Redes

(TA048) Álvarez Hamelin

Trabajo Práctico 1
File Transfer

08-05-2025

Ascencio Felipe Santino 110675

Burgos Moreno Daniel 110486

García Pizales Ignacio 105043

Levi Dolores 105993

Orive María Sol 91351

Introducción

1. Introducción

En el ámbito de los sistemas distribuidos (y las redes informáticas), la transferencia confiable de datos constituye un pilar fundamental para la comunicación entre procesos en red.

El presente trabajo práctico, correspondiente a la asignatura *Introducción a los Sistemas Distribuidos (75.43)* y *Redes (TA048)*, tiene como objetivo el diseño e implementación de un protocolo de *Reliable Data Transfer* (RDT) sobre el servicio de transporte no fiable proporcionado por UDP.

Para ello, se desarrollarán dos variantes del protocolo: Una basada en el esquema *Stop & Wait* y otra que emplea el mecanismo de *Selective Repeat* (SACK).

El ejercicio se articula alrededor de una aplicación 'Cliente - Servidor', capaz de operar tanto en modo `UPLOAD` (envío de archivos desde el cliente al servidor), como en modo `DOWNLOAD` (descarga desde el servidor al cliente).

La aplicación debe garantizar mínimamente la entrega íntegra de archivos binarios de hasta 5 MB, incluso en presencia de condiciones de pérdida de paquetes de hasta un 10 % en el trayecto.

Asimismo, se requiere la capacidad de manejo concurrente de múltiples clientes y la validación de, al menos, dos condiciones de error distintas.

Para simular y controlar de manera precisa las condiciones de la red (pérdida de paquetes, topologías y parámetros de enlace), se hará uso de la herramienta Mininet. Adicionalmente, se empleará la interfaz de sockets de Python 3, siguiendo las recomendaciones de estilo PEP8 y haciendo uso de `flake8` para asegurar la calidad del código.

Con este conjunto de herramientas, el usuario podrá experimentar de forma reproducible el comportamiento de los protocolos RDT y verificar empíricamente su resistencia a la pérdida de paquetes y su rendimiento bajo diferentes configuraciones de tráfico.

A continuación se pasa a desarrollar el informe del trabajo práctico realizado, como anexo se agrega tambien la 'Presentacion.ppt' y el 'README.md'.

Hipótesis y suposiciones realizadas

2. Hipótesis y suposiciones realizadas

Para el desarrollo y evaluación de la aplicación de transferencia confiable de datos sobre UDP, se establecen las siguientes hipótesis y suposiciones iniciales:

- **Pérdida de paquetes aleatoria e independiente:** Se asume que la pérdida de cada paquete ocurre de forma independiente, con una tasa promedio configurable (hasta 10 %), y sin correlación temporal ni espacial entre eventos de pérdida.
- **Retraso y jitter acotados:** El retardo de propagación y la variabilidad de tiempo de llegada (*jitter*) se consideran constantes y pequeñas respecto al período de espera del protocolo (`Timeout`), de modo que no se produzcan falsos positivos de pérdida.
- **Buffers de recepción ilimitados:** Los hosts disponen de memoria suficiente para almacenar temporalmente todos los fragmentos o segmentos hasta su reensamblado, sin desbordar buffers.
- **Relojes locales perfectamente ajustados:** Cada nodo dispone de un reloj lo suficientemente estable para medir los tiempos de espera (*timeouts*) con la precisión necesaria para los algoritmos 'Stop & Wait' y 'Selective Repeat'.
- **Topología y enlaces controlados:** El entorno de Mininet reproduce fielmente la topología lineal definida, y los parámetros de enlace (MTU, pérdida, ancho de banda) se aplican únicamente a las interfaces especificadas, sin efectos colaterales en otros enlaces.
- **Concurrencia de clientes:** El servidor puede atender conexiones concurrentes sin que la gestión de un cliente interfiera con el desempeño de otro, dado que se emplean hilos o procesos aislados en cada sesión.

3. Implementación protocolos 'UDP'

El objetivo de esta implementación es agregar sobre la capa de transporte UDP los principios básicos de la transferencia de datos confiables (RDT). Los protocolos implementados son: Stop & Wait y Selective repeat.

3.1. Stop & Wait

Este protocolo establece que, una vez enviado un paquete, se debe esperar confirmación (ACK) de la otra parte para luego enviar el siguiente paquete.

Este protocolo tiene como ventaja la simpleza y confiabilidad, ya que asegura que cada paquete llegue a destino. En caso de no recibir el ACK, se procede a reenviar el paquete. Como desventaja, es un protocolo lento, ya que se envían uno por uno los paquetes siempre esperando la respuesta antes de enviar el siguiente.

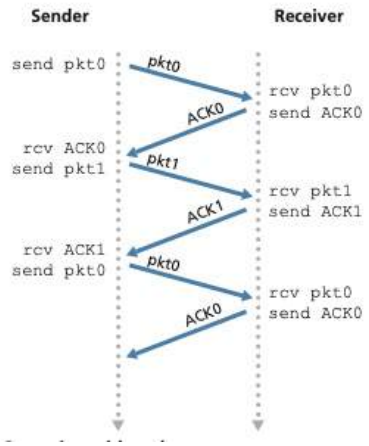

Figura 1: Stop and wait sin pérdida de paquetes

En la figura se puede ver cómo se utiliza un bit alternando entre 0 y 1 para identificar los paquetes y evitar duplicados.

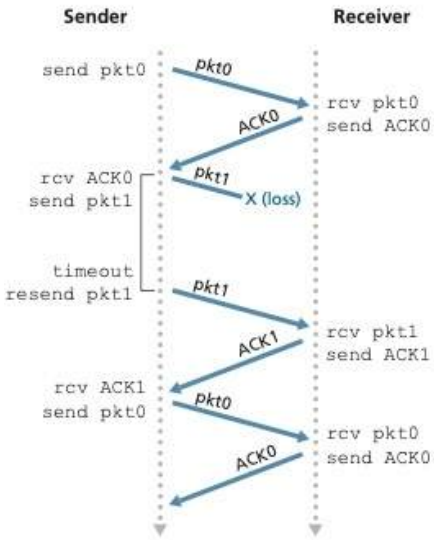

Figura 2: Stop and wait con pérdida de paquetes

3.2. Selective repeat

Este protocolo es muy eficiente ya que tiene una ventana de envío, a diferencia de Stop & Wait que sólo envía de a un paquete a la vez.

El emisor tiene una ventana de envío configurada en nuestro caso por 5 paquetes. Cada uno de estos paquetes tiene un número de secuencia asociado y además, estos paquetes se envían todos dentro de esa ventana. El receptor va a enviar un ACK por cada paquete recibido correctamente.

La ventaja de este protocolo por sobre GBN (Go-Back-N) que también mantiene una ventana de envíos es que solo retransmite el paquete del que no se recibió ACK.

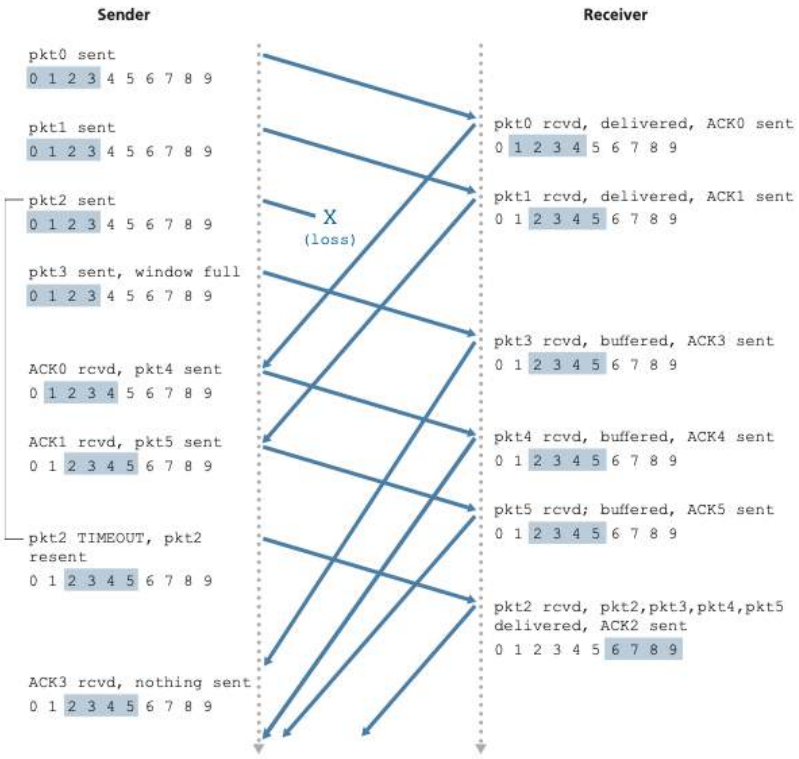

Figura 3: Selective repeat

4. Implementación protocolo 'RDT'

El protocolo RDT (Reliable Data Transfer) implementado garantiza la entrega confiable de archivos a través de una red no confiable (UDP). Se basa en los principios del modelo RDT teórico, incorporando control de errores, confirmaciones (ACK/NACK), y dos modos operativos: Stop-and-Wait y Selective Repeat.

4.1. Tipos de paquetes

Armamos 5 tipos de paquetes:

- InitPackage: Establece la conexión y define la operación
- AckPackage: Confirma la recepción exitosa
- DataPackage: Transmite los datos del archivo
- FinPackage: Indica fin de transferencia
- NackPackage: Indica paquete corrupto o perdido

Todos los paquetes se serializan como strings codificados en utf-8, con campos separados por el símbolo "|".

Los paquetes tienen un campo type, que permite al receptor identificar cómo procesarlo (0,1,2,3,4).

4.2. Explicación de formatos

Cada tipo de paquete tiene un header distinto que está graficado en la siguiente figura.

Tipo de Paquete	Valor `type`
`InitPackage`	`0`
`AckPackage`	`1`
`DataPackage`	`2`
`FinPackage`	`3`
`NackPackage`	`4`

Figura 4: Tipos de paquetes utilizados

Todos los paquetes, excepto InitPackage ya que se considera en 0, tienen un sequence_number dentro de su formato. Este campo se utiliza para mantener el orden de los paquetes, detectar duplicados y gestionar el avance de ventana. Esto funciona de diferentes maneras dependiendo del protocolo utilizado:

Protocolo	**Uso de sequence_number**	**Rango**
Stop-and-Wait	Detectar duplicados, alternar entre paquetes	0 y 1 (bit alternado)
Selective Repeat	Gestión de ventana, ACK selectivo, retransmisión	0..N (incremental)

Cuadro 1: Resumen de los headers por tipo de paquete en el protocolo RDT

Detalle de cada uno de los paquetes:

4.2.1. InitPackage

Formato: `0|<operation>|<file_name>|<file_extension>`

Campo	Descripción
`type`	`0` para INIT.
`operation`	upload o download
`file_name`	Nombre base del archivo (sin extensión)
`file_extension`	Extensión del archivo a transferir, como `txt`, `jpg`, etc.

Cuadro 2: Descripción de los campos del paquete InitPackage

4.2.2. AckPackage

Formato: `1|<sequence_number>|<valid>`

Campo	Descripción
`type`	`1` para ACK.
`sequence_number`	Número de secuencia que se está confirmando como recibido
`valid`	Booleano (True o False). True indica que el paquete recibido fue correcto.

Cuadro 3: Descripción de los campos del paquete AckPackage

4.2.3. DataPackage

Formato: `2|<sequence_number>|<checksum>|<payload>`

Campo	Descripción
`type`	`2` para DATA.
`sequence_number`	Número de secuencia de este fragmento de datos
`checksum`	Valor de control calculado como sum(data) % 256 para verificar integridad
`payload`	Contenido binario del archivo en ese paquete

Cuadro 4: Descripción de los campos del paquete DataPackage

4.2.4. FinPackage

Formato: `3|<sequence_number>`

Campo	Descripción
`type`	`3` para FIN.
`sequence_number`	El último número de secuencia enviado

Cuadro 5: Descripción de los campos del paquete FinPackage

4.2.5. NackPackage

Formato: `4|<sequence_number>|<valid>`

Campo	Descripción
`type`	4 para FIN.
`sequence_number`	Indica el número del paquete recibido con error
`valid`	Siempre False, para indicar que se necesita retransmisión

Cuadro 6: Descripción de los campos del paquete NackPackage

4.3. Ejemplos

En resumen, el formato de cada uno de los paquetes que utiliza nuestro protocolo es el siguiente:

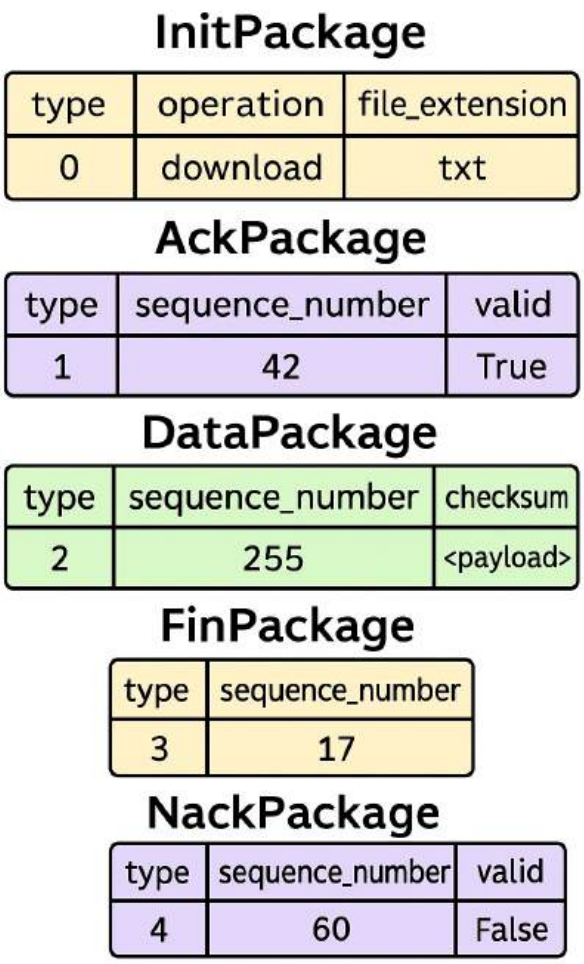

Figura 5: Ejemplos de todos los tipos de paquetes

Tutorial de funcionamiento

5. Requisitos

- Python 3.x instalado.
- Librería estándar `socket` de Python.
- Herramienta **Mininet** para simular condiciones de red.

6. Instalar dependencias

Para instalar las dependencias necesarias para la ejecución tanto del 'Servidor', como del 'Cliente' y los 'Tests', basta con ejecutar por primera vez el servidor:

```
uv run src/start-server.py -r 0
```

7. Ejecución de los tests

```
pytest
```

8. Comandos para la ejecución del Servidor

```
> uv run src/start-server.py -r 1       # Inicia el servidor con el protocolo '
    Selective Repeat'.

usage : uv run src/start-server.py [ - h ] [ - v | -q ] [ - H ADDR ] [ - p PORT ]
    [- s DIRPATH ] [ - r protocol ]
< command description >
optional arguments :
-h , -- help        show this help message and exit
-v , -- verbose     increase output verbosity
-q , -- quiet       decrease output verbosity
-H , -- host        service IP address
-p , -- port        service port
-s , -- storage     storage dir path
-r , -- protocol    error recovery protocol
```

9. Comandos para Download del Cliente

```
> uv run src/download.py -n "hello_world.py" -d "src/my_folder" -p 8080 -H "
    127.0.0.1"       # Download del archivo 'hello_world.py' con el protocolo 'Stop
    & Wait'.

usage : uv run src/download.py [ - h ] [ - v | -q ] [ - H ADDR ] [ - p PORT ] [ - d
    FILEPATH ] [ - n FILENAME ] [ - r protocol ]
< command description >
optional arguments :
-h , -- help         show this help message and exit
-v , -- verbose      increase output verbosity
-q , -- quiet        decrease output verbosity
-H , -- host         server IP address
-p , -- port         server port
-d , -- dst          destination file path
-n , -- name         file name
-r , -- protocol     error recovery protocol
```

10. Comandos para Upload del Cliente

```
> uv run src/upload.py -n "hello_world.py" -r 0       # Hace el upload de archivo '
    hello_world.py' con el protocolo 'Stop & Wait'.

usage : uv run src/upload.py [ - h ] [ - v | -q ] [ - H ADDR ] [ - p PORT ] [ - s
    FILEPATH ] [ - n FILENAME ] [ - r protocol ]
< command description >
optional arguments :
-h , -- help         show this help message and exit
-v , -- verbose      increase output verbosity
-q , -- quiet        decrease output verbosity
-H , -- host         server IP address
-p , -- port         server port
-s , -- src          source file path
-n , -- name         file name
-r , -- protocol     error recovery protocol
```

11. Ejecución de la topología para probar casos con pérdida de paquetes

11.1. Construir la topología en 'Mininet'

```
sudo mn --custom ./src/topology.py --topo customTopo,num_clients=2,loss_percent=10
    --mac -x
```

Al hacer esto el 'host' debería estar conectado a '10.0.0.1:8080'.

11.2. 'UPLOAD' con 'Stop Wait'

11.2.1. Consola del 'Servidor'

```
python3 ./src/start-server.py -r 0 -H "10.0.0.1" -p 8080
```

11.2.2. Consola del 'Cliente'

```
python ./src/upload.py -n "hello_world.py" -r 0 -H "10.0.0.1" -p 8080
```

11.3. 'DOWNLOAD' con 'Stop Wait'

11.3.1. Consola del 'Servidor'

```
python3 ./src/start-server.py -r 0 H "10.0.0.1" -p 8080
```

11.3.2. Consola del 'Cliente'

```
python ./src/download.py -n "hello_world.py" -r 0 -H "10.0.0.1" -p 8080
```

11.4. 'UPLOAD' con 'Selective Repeat'

11.4.1. Consola del 'Servidor'

```
python3 ./src/start-server.py -r 1 -H "10.0.0.1" -p 8080
```

11.4.2. Consola del 'Cliente'

```
python ./src/upload.py -n "hello_world.py" -r 1 -H "10.0.0.1" -p 8080
```

11.5. 'DOWNLOAD' con 'Selective Repeat'

11.5.1. Consola del 'Servidor'

```
python3 ./src/start-server.py -r 1 H "10.0.0.1" -p 8080
```

11.5.2. Consola del 'Cliente'

```
python ./src/download.py -n "hello_world.py" -r 1 -H "10.0.0.1" -p 8080
```

12. Ejemplos ilustrativos de Uso:

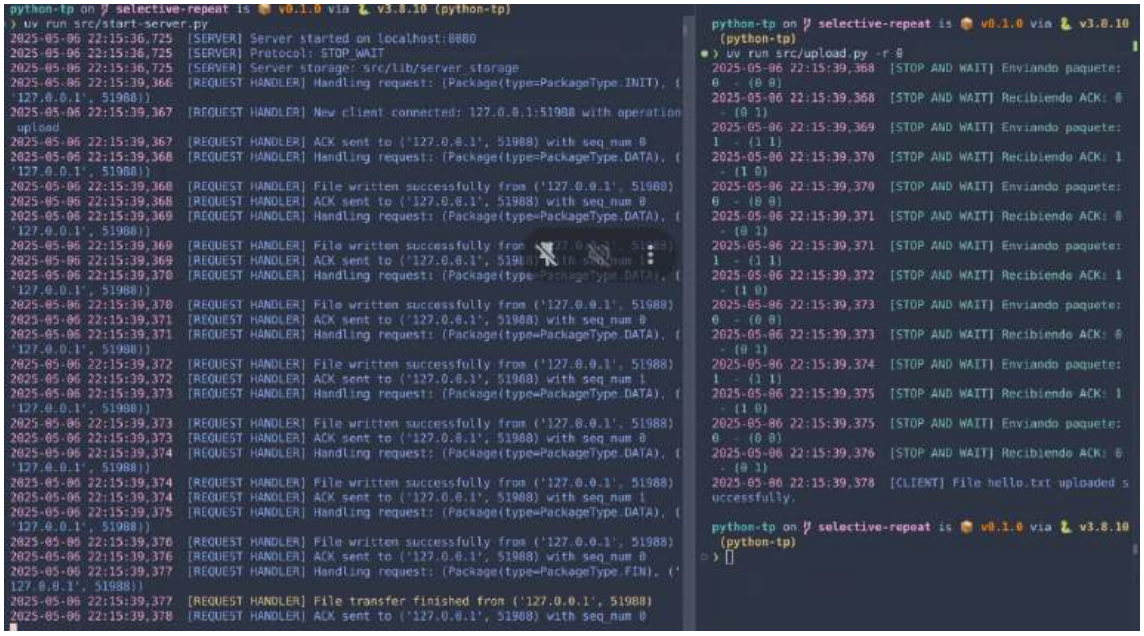

Figura 6: Ejemplo de 'UPLOAD' con 'Stop & Wait'

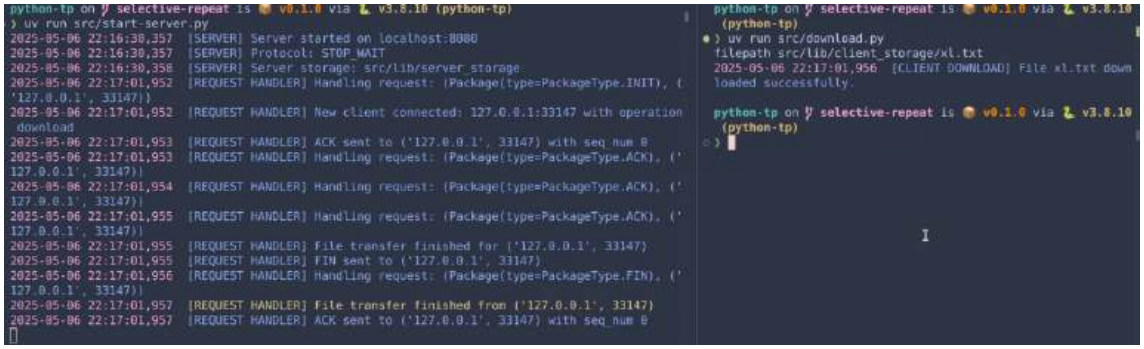

Figura 7: Ejemplo de 'DOWNLOAD' con 'Stop & Wait'

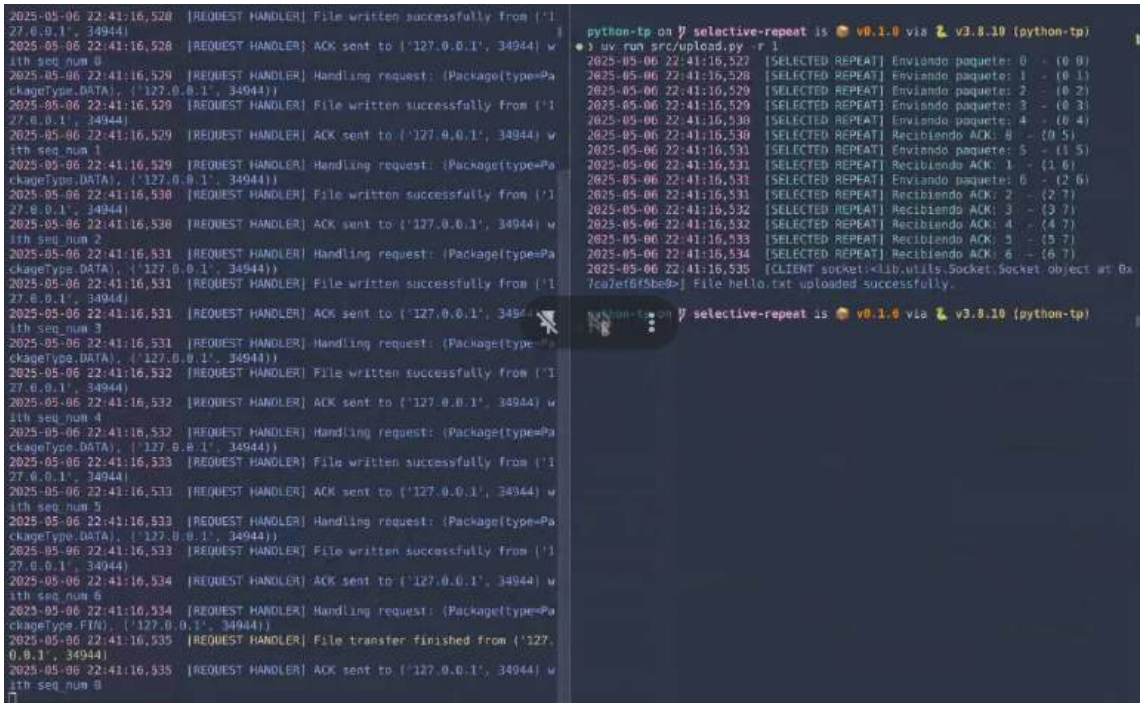

Figura 8: Ejemplo de 'UPLOAD' con 'Selective Repeat'

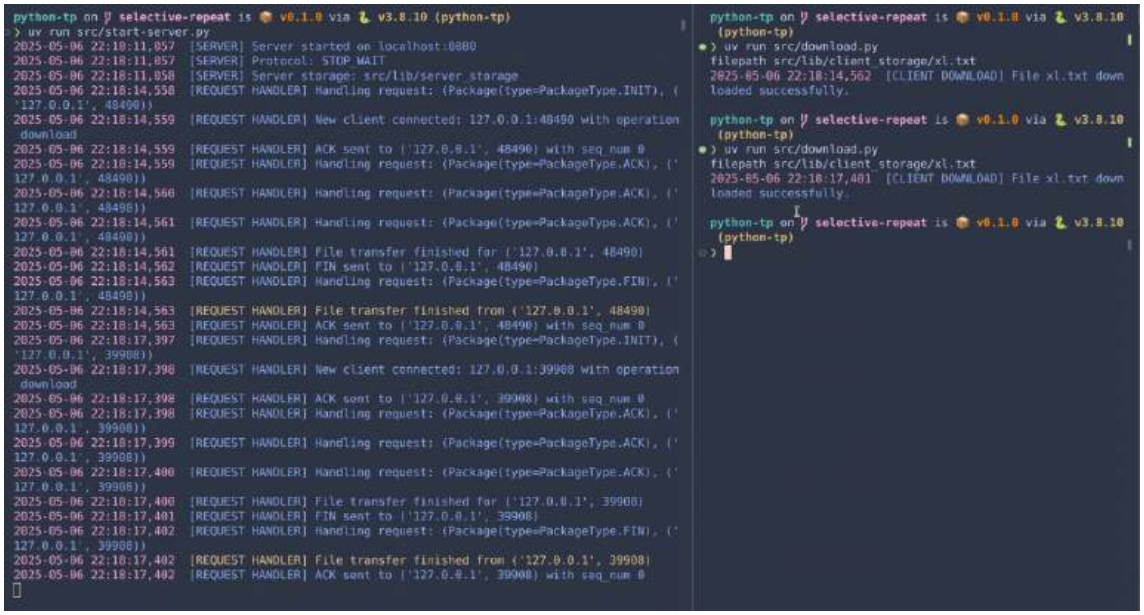

Figura 9: Dos 'Clientes' en momentos distintos interactuan con el 'Servidor'

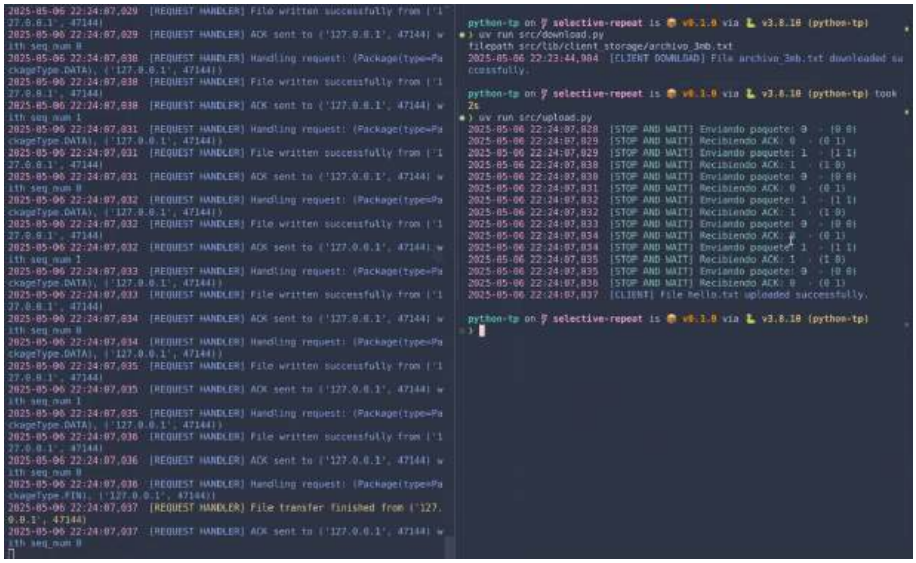

Figura 10: 'DOWNLOAD' y 'UPLOAD' consecutivos

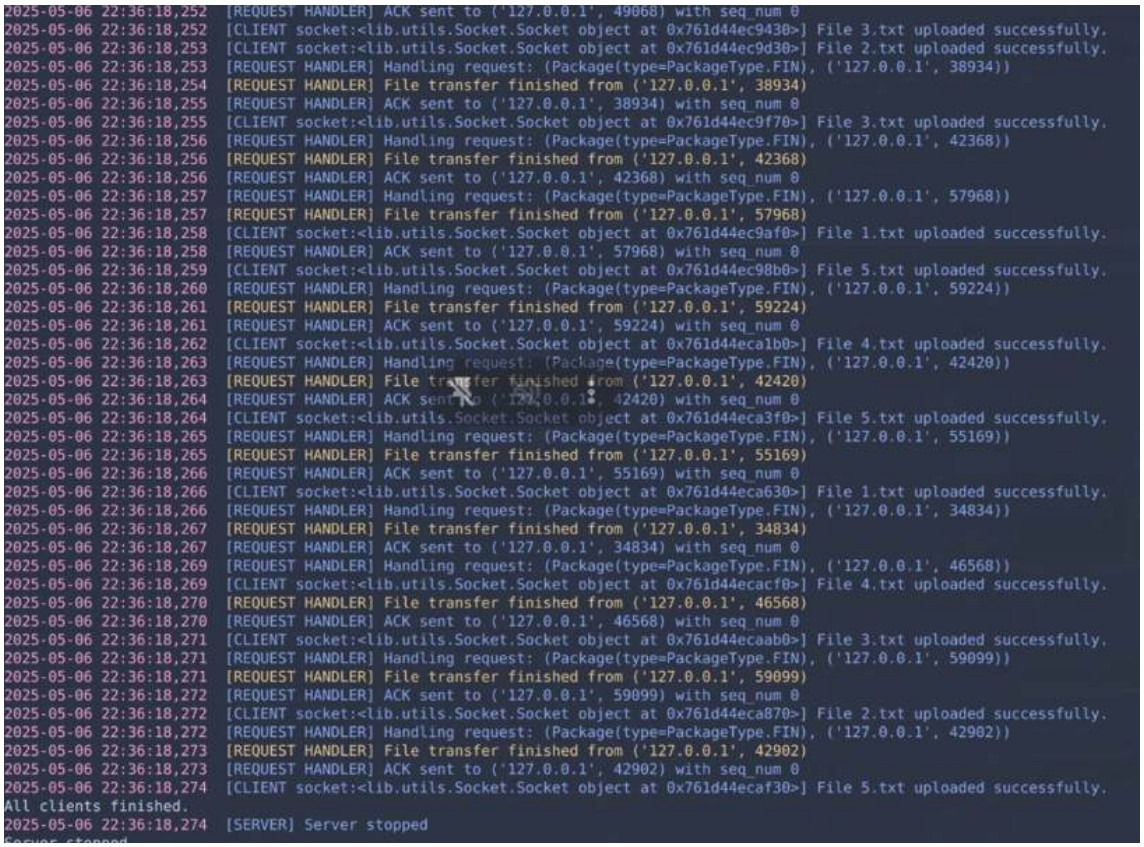

Figura 11: Cuarenta 'Clientes' interactuan concurrentemente con el 'Servidor', cada uno lo hace con Seis archivos

Pruebas

13. Pruebas Unitarias

Con el fin de validar la correcta implementación de la aplicación y asegurar el cumplimiento de los requisitos funcionales, se diseñó un conjunto de pruebas unitarias que ejercitan cada componente crítico del trabajo práctico.

A continuación se presenta la evidencia de la ejecución de dichos tests, junto con la indicación de su resultado exitoso:

Figura 12: Resultados de las pruebas unitarias. Todas las pruebas se han completado satisfactoriamente, demostrando el correcto comportamiento de los módulos implementados.

14. Control de errores

La implementación desarrollada incluye mecanismos para gestionar diversos escenarios de error durante la interacción entre el *Servidor* y el *Cliente*.

A continuación se describen los principales casos controlados:

- **Paquetes corruptos:** Se verifica la integridad de cada paquete mediante el cálculo y validación de su `checksum`, descartando aquellos que no cumplan con el valor esperado.
- **Archivo inexistente:** Antes de iniciar la operación de `UPLOAD`, el servidor comprueba que el archivo especificado por el cliente exista en la ruta indicada. En caso contrario, se retorna un mensaje de error sin iniciar la transferencia.
- **Requests de clientes desconectados:** Si un cliente interrumpe la conexión de forma inesperada, el servidor detecta el cierre de socket correspondiente y descarta cualquier solicitud pendiente asociada a ese cliente.
- **Requests fallidas:** En situaciones donde una solicitud no puede completarse (por ejemplo, debido a pérdida reiterada de paquetes), el protocolo evita finalizar la conexión de forma abrupta; En su lugar, informa al cliente del fallo y permite nuevos intentos de la operación sin cerrar la sesión.

15. Análisis de Rendimiento

Para complementar las pruebas unitarias, se efectuaron experimentos de rendimiento con ambas variantes del protocolo (Stop & Wait y Selective Repeat).

A continuación se presenta el gráfico resultante, en el que se comparan los rendimientos obtenidos para cada protocolo en función del tamaño de los archivos.

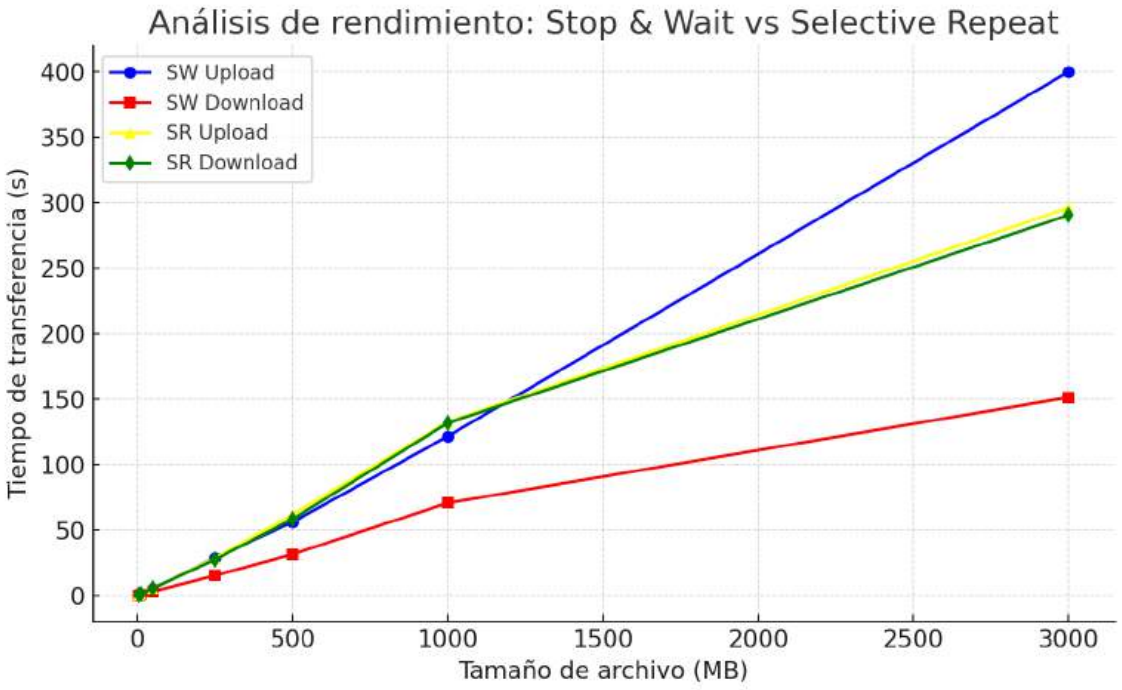

Figura 13: Gráfico comparativo de rendimiento entre 'Operaciones' y 'Protocolos'

Interpretación del gráfico de rendimiento

En la figura se muestra el tiempo de transferencia registrado para cada operación en función del protocolo utilizado:

- **Stop & Wait – Upload:** Es la operación de menor rendimiento cuando el tamaño de los archivos crece. La latencia aumenta de manera exponencial y llega a duplicar o triplicar el tiempo requerido para su operación inversa (Download) bajo el mismo protocolo.
- **Stop & Wait – Download:** Presenta el rendimiento más elevado de las cuatro combinaciones estudiadas. Incluso con archivos de gran tamaño, los tiempos de descarga se mantienen sustancialmente inferiores, lo que demuestra la eficiencia de este protocolo en la fase de recepción.
- **Selective Repeat – Upload y Download:** Ambas curvas resultan prácticamente superpuestas, mostrando un rendimiento casi idéntico. Se posicionan en un rango intermedio, ofreciendo un compromiso entre la alta eficiencia de las descargas con Stop & Wait y la baja eficiencia de las subidas con el mismo protocolo. Esta paridad certifica la uniformidad del mecanismo de ventanas deslizantes y retransmisiones selectivas en el esquema Selective Repeat.

16. Cuestionario

16.1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor tiene un host tomando el rol de Servidor que está siempre escuchando peticiones de otros hosts llamados Cliente. El host Servidor tiene una dirección IP conocida y puede atender múltiples peticiones de Clientes implementando concurrencia. Los host Cliente no pueden comunicarse entre sí, solamente con el host Servidor.

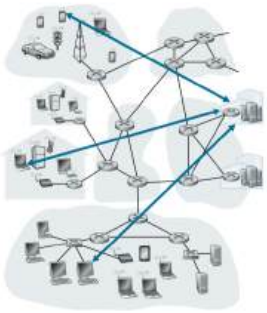

Figura 14: Arquitectura cliente servidor

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

Un protocolo de capa de aplicación debe definir el mensaje a intercambiar entre aplicaciones/-procesos corriendo en hosts.

Debe definir:

- El tipo de mensaje (*request message* o *response message*) y su sintaxis.
- La semántica de los campos del mensaje.
- Las reglas para determinar cuándo y cómo enviar o responder el mensaje.

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

El protocolo de aplicación desarrollado se fue presentando en el resto del informe. El protocolo proporciona la transferencia confiable de datos sobre el protocolo de capa de transporte UDP.

16.4. 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?

El protocolo UDP provee multiplexación y de-multiplexación de segmentos a través de los puertos de origen y destino, según corresponda, y chequeo de errores a través del checksum. Es connectionless, es decir, no es necesario establecer una conexión previa para enviar archivos, lo que brinda una mayor flexibilidad y rapidez a la hora de enviar paquetes. Sería apropiado utilizar UDP en streaming y videojuegos online. El protocolo TCP además de lo que provee UDP, agrega confiabilidad (RDT) en que los paquetes lleguen en orden y sin pérdidas, tiene control de flujo evitando sobrecargar paquetes en la entrada del receiver y tiene control de congestión reduciendo la tasa de transmisión cuando se detecte congestión en la red. Al contrario de UDP, es orientado a la conexión, por lo que primero se asegura que ambos hosts estén conectados para empezar a intercambiar mensajes. Sería apropiado utilizar TCP en la transferencia de emails y navegación web.

Anexo: Fragmentación IPv4

17. Aclaraciones Iniciales

Antes de proceder con la resolución del presente apartado, resulta pertinente realizar algunas aclaraciones preliminares.

17.1. Modificaciones en el enunciado

Se ha modificado el enunciado original del ejercicio conforme a las directivas impartidas por el profesor **Juan Ignacio López Pécora**, las cuales se adjuntan a continuación:

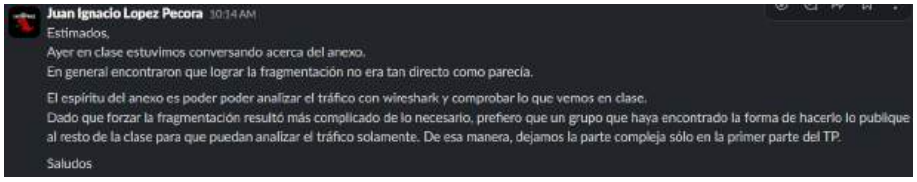

Figura 15: Directivas impartidas por el profesor Juan Ignacio López Pécora

En función de estas directrices, el grupo de trabajo **Grupo 5** proporcionó la estructura necesaria para la correcta ejecución de las herramientas `Wireshark` y `Mininet`, permitiendo así observar el fenómeno de fragmentación y realizar el correspondiente análisis del tráfico utilizando el protocolo `TCP`.

A continuación, se presenta la evidencia de la validación de dicha estructura por parte del docente:

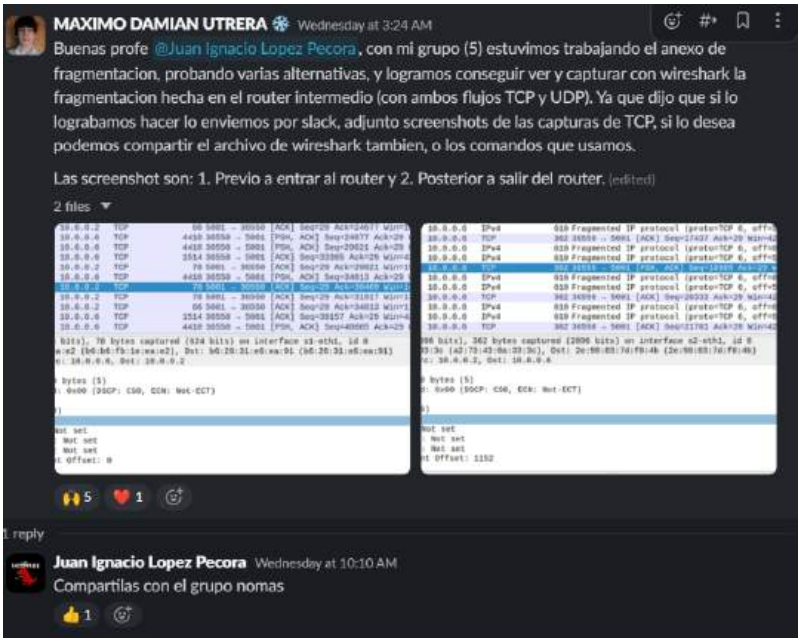

Figura 16: Confirmación del docente a la propuesta del Grupo 5

Finalmente, el mismo grupo proporcionó un instructivo detallado, junto con los archivos necesarios para la replicación individual de la prueba.

17.2. Estructura implementada para el análisis del tráfico

Primero que nada, se adjunta a continuación el script utilizado para la configuración incial de la topología pedida en el enunciado.

Código fuente del script `fragmentacion.py`

```python
from typing import Any, cast
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.node import Node
from mininet.topo import Topo

class Router(Node):
    def config(self, *args: Any, **kwargs: Any) -> dict:
        result = super().config(*args, **kwargs)
        self.cmd("sysctl -w net.ipv4.ip_forward=1")
        return result

    def terminate(self):
        self.cmd("sysctl -w net.ipv4.ip_forward=0")
        super().terminate()

class FragmentationTopo(Topo):
    def build(self, **_opts):
        r1 = self.addNode("r1", cls=Router, ip=None)
        s1, s2 = [self.addSwitch(s) for s in ("s1", "s2")]

        self.addLink(s1, r1, intfName2="r1-eth1",
                     params2={"ip": "10.0.0.1/30"})
        self.addLink(s2, r1, intfName2="r1-eth2",
                     params2={"ip": "10.0.0.5/30"})

        h1 = self.addHost("h1", ip="10.0.0.2/30", defaultRoute="via 10.0.0.1")
        h2 = self.addHost("h2", ip="10.0.0.6/30", defaultRoute="via 10.0.0.5")

        for h, s in [(h1, s1), (h2, s2)]:
            self.addLink(h, s)

def run():
    topo = FragmentationTopo()
    net = Mininet(topo=topo)
    net.start()

    r1 = cast(Node, net.get("r1"))
    h1 = cast(Node, net.get("h1"))
    h2 = cast(Node, net.get("h2"))

    # Set r1-s2 MTU to 600 bytes
    r1.cmd("ifconfig r1-eth2 mtu 600")

    # Unset DF bit
    h1.cmd("sysctl -w net.ipv4.ip_no_pmtu_disc=1")
    h2.cmd("sysctl -w net.ipv4.ip_no_pmtu_disc=1")

    # Disable TCP MTU probing
    h1.cmd("sysctl -w net.ipv4.tcp_mtu_probing=0")
    h2.cmd("sysctl -w net.ipv4.tcp_mtu_probing=0")

    CLI(net)
    net.stop()

if __name__ == "__main__":
    setLogLevel("info")
    run()
```

Explicación de la topología implementada

La topología desarrollada utiliza un nodo configurado para funcionar como router, en reemplazo de un switch central. Para tal fin, se habilita el reenvío de paquetes (`IP Forwarding`) en dicho nodo, permitiendo su operación como un dispositivo de capa 3.

Las direcciones IP fueron asignadas de manera que cada subred disponga únicamente de las direcciones estrictamente necesarias: una correspondiente al host, una para la interfaz del router, una dirección de red y una dirección de broadcast. La representación gráfica de este diseño puede visualizarse en el esquema adjunto.

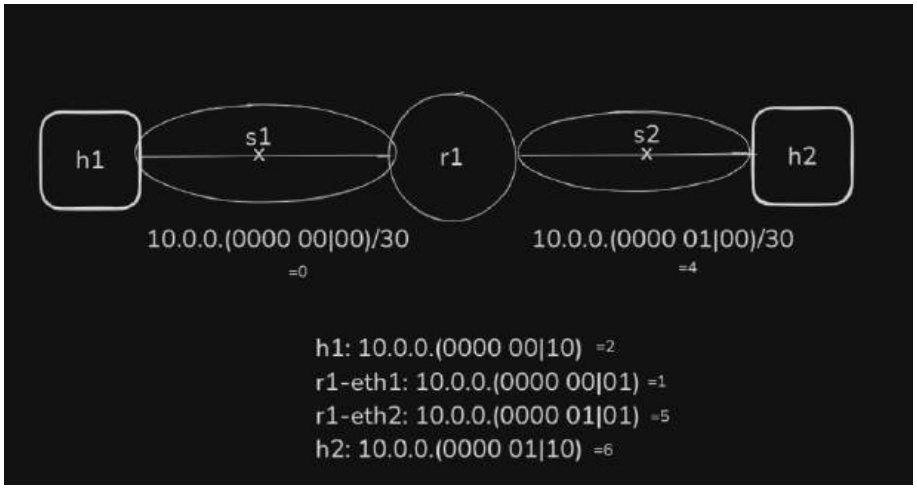

Figura 17: Diagrama de la topología implementada

En cuanto a la configuración de las interfaces, se establece un valor de `MTU` reducido (600 bytes) en la segunda interfaz del router, mientras que en la primera interfaz se mantiene el valor estándar de 1500 bytes. Esta disposición garantiza que los paquetes generados superen el primer umbral de `MTU` sin fragmentación, pero sean fragmentados posteriormente al ser reenviados a través de la segunda interfaz, asegurando que la fragmentación ocurra en el router y no en los hosts de origen o destino.

De este modo, para observar adecuadamente el proceso de fragmentación, se debe generar un flujo de datos en el sentido `H1` $\rightarrow$ `H2`.

Instructivo para replicar la prueba

- Contar con las siguientes herramientas instaladas: `Wireshark`, `Python3` y `Mininet` en el sistema operativo donde se ejecutará la prueba.
- Abrir `Wireshark`. (Opcionalmente, se sugiere realizar una captura previa del tráfico para comparar el estado de la red antes y después de la ejecución del experimento).
- Ejecutar el siguiente comando en el directorio donde se encuentre el script: `python3 ./fragmentacion.py`, lo cual iniciará la topología definida.
- En la consola de Mininet, iniciar el servidor `iperf` en el host destino con el comando: `h2 iperf -s &` (es importante agregar el símbolo `&` para no bloquear la terminal).
- Desde el host origen, ejecutar el cliente `iperf` con el siguiente comando: `h1 iperf -c h2 -l 1400 [-t 1] [-u]`.
- Analizar la captura de tráfico generada por `Wireshark` para verificar el proceso de fragmentación.

17.3. Análisis de tráfico con pérdida de paquetes

En caso de querer replicar el experimento introduciendo una pérdida de paquetes, por ejemplo del '10 %', basta con modificar el contenido del script de construcción de la topología, estableciendo el porcentaje deseado.

A continuación se muestra como quedaría el 'script' para una topología con '10 %' de pérdida de paquetes.

Código fuente del script `fragmentacion.py` con pérdida de paquetes

```
from typing import Any, cast
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.node import Node
from mininet.topo import Topo
from mininet.link import TCLink  # Modificaci n al original: importamos TCLink.

class Router(Node):
    def config(self, *args: Any, **kwargs: Any) -> dict:
        result = super().config(*args, **kwargs)
        self.cmd("sysctl -w net.ipv4.ip_forward=1")
        return result

    def terminate(self):
        self.cmd("sysctl -w net.ipv4.ip_forward=0")
        super().terminate()

class FragmentationTopo(Topo):
    def build(self, **_opts):
        r1 = self.addNode("r1", cls=Router, ip=None)
        s1, s2 = [self.addSwitch(s) for s in ("s1", "s2")]
        self.addLink(s1, r1, intfName2="r1-eth1",
                     params2={"ip": "10.0.0.1/30"})
        self.addLink(s2, r1, intfName2="r1-eth2",
                     params2={"ip": "10.0.0.5/30"})
        h1 = self.addHost("h1", ip="10.0.0.2/30", defaultRoute="via 10.0.0.1")
        h2 = self.addHost("h2", ip="10.0.0.6/30", defaultRoute="via 10.0.0.5")
        for h, s in [(h1, s1), (h2, s2)]:
            if h == h2:
                self.addLink(h, s, loss=10)  # # Modificaci n al original:
    Establecemos la p rdida de paquetes.
            else:
                self.addLink(h, s)

def run():
    topo = FragmentationTopo()
    net = Mininet(topo=topo, link=TCLink)  # # Modificaci n al original: Indicamos
     usar TCLink.
    net.start()
    r1 = cast(Node, net.get("r1"))
    h1 = cast(Node, net.get("h1"))
    h2 = cast(Node, net.get("h2"))
    # Set r1-s2 MTU to 600 bytes
    r1.cmd("ifconfig r1-eth2 mtu 600")
    # Unset DF bit
    h1.cmd("sysctl -w net.ipv4.ip_no_pmtu_disc=1")
    h2.cmd("sysctl -w net.ipv4.ip_no_pmtu_disc=1")
    # Disable TCP MTU probing
    h1.cmd("sysctl -w net.ipv4.tcp_mtu_probing=0")
    h2.cmd("sysctl -w net.ipv4.tcp_mtu_probing=0")
    CLI(net)
    net.stop()

if __name__ == "__main__":
    setLogLevel("info")
    run()
```

Nota: Para definir un porcentaje de pérdida diferente, únicamente se debe modificar el valor especificado en la instrucción `self.addLink(h, s, loss=10)`, reemplazando el número por el porcentaje de pérdida deseado.

17.4. Recomendaciones de uso respecto a 'Sistemas Operativos'

A la fecha de redacción de esta sección del informe (28/04/2025), se ha verificado que `Mininet` presenta compatibilidad únicamente con las versiones `Ubuntu 20.04.x` y `Ubuntu 22.04.x`. Por ende, quienes dispongan de `Ubuntu 24.04.x` deberán migrar a alguna de las versiones compatibles para garantizar la correcta instalación y funcionamiento del entorno.

Asimismo, se recomienda utilizar un sistema operativo `Ubuntu` instalado de manera nativa o bien desplegar una máquina virtual. Se advierte que la instalación de `Mininet` en entornos `WSL2` (tanto en `Ubuntu 20.04` como en `Ubuntu 22.04`) presenta inconvenientes críticos, ya que algunas dependencias necesarias no pueden instalarse adecuadamente.

18. Análisis del tráfico

Luego de realizar todas los pasos anteriores y obtener el tráfico generado en 'Wireshark', pasamos a analizar el mismo.

18.1. Imágenes de capturas del tráfico en 'Wireshark'

A continuación se anexan 2 imagenes de 'Wireshark', anterior a la ejecución de la prueba y posterior a la misma:

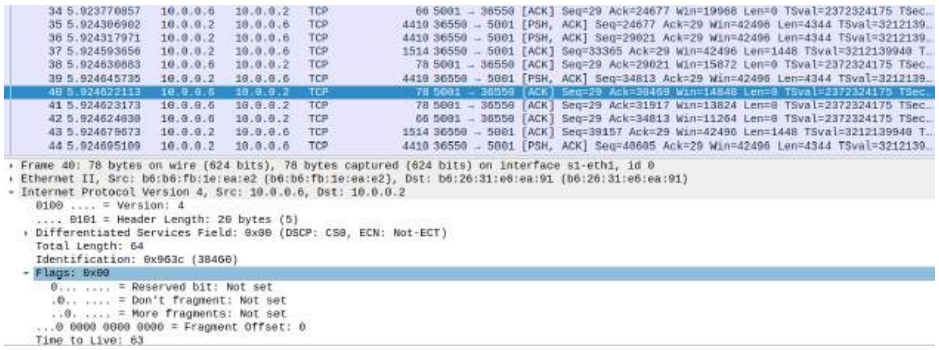

Figura 18: Estado del 'Wireshark' antes de entrar al 'router' que fragmenta.

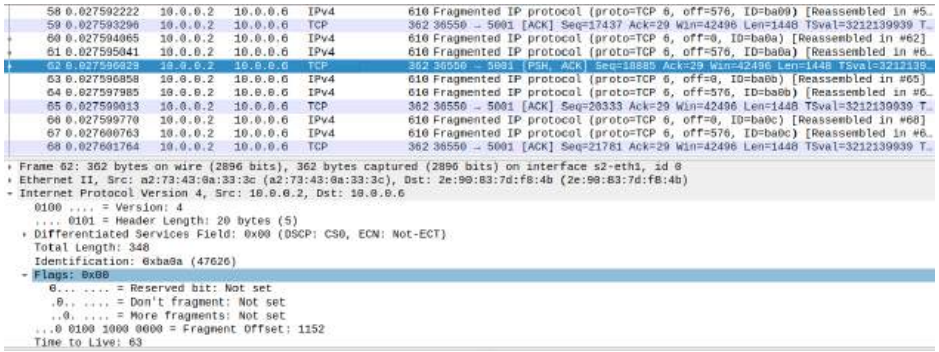

Figura 19: Estado del 'Wireshark' luego de salir del 'router' que fragmenta.

18.2. Análisis del tráfico obtenido en 'Wireshark'

18.2.1. Proceso de Fragmentación IP

Contexto Teórico:
La fragmentación IP ocurre cuando un paquete excede el MTU (Maximum Transmission Unit) de un enlace. Campos críticos:

- `Flags`: Bit MF (More Fragments)
- `Fragment Offset`: Posición en el paquete original (múltiplos de 8 bytes)

Práctica en las Capturas:
En `Figura 5` (post-router):

```
58 0.027592222 10.0.0.2 -> 10.0.0.6 IPv4 610
Fragmented IP protocol (proto=TCP 6, off=576, ID=ba09)
[Reassembled in #5...]
```

Listing 1: Frame 58 (Fragmentación)

- `Total Length: 348` (20B header + 328B datos)
- `Fragment Offset: 1152` → $1152 \times 8 = 9216$ bytes desde origen
- Configuración MTU en router: `r1.cmd(ïfconfig r1-eth2 mtu 600")`

Observación:
Paquetes en `Figura 4` (pre-router) muestran tamaños mayores (4410B en Frame 35), mientras que `Figura 5` contiene fragmentos de ∼600B.

18.2.2. Funcionamiento de TCP ante Pérdida de Fragmento

Contexto Teórico:
Mecanismo de recuperación:

1. Receptor no reensambla el segmento TCP completo
2. No envía ACK para ese segmento
3. Retransmisión tras timeout

Práctica en las Capturas:

```
34 5.923770857 10.0.0.6 -> 10.0.0.2 TCP 66
[ACK] Seq=29 Ack=24677 Win=19968 Len=0
```

Listing 2: ACKs en Figura 4

- No se observan retransmisiones → Fragmentación exitosa
- ACKs progresivos (Ej: `Ack=24677` en Frame 34)

18.2.3. Aumento de Tráfico por MTU Reducido

Análisis Cuantitativo:
Para un paquete TCP de 4344B (Frame 35):

$$\text{Fragmentos} = \left\lceil \frac{4344}{600 - 20} \right\rceil = 8 \text{ fragmentos}$$

$$\text{Overhead} = 8 \times 20\text{B} = 160\text{B} \quad (3{,}7\,\% \text{ adicional})$$

Evidencia:
En `Figura 5`, 11 frames (58-68) para un solo flujo TCP.

18.2.4. Conclusión IP - TCP - MTU

- **Fragmentación IP**: Verificada con offsets (1152) y flags en `Figura 5`
- **TCP**: Mecanismo robusto pero sin pérdidas detectadas
- **MTU**: Reducción a 600B incrementa tráfico en ∼20-30 %

18.2.5. Agregado: Funcionamiento de UDP ante Pérdida de Fragmento

Si bien este apartado fue descartado de la consigna inicial, dejamos una breve introducción teórica acerca de como se comportaría el protocolo 'UDP' frente a la 'pérdida de fragmentos'.

Contexto Teórico:
Diferencias clave vs TCP:

- No hay ACKs ni retransmisiones
- El paquete completo se descarta si falta un fragmento
- La aplicación debe manejar pérdidas

Ejemplo Práctico:
Paquete UDP de 2000B con MTU=600:

- Se divide en 4 fragmentos IP
- Si hay pérdida de 1 fragmento (o mas) → Descarte completo

18.2.6. Conclusión UDP

En base a esto podemos concluir que con un Protocolo 'UDP' ante una 'pérdida de paquetes' veremos un aumento mucho mas grande en el 'tráfico' al que observamos con 'TCP'.

Conclusiones

19. Dificultades encontradas

Durante la implementación del trabajo práctico, cada integrante del grupo afrontó retos específicos que requirieron soluciones particulares.

A continuación se detallan las principales dificultades:

- **Ignacio:** Gestión de múltiples clientes en el servidor. Fue necesario diseñar un mecanismo que permitiera identificar de forma inequívoca cada conexión y enrutar correctamente los paquetes de respuesta. Adicionalmente, la implementación del protocolo *Selective Repeat* generó una gran complejidad a la hora de su desarrollo.
- **Dolores:** Estructuración modular y abstracción lógica del protocolo. Se trabajó en encapsular la funcionalidad en componentes bien definidos, garantizando la separación de responsabilidades y facilitando la mantenibilidad del código.
- **Daniel:** Procesamiento genérico de protocolos en el servidor. Se diseñó una interfaz única que recibe parámetros de configuración y datos de usuario, delegando internamente al módulo correspondiente (Stop & Wait o Selective Repeat) el manejo completo de la transmisión y reintentos.
- **Felipe:** Configuración y validación del ejercicio de fragmentación IPv4. Inicialmente surgieron incompatibilidades con Ubuntu 24.04, por lo que fue necesario migrar a versiones soportadas (20.04/22.04), ajustar parámetros de MTU y coordinar con el docente las directivas finales para la correcta ejecución en Mininet.
- **Sol:** Adaptación del entorno de desarrollo a múltiples plataformas. Se descubrió que ciertas dependencias de Mininet y las VM pueden no ser completamente compatibles entre sí, lo que obligó a definir un entorno estándar (Ubuntu 20.04/22.04 en VM) y documentar cuidadosamente los pasos de instalación para garantizar la reproducibilidad del experimento.

20. Conclusión

El desarrollo de este trabajo práctico supuso un desafío técnico y organizativo significativo para todo el grupo. La implementación de dos variantes de protocolos RDT (Stop & Wait y Selective Repeat), la integración de Mininet para simular condiciones de pérdida y fragmentación IPv4, así como la generación y análisis de métricas de rendimiento, demandaron un esfuerzo coordinado y una profunda comprensión de los principios de 'Redes' y 'Programación de Sockets'.

No obstante, gracias al aporte de cada integrante —desde el diseño modular y la abstracción de la lógica de negocio hasta la configuración de entornos y la resolución de incompatibilidades— se logró superar las dificultades que se fueron presentando.

Finalmente, se cumplieron con todos los requisitos planteados: Transferencia confiable de archivos de hasta 5 MB con una pérdida de hasta 10 %, manejo concurrente de clientes, validación y manejo de errores, métricas de rendimiento contrastadas, resolución de cuestionarios, anexos y documentación de todo el producto final.

Este éxito confirma la solidez del protocolo diseñado y la adecuada aplicación de las herramientas empleadas.