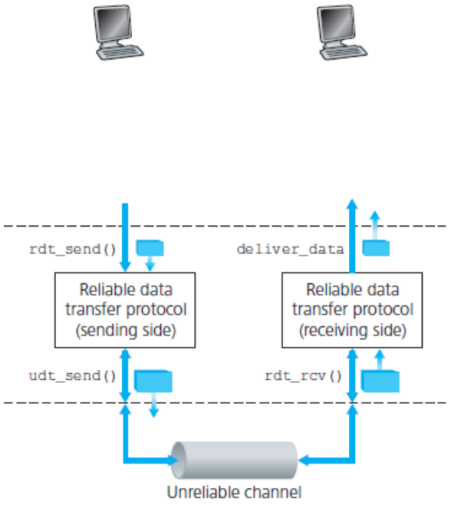


Figura 2: Reliable data transfer

En parte, el objetivo central esta en implementar los metodos *rdt_send()* y *rdt_recv()* los cuales estan asociados a los protocolos mencionados en la introduccion (GBN y SAW). De esta forma proveemos tanto al *sender* como al *receiver* una forma de **enviar** y **recibir** paquetes a traves de la red en orden y de manera confiable, ya que recordemos que nuestra capa de transporte es UDP.

En la capa de aplicacion, tanto el *servidor* como el *cliente* son los encargados de manejar la logica pertinente para el manejo de archivos, posibles errores que puedan surgir en ese proceso, y de llamar al metodo correspondiente (*rdt_send()*, *rdt_recv()*), dependiendo la operacion (`UPLOAD`, `DOWNLOAD`) que se este ejecutando en ese momento.

- Servidor: Podemos comenzar planteando la solucion que usaria cualquier programador para crear un servidor concurrente. El metodo *server_socket.accept()* de la libreria *socket* de *Python* extrae de forma sincrónica la primera solicitud de conexión pendiente de la cola de solicitudes de conexión del socket de escucha y, a continuación, crea y devuelve un nuevo Socket, es decir que por cada cliente que se conecta al servidor se obtiene un "client_socket"para poder enviar y recibir paquetes del cliente correspondiente.

```
        client_socket, addr = server_socket.accept()

```

  En nuestro caso, como UDP es un protocolo sin conexión (connectionless) simplemente se envían y reciben datagramas (paquetes individuales), entonces debido a que todos los paquetes se reciben atraves del mismo socket en principio puede resultar un tanto dificil el manejo de paquetes del lado del servidor. Debido a eso decidimos crear la siguiente estructura que abstrae cada conexion con el cliente.

```
    class StreamWrapper:
    def __init__(self, socket, queue):
        self.socket = socket
        self.queue = queue
        self.socket.settimeout(TIMEOUT_SOCKET)

    def receive(self):
        if self.queue is None:
            data, _address = self.socket.recvfrom(PACKET_SIZE)
            return Packet.from_bytes(data)
        else:
            return self.queue.get(True, TIMEOUT_QUEUE)

    def send_to(self, bytes, address):
        self.socket.sendto(bytes, address)

    def close(self):
        self.socket.close()

    def enqueue(self, packet):
        self.queue.put(packet)

```

  Lo importante del lado del servidor es prestar atencion a la cola de mensajes que se tiene como campo en la estructura *StreamWrapper*. A medida que se conecten nuevos clientes, el servidor ira creando nuevas conexiones (threads) e ira encolando mensajes en la queue de cada cliente, de esta forma logramos "simular" un

client_socket, ya que cada cliente tiene su *StreamWrapper*. Luego, a medida que avance el flujo del programa para cada thread, cada "ClientHandler" en el servidor, utilizara los metodos correspondientes *rdt_send()* y *rdt_recv()*, dependiendo de la operacion que el cliente haya enviado a traves de la red.

```
    while True:
            try:
                data, addr = self.sock.recvfrom(PACKET_SIZE)
                packet = Packet.from_bytes(data)
                if packet.is_syn():
                    filename_length = packet.get_payload()[0]
                    filename = packet.get_payload()[1:
filename_length + 1].decode('utf-8')
                    handler = ClientHandler(
                        addr,
                        packet.sequence_number,
                        self.protocol,
                        self.logger,
                        filename,
                        packet.is_download(),
                        self.storage_dir
                        )
                    self.client_handlers[addr] = handler
                    self.client_handlers[addr].start()
                else:
                    if(addr in self.client_handlers):
                        self.client_handlers[addr].enqueue(packet)
            except socket.timeout:
                continue

```

Listing 1: Manejo de conexiones en el servidor

- Cliente: En este caso, se utilizara la misma estructura *StreamWrapper* mencionada anteriormente, pero en este caso, el cliente no utilizara una cola de mensajes para manejar los paquetes, si no que directamente leera del socket ya que es una unica conexion. Por otro lado, al igual que el servidor, dependiendo de la operacion que quiera realizar (`UPLOAD`, `DOWNLOAD`), utilizara los metodos correspondientes de RDT.

Cabe destacar que, en esta implementacion, el metodo *rdt_send()* envia la data de a *chunks*, es por eso que entonces el metodo *rdt_recv()* nos devolvera *chunks*. Es decir que no se le pasa el archivo por completo al metodo *rdt_send()*, como podria pasar en otras implementaciones

3.2. Handshake

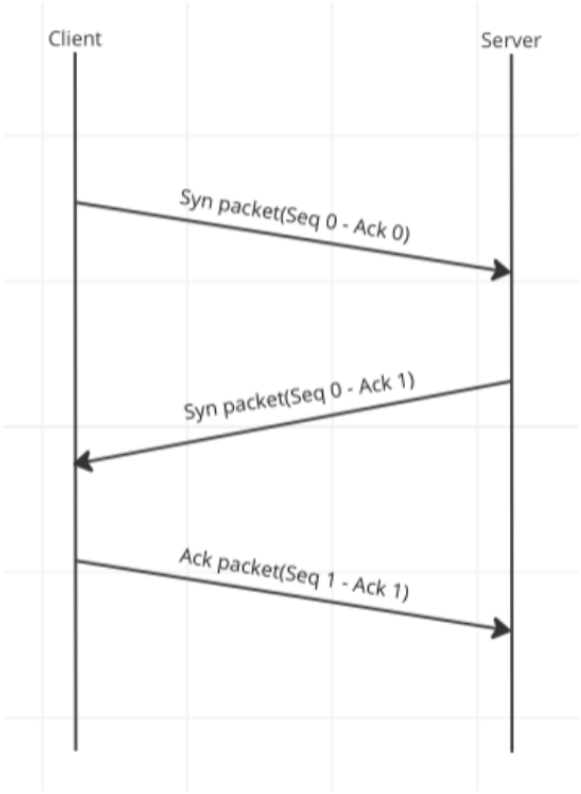


Figura 3: Positive handshake

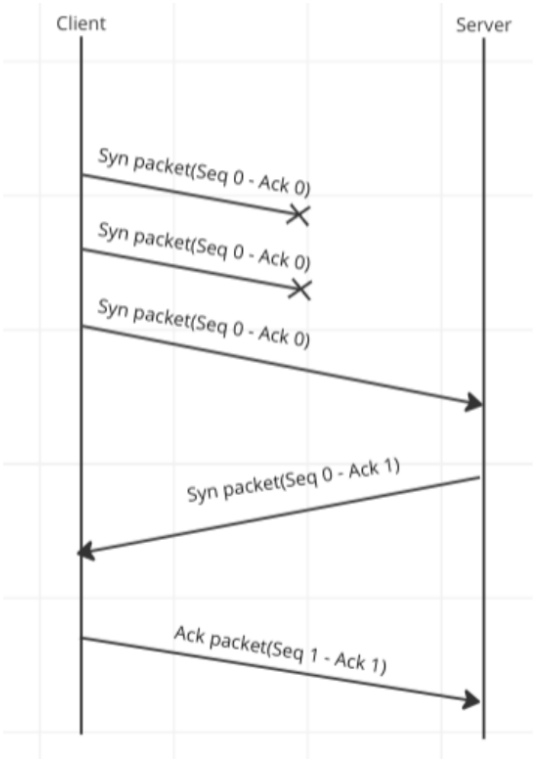


Figura 4: Perdida del paquete SYN del lado del cliente

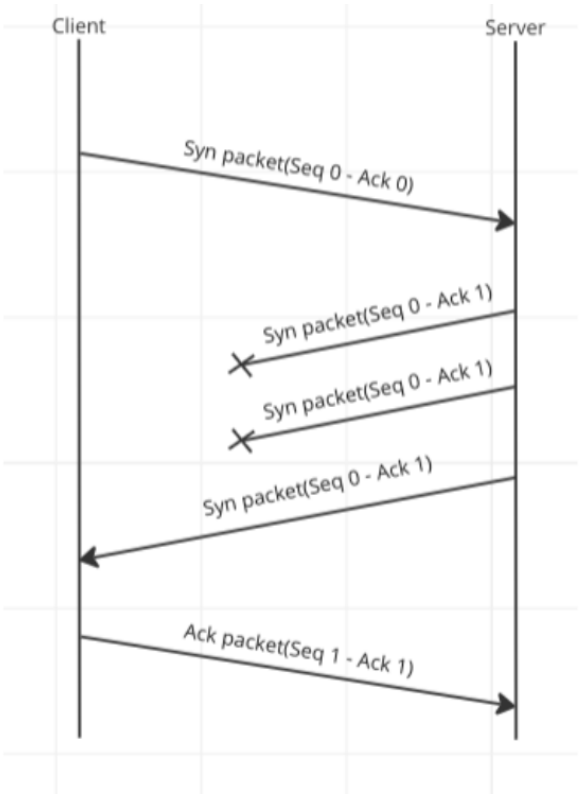


Figura 5: Perdida del paquete SYN del lado del servidor

3.3. Paquete

Cada paquete transmitido en la red sigue una estructura de tamaño fijo, compuesta por los siguientes campos:

- **Sequence Number (Número de secuencia)**: campo de 4 bytes que identifica de manera única cada paquete enviado. Este valor permite mantener el orden correcto de los datos en el receptor.
- **Acknowledgment Number (Número de reconocimiento)**: campo de 4 bytes que indica el número de secuencia del próximo paquete que el receptor espera recibir. Este número permite confirmar la recepción correcta de los datos y detectar posibles pérdidas de paquetes.
- **Flags (Banderas)**: campo de 1 byte que contiene bits de control. Estos bits determinan el tipo de operación asociada al paquete, tales como ACK (reconocimiento), SYN (sincronización), FIN (finalización), así como las operaciones de `upload` o `download`. Cada bit representa una bandera específica y su activación depende del propósito del paquete.
- **Payload (Carga útil)**: contiene los datos efectivos transmitidos. Su tamaño puede variar, pero en esta implementación se fija en 4087 bytes. Este campo incluye, por ejemplo, fragmentos de archivos a transferir.

3.4. Protocolo Stop And Wait

Nuestro protocolo Stop&Wait fue implementado siguiendo el comportamiento clásico del protocolo: el emisor transmite un solo paquete y espera a recibir su ACK antes de enviar el siguiente.

Mecanismo de envío (`send`)

El método `send` construye un paquete con el número de secuencia y número de ACK actuales, y lo envía al destinatario utilizando el método `send_to` del stream. Luego, espera un ACK con el número esperado (`ack_number = sequence_number + 1`). Si no se recibe un ACK válido dentro del tiempo estipulado, se realiza un reintento, con un máximo definido por la constante `RETRIES`. En caso de éxito, se avanza el número de secuencia.

Mecanismo de recepción (`recv`)

El receptor espera un paquete y verifica que su número de secuencia coincida con el valor esperado (`ack_number`). Si es correcto, responde con un ACK y entrega el paquete recibido. Si el número de secuencia no coincide, responde con un ACK duplicado, enviando el numero de ACK que espera. Esta lógica también incluye el manejo de paquetes `FIN` para cerrar la conexión, enviando una confirmación adecuada si se recibe uno correctamente.

Manejo de errores

Ambos métodos implementan control de errores mediante `try/except`, manejando eventos de timeout y errores de cola (`queue.Empty`). Esto se usa para retransmitir los diferentes paquetes si hubiera un timeout producido por la lectura del stream. Esta es una herramienta clave del protocolo ya que aprovehca los propios timeouts del stream para no tener que crear un timer interno. Va a ser utilizado también en GBN.

Resumen del comportamiento

El flujo básico de Stop&Wait en esta implementación es:

1. Enviar un paquete de datos.
2. Esperar el ACK correspondiente.
3. Retransmitir si hay timeout o ACK incorrecto.
4. Al recibir el ACK válido, avanzar y repetir.

Este enfoque garantiza confiabilidad pero introduce latencias considerables en redes con alta pérdida, tal como se refleja en los resultados de análisis posteriores.

3.5. Protocolo Go-Back-N

El protocolo *Go-Back-N* (GBN) es un mecanismo de control de flujo y control de errores perteneciente a la familia de *sliding-window-protcol*. Este tipo de protocolos permite mejorar la utilización del canal al evitar que el emisor deba esperar una confirmación por cada paquete antes de enviar el siguiente.

En el caso de GBN, el emisor es el encargado de gestionar la secuencia de envíos, mientras que el receptor cumple un rol pasivo, aceptando únicamente paquetes en orden y descartando cualquier otro fuera de secuencia o corrupto.

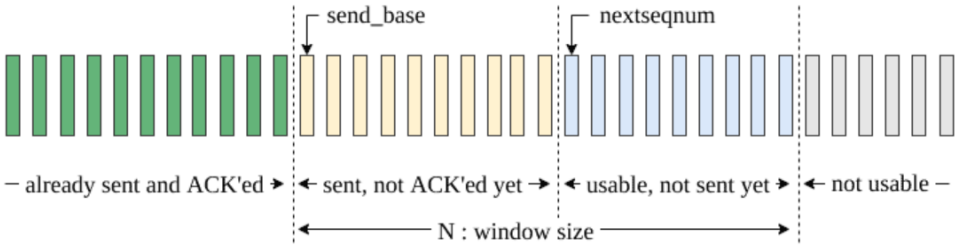


Figura 6: Ventana en GBN

3.5.1. Funcionamiento del Emisor

El emisor mantiene una ventana de tamaño N y en nuestra implementacion unicamente nos fijamos en si el ultimo ACK recibido se encuentra en nuestra ventana.

```
if ack_packet.is_ack():
     if ack_packet.ack_number - 1 in self.packet_window:
         self.packet_window.pop(ack_packet.ack_number - 1)

         new_dic = {}
         for sec in self.packet_window.keys():
             if sec >= ack_packet.ack_number:
                 new_dic[sec] = self.packet_window[sec]
         self.packet_window = new_dic
         return OK

```

Listing 2: Manejo de ACKs del lado del emisor

3.5.2. Funcionamiento del Receptor

El receptor únicamente mantiene el número de secuencia esperado. No posee un búfer de recepción, por lo que descarta cualquier paquete fuera de orden.

Cada vez que recibe un paquete válido, envía una confirmación (ACK) con el número del último paquete recibido en orden. Si recibe un paquete fuera de orden, reenvía el último ACK correspondiente, lo que puede generar ACKs duplicados.

4. Análisis

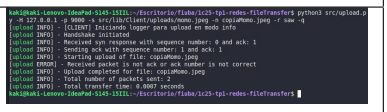
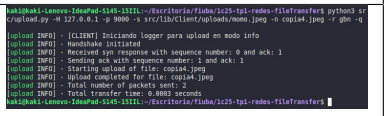
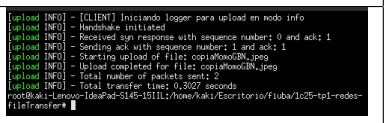
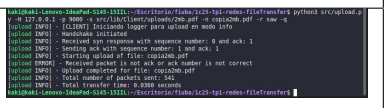
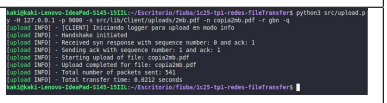
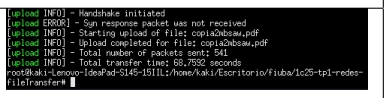
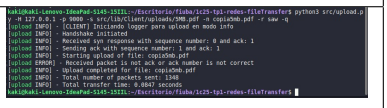
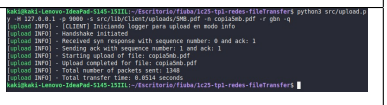
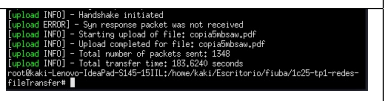
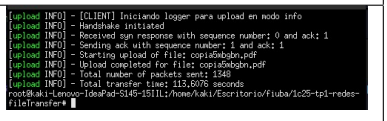
Se evaluó la performance de las versiones Stop&Wait y Go-Back-N (GBN) del protocolo de transferencia, bajo distintas condiciones de red y tamaños de archivo. Las pruebas se realizaron utilizando tres archivos con tamaños representativos:

- Pequeño: **7 KB**
- Mediano: **2 MB**
- Grande: **5 MB**

Cada archivo fue transferido bajo dos escenarios de red:

- **Sin pérdida de paquetes** (`loss = 0%`)
- **Con pérdida de paquetes** (`loss = 10%`)

A continuación se presenta una tabla comparativa con los resultados obtenidos:

Archivo	Protocolo	Pérdida	Tiempo (s)	Captura terminal
7 KB	Stop&Wait	No	0.0007	
7 KB	GBN	No	0.0003	
7 KB	Stop&Wait	Si	0.0025	
7 KB	GBN	Si	0.3027	
2 MB	Stop&Wait	No	0.036	
2 MB	GBN	No	0.0212	
2 MB	Stop&Wait	Si	68.7592	
2 MB	GBN	Si	45.7889	
5 MB	Stop&Wait	No	0.0847	
5 MB	GBN	No	0.0514	
5 MB	Stop&Wait	Si	183.6240	
5 MB	GBN	Si	113.6076	

Cuadro 1: Comparación de performance entre Stop&Wait y GBN con y sin pérdida. Última columna: salida de la terminal.

Ae puede destacar que:

- En condiciones sin pérdida, ambos protocolos muestran tiempos de transferencia muy bajos, siendo GBN levemente más eficiente, especialmente en archivos de mayor tamaño.
- Bajo pérdida del 10 %, tanto GBN como Stop&Wait sufren una degradación en el rendimiento, con tiempos de transferencia que llegan a los 2 minutos para archivos de 5MB.
    - Se puede notar que Stop&Wait tarda considerablemente mas que GBN en estos casos. Esto se debe a la necesidad de retransmitir cada paquete individualmente tras cada pérdida.
    - GBN también se ve afectado por la pérdida, pero su capacidad de enviar múltiples paquetes antes de recibir un ACK le permite recuperarse de errores de manera más eficiente, logrando tiempos menores que Stop&Wait.
    - El impacto de la pérdida es proporcional al tamaño del archivo: a mayor tamaño, mayor es la diferencia de rendimiento entre los protocolos. Esto se debe al número de paquetes transmitidos y las posibles retransmisiones aumentan considerablemente.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

La arquitectura Cliente-Servidor es un modelo de comunicación donde un cliente solicita servicios o recursos y un servidor los proporciona. El cliente inicia la conexión y el servidor permanece a la espera de solicitudes, respondiendo con los datos o acciones requeridas. Esta arquitectura permite centralizar el control y facilita la escalabilidad y mantenimiento de sistemas distribuidos.

5.2. Cual es la función de un protocolo de capa de aplicación

Un protocolo de capa de aplicación es un conjunto de reglas que permite la comunicación entre procesos de aplicaciones ubicados en diferentes dispositivos a través de una red. Su función principal es definir cómo se estructuran, interpretan y gestionan los mensajes que se intercambian, incluyendo el formato de los datos, el control de sesiones, y mecanismos de seguridad y detección de errores. Para que la comunicación sea efectiva y fiable, ambos extremos deben implementar protocolos compatibles.

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

El protocolo desarrollado en este trabajo práctico se encuentra detalladamente explicado en la sección 3 del informe.

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 cada uno?

La capa de transporte del stack TCP/IP ofrece dos protocolos principales: TCP y UDP.

UDP (User Datagram Protocol) proporciona un servicio simple, no orientado a la conexión, con bajo retardo y sobrecarga mínima. Permite la comunicación directa entre procesos (multiplexado/demultiplexado) e incluye un control básico de integridad mediante checksum, pero no garantiza la entrega, el orden ni la detección de errores.

TCP (Transmission Control Protocol), en cambio, es un protocolo orientado a la conexión que garantiza una transmisión confiable de los datos, asegurando que lleguen en orden y sin errores. Implementa mecanismos de control de flujo, control de congestión y retransmisión de paquetes perdidos. UDP es apropiado para aplicaciones donde se prioriza la velocidad, como streaming o videollamadas, mientras que TCP es preferible cuando la integridad y fiabilidad son críticas, como en transferencias de archivos o navegación web.

6. Dificultades encontradas

Manejo de ACKs (acknowledgements): Asegurar que los ACKs se reciban correctamente y distinguir entre ACKs válidos, duplicados o retrasados puede volverse complejo, especialmente en Go-Back-N. También, al finalizar la transmisión, esperar por los últimos ACKs fue un aspecto especialmente delicado y difícil de manejar correctamente.

Control de temporizadores (timeouts): Definir criterios adecuados para los timeouts fue clave, en particular para Go-Back-N, donde un timeout mal ajustado puede disparar retransmisiones innecesarias o provocar pérdida de rendimiento.

Coordinación de los números de secuencia y confirmación: Fue necesario mantener una sincronización precisa de los números de secuencia y de ACK en los paquetes de ambas partes de la comunicación para asegurar una transferencia correcta.

Mensajes fuera de orden: En UDP los paquetes pueden llegar desordenados. Aunque Go-Back-N no almacena mensajes fuera de orden, fue importante reconocer y descartar los que no correspondían a la secuencia esperada. Además, la lógica de retransmisión de dichos paquetes presentó varios desafíos.

Manejo de threads para la escucha de ACKs: En un principio adoptamos un enfoque en el cual se creaban muchos hilos, lo que derivó en condiciones de carrera difíciles de debuggear. Eventualmente, abandonamos esa estrategia en favor de una solución más simple y controlada.

Proveer un diseño extensible: Nos propusimos que el diseño de la aplicación permitiera agregar nuevos protocolos sin modificar el código ya existente, sino construyendo sobre una base común. Esto implicó separar responsabilidades y abstraer la lógica de cada protocolo.

7. Anexo: Fragmentación IPv4

7.1. Objetivos

Este experimento tiene como objetivo observar y comprender el proceso de fragmentación en IPv4, así como el comportamiento de los protocolos TCP y UDP ante la pérdida de fragmentos, y el impacto que tiene la reducción del MTU en el volumen de tráfico de red. Para ello, se diseña una red virtual en Mininet que permite generar y analizar escenarios donde la fragmentación ocurre de forma controlada.

Se busca comprobar empíricamente:

- El proceso de fragmentación IPv4.
- El comportamiento de TCP y UDP ante la pérdida de fragmentos.
- El aumento de tráfico al reducirse el MTU mínimo en la red.

Para esto se utilizan las siguientes herramientas:

- Mininet para simular la red.
- iperf para generar tráfico TCP/UDP.
- Wireshark para capturar y analizar los paquetes.

7.2. Estructura del experimento

El experimento está automatizado mediante un script de Python que utiliza Mininet para crear la red, aplicar configuraciones, y levantar los servicios necesarios.

7.2.1. Topología

Se construyó una topología lineal en Mininet conformada por dos hosts (h1 y h2) conectados a través de tres nodos intermedios. El nodo central (s2) se implementó como un router (usando una clase personalizada que habilita el reenvío de paquetes mediante ip forward), mientras que los extremos (s1 y s3) actúan como switches.

```
h1 --- s1 --- s2 --- s3 --- h2
```

- 2 hosts (h1, h2)
- 3 switches intermedios
  - Se usa un nodo en lugar de un switch en el centro de la topologia (s2), y a este nodo se le setea que pueda hacer ip-forwarding, ya que esto es lo que lo hace comportarse como un router.
- MTU reducido en una interfaz de s2: s2-eth2.
- Pérdida de paquetes simulada en una interfaz de s3.

7.3. Ejecución del experimento

El script de fragmentación propone la siguiente interfaz:

```
fragmentacion.py [--mtu MTU] [--loss LOSS]
```

7.3.1. Ejecutamos el script de fragmentación, inicialmente sin pérdida de paquetes

Desde el root corremos:

```
sudo python3 src/lib/Anexo/fragmentacion.py --mtu 600 --loss 0
```

7.3.2. Enviar trafico usando UDP o TCP

Por ejemplo usando UDP:

```
mininet> h2 iperf -s -u &
mininet> h1 iperf -c h2 -u -l 1400
```

7.3.3. Analizar el comportamiento con Wireshark

En Wireshark, analizar s1-eth2 (sin fragmentación) y s3-eth2 (con fragmentación). En s1 se verán los paquetes enviados, como paquetes completos, y en s3 se verán los paquetes fragmentados.

Se observa cómo los paquetes se dividen en fragmentos:

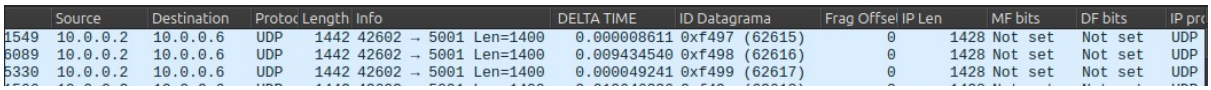
- El campo **ID Datagrama** permite agrupar los fragmentos y ver que hay tres fragmentos que pertenecen al mismo datagrama.
- El bit **MF (More Fragments)** indica si hay más fragmentos.
- El campo **Fragment Offset** indica la posición del fragmento en el paquete original.
- El campo **IP Length** muestra la longitud del paquete fragmentado
- Se puede ver como cada paquete aparece fragmentado en s3-eth2 en la siguiente captura de Wireshark:

	DELTA TIME	ID Datagrama	Frag Offset	IP Len	MF bits	DF bits	IP proto
› protocol (proto=UDP 17, off=0, ID=d13f)	0.000000000	0xd13f (53…	0	596	Set	Not set	UDP
› protocol (proto=UDP 17, off=576, ID=d13f) [Reassembled…	0.000015448	0xd13f (53…	72	596	Set	Not set	UDP
Len=1400	0.000001561	0xd13f (53…	144	276	Not set	Not set	UDP

Figura 7: Fragmentacion de un paquete

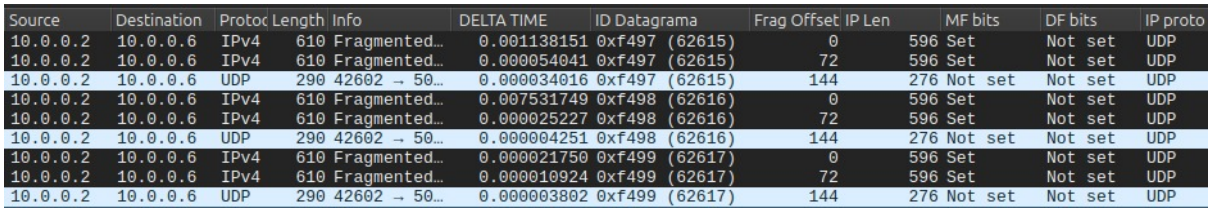
- En Wireshark se ofrece una herramienta que permite fácilmente comparar como la fragmentación afecta el tráfico en la red. En la parte de estadísticas, y luego en Summary se puede observar el total de paquetes en la red. En esta ocasión se pudo ver:
  - En s1-eth2 hay 941.
  - En s3-eth2 hay 2821.

- En TCP vamos a ver mucho mas trafico que en UDP porque TCP envia ACKs, pero no vamos a ver paquetes retransmitidos porque no hay perdida de paquetes, eso lo analizaremos a continuacion.



	Source	Destination	Protoc	Length	Info	DELTA TIME	ID Datagrama	Frag Offse	IP Len	MF bits	DF bits	IP pr
1549	10.0.0.2	10.0.0.6	UDP	1442	42602 → 5001 Len=1400	0.000008611	0xf497 (62615)	0	1428	Not set	Not set	UDP
6089	10.0.0.2	10.0.0.6	UDP	1442	42602 → 5001 Len=1400	0.009434540	0xf498 (62616)	0	1428	Not set	Not set	UDP
6330	10.0.0.2	10.0.0.6	UDP	1442	42602 → 5001 Len=1400	0.000049241	0xf499 (62617)	0	1428	Not set	Not set	UDP

Figura 8: Paquetes antes de ser fragmentados, en s1-eth2



Source	Destination	Protoc	Length	Info	DELTA TIME	ID Datagrama	Frag Offset	IP Len	MF bits	DF bits	IP proto
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.001138151	0xf497 (62615)	0	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.000054041	0xf497 (62615)	72	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	UDP	290	42602 → 50…	0.000034016	0xf497 (62615)	144	276	Not set	Not set	UDP
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.007531749	0xf498 (62616)	0	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.000025227	0xf498 (62616)	72	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	UDP	290	42602 → 50…	0.000004251	0xf498 (62616)	144	276	Not set	Not set	UDP
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.000021750	0xf499 (62617)	0	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	IPv4	610	Fragmented…	0.000010924	0xf499 (62617)	72	596	Set	Not set	UDP
10.0.0.2	10.0.0.6	UDP	290	42602 → 50…	0.000003802	0xf499 (62617)	144	276	Not set	Not set	UDP

Figura 9: Paquetes luego de la fragmentacion, en s3-eth2

Finalmente, se puede observar en las capturas que son las figuras 8 y 9, cómo el proceso de fragmentación mantiene la coherencia del datagrama original. En particular, los paquetes con número de secuencia **62615**, **62616** y **62617** corresponden a fragmentos del mismo datagrama IP, identificados por compartir el mismo campo **ID Datagrama** y diferenciarse en los campos **Fragment Offset**. Se puede ver que aquellos datagramas que deben seguir fragmentando tienen el bit **MF** seteado. Estos fragmentos fueron capturados en las interfaces s1-eth2 y s3-eth2, lo que confirma que la fragmentación ocurrió en el router s2 y que los fragmentos fueron correctamente encaminados hacia el host destino. Esta observación permite verificar empíricamente que la fragmentación preserva la integridad del contenido original a través de múltiples paquetes, siempre y cuando todos los fragmentos lleguen a destino de forma completa.

7.3.4. Ejecutamos el script de fragmentación con pérdida de paquetes

```
sudo python3 src/lib/Anexo/fragmentacion.py --mtu 600 --loss 10
```

Nuevamente generamos tráfico y abrimos Wireshark para analizar la fragmentación de paquetes en las interfaces s1-eth2 y s3-eth2. La diferencia en este caso es que, al haber pérdida de paquetes en el enlace entre s3 y h2, el comportamiento observado varía significativamente según el protocolo de transporte utilizado. Mientras que en UDP la pérdida de un fragmento implica la pérdida total del datagrama y se generan mensajes ICMP de error, en TCP se activan mecanismos de recuperación que intentan reenviar los segmentos afectados. Esto permite comparar el enfoque confiable de TCP contra el no confiable de UDP.

Se repiten las pruebas de tráfico UDP y TCP.

7.3.5. UDP

- Al perderse un fragmento, el datagrama completo se descarta.
- Wireshark muestra fragmentos huérfanos (no reensamblados).
- Además, iperf reporta una tasa de pérdida de datagramas que refleja este fenómeno.

```
mininet> h1 iperf -c h2 -u -l 1400
------------------------------------------------------------
Client connecting to 10.0.0.6, UDP port 5001
Sending 1400 byte datagrams, IPG target: 10681.15 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 36387 connected with 10.0.0.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0192 sec  1.26 MBytes  1.05 Mbits/sec
[  1] Sent 941 datagrams
[  1] Server Report:
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  1] 0.0000-10.0181 sec   972 KBytes   795 Kbits/sec   0.017 ms 229/940 (24%)
mininet>
```

Figura 10: Porcentaje de pérdidas en iperf

- Se observan mensajes ICMP de tipo "Fragment reassembly time exceeded", lo que indica que el host receptor descartó datagramas IP fragmentados debido a la pérdida de alguno del paquete.

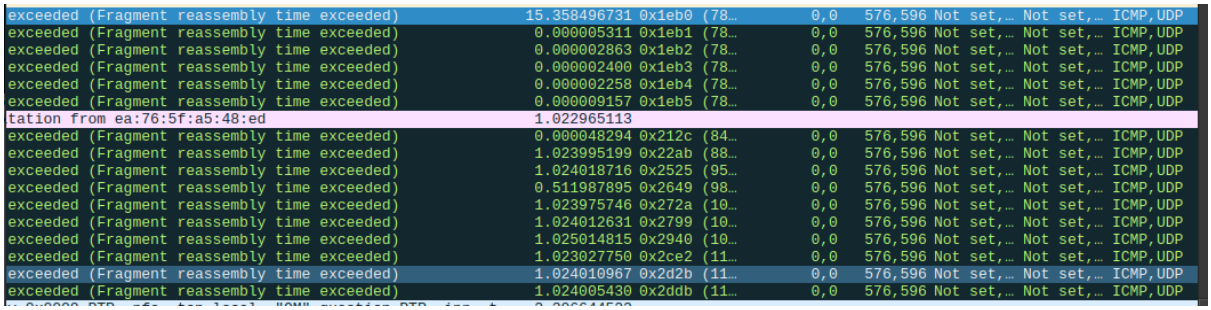


Figura 11: Fragmentos descartados por el host

Por último, en la figura 12 se puede ver un conjunto de fragmentos que llegan a la interfaz s3-eth2, pero al faltar al menos uno de ellos, no es posible formar el datagrama completo. Por lo tanto, serán descartados por el host destino. Por ejemplo se puede ver que el datagrama con ID **19541** solo consigue llegar hasta el final del recorrido con dos fragmentos, faltandole el ultimo que deberia ser el de offset **144**.

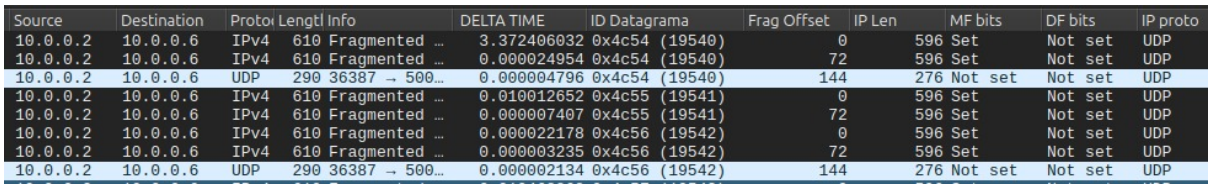


Figura 12: Paquetes fragmentados perdidos

7.3.6. TCP

- El protocolo detecta pérdida y retransmite los segmentos afectados.
- En Wireshark se observan:
  - `TCP Retransmission`: retransmisión por timeout.
  - `TCP Fast Retransmission`: retransmisión tras 3 ACKs duplicados.
- Se pueden aplicar los siguientes filtros para analizar estos eventos:
  - `tcp.analysis.retransmission`
  - `tcp.analysis.fast_retransmission`
  - `tcp.analysis.retransmission && !tcp.analysis.fast_retransmission`
- A diferencia con UDP, iperf no indica que hubieron paquetes perdidos.

```
mininet> h1 iperf -c h2 -l 1400
------------------------------------------------------------
Client connecting to 10.0.0.6, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 58202 connected with 10.0.0.6 port 5001 (icwnd/mss/irtt=14/1448/1458)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-15.6686 sec   180 KBytes  93.9 Kbits/sec
mininet>
```

Figura 13: Salida de iperf con TCP

Source	Destination	Protoc	Length	Info	DELTA TIME	ID Datagrama	Frag	IP Len	MF bits	DF bits	IP proto
10.0.0.2	10.0.0.6	TCP	2962	58202 → 5001 [PSH, ACK] Se…	0.000010892	0x78ee (30958)	0	2948	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	66	5001 → 58202 [ACK] Seq=1 A…	0.000021967	0x3be7 (15335)	0	52	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	78	[TCP Previous segment not …	0.000013568	0x3be9 (15337)	0	64	Not set	Not set	TCP
10.0.0.2	10.0.0.6	TCP	8754	58202 → 5001 [PSH, ACK] Se…	0.000007088	0x78f0 (30960)	0	8740	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 27#1] 5001 → …	0.000019757	0x3bea (15338)	0	64	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 27#2] 5001 → …	0.000014768	0x3beb (15339)	0	64	Not set	Not set	TCP
10.0.0.2	10.0.0.6	TCP	1514	[TCP Fast Retransmission] …	0.000005867	0x78f6 (30966)	0	1500	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 27#3] 5001 → …	0.000019184	0x3bec (15340)	0	64	Not set	Not set	TCP
10.0.0.2	10.0.0.6	TCP	1514	[TCP Retransmission] 58202…	0.000004058	0x78f7 (30967)	0	1500	Not set	Not set	TCP
10.0.0.6	10.0.0.2	TCP	78	5001 → 58202 [ACK] Seq=29 …	0.001154760	0x3bed (15341)	0	64	Not set	Not set	TCP

Figura 14: Paquetes enviados en s1-eth2 con TCP

En la figura 14, correspondiente a la interfaz s1-eth2, se observa el envío de segmentos TCP. Se destacan eventos como **TCP Previous Segment Not Captured**, lo que indica que Wireshark identificó una discontinuidad en la secuencia esperada. Además, se observan paquetes etiquetados como **TCP Dup ACK x**, seguidos por una retransmisión rápida (**Fast Retransmission**) o una retransmisión por timeout (**Retransmission**), lo cual confirma que TCP activa ambos mecanismos según el contexto.

Source	Destination	Protocol	Leng	Info	DELTA TIME	ID Datagrama	Frag C	IP Len	MF bits
10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=78e9) [Reassembled …	0.000011602	0x78e9 (30953)	72	596	Set
10.0.0.2	10.0.0.6	TCP	362	[TCP Previous segment not captured] 58202 → 5001 [ACK] Seq=5853 Ack=…	0.000027506	0x78e9 (30953)	144	348	Not set
10.0.0.6	10.0.0.2	TCP	78	[TCP Previous segment not captured] 5001 → 58202 [ACK] Seq=29 Ack=29…	0.000008141	0x3be9 (15337)	0	64	Not set
10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=78ea) [Reassembled in…	0.000018204	0x78ea (30954)	0	596	Set
10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=78ea) [Reassembled …	0.000020024	0x78ea (30954)	72	596	Set
10.0.0.2	10.0.0.6	TCP	362	58202 → 5001 [ACK] Seq=7301 Ack=1 Win=42496 Len=1448 TSval=387575070…	0.000013427	0x78ea (30954)	144	348	Not set
10.0.0.6	10.0.0.2	TCP	78	[TCP Dup ACK 24#1] 5001 → 58202 [ACK] Seq=29 Ack=2957 Win=41472 Len=…	0.000006439	0x3bea (15338)	0	64	Not set
10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=78eb) [Reassembled in…	0.000018532	0x78eb (30955)	0	596	Set
10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=78eb) [Reassembled …	0.000016328	0x78eb (30955)	72	596	Set
10.0.0.2	10.0.0.6	TCP	362	58202 → 5001 [ACK] Seq=8749 Ack=1 Win=42496 Len=1448 TSval=387575070…	0.000010592	0x78eb (30955)	144	348	Not set

Figura 15: Paquetes recibidos en s3-eth2 con TCP

En la interfaz s3-eth2, Wireshark muestra múltiples fragmentos con el mismo **ID Datagrama**, con campos **MF** (More Fragments) activados y offsets variables. Cuando uno de estos fragmentos se pierde, TCP no logra completar la entrega del segmento, lo cual genera retransmisiones.

7.4. Resultados observados

7.4.1. Proceso de fragmentación

Se verificó el proceso de fragmentación al enviar paquetes de 1400 bytes a través de un enlace con MTU de 600 bytes. Al generar tráfico con tamaño mayor al MTU del enlace (600 bytes), Wireshark muestra cómo se divide un paquete en múltiples fragmentos IPv4, identificables por el mismo ID de paquete, campos MF y offsets. La fragmentación es manejada por el router s2.

7.4.2. Comportamiento de TCP ante pérdida de fragmentos

Al introducir pérdida en el s3, se observó que TCP retransmite automáticamente el paquete completo cuando falta un fragmento, dado que TCP requiere entrega confiable. Esto implica mayor latencia y tráfico adicional por retransmisiones.

7.4.3. Comportamiento de UDP ante pérdida de fragmentos

En el caso de UDP, si se pierde uno de los fragmentos, el datagrama completo no puede ser reconstruido y se descarta sin notificación. Esto se refleja en que iperf muestra pérdida de paquetes sin intento de recuperación, ya que UDP no implementa mecanismos de fiabilidad.

7.4.4. Aumento del tráfico al reducirse el MTU

La reducción del MTU implica una mayor cantidad de fragmentos para transmitir la misma cantidad de datos. Esto genera un aumento del número total de paquetes enviados y una sobrecarga en la red. Esta condición se confirmó al observar un mayor número de paquetes IP en Wireshark durante transmisiones con MTU reducido. En capturas sin fragmentación hubo alrededor de 900 paquetes, mientras que con fragmentación se superaron los 2800.

7.5. Conclusión

El experimento permitió observar de forma práctica cómo IPv4 maneja la fragmentación y qué impacto tiene en el tráfico de red. Se verificó que TCP es resiliente ante la pérdida de fragmentos mediante retransmisiones, mientras que UDP pierde datos irreversiblemente. Además, se evidenció que la fragmentación genera un aumento significativo en la cantidad de paquetes transmitidos, afectando el rendimiento y eficiencia de la red.