



Trabajo Práctico 1: File Transfer

Grupo 2

[TA048] Redes
Primer cuatrimestre de 2025

Alumno	Padrón	Email
AVALOS, Victoria	108434	vavalos@fi.uba.ar
CASTRO MARTINEZ, Jose Ignacio	106957	jcastrom@fi.uba.ar
CIPRIANO, Victor	106593	vcipriano@fi.uba.ar
DEALBERA, Pablo Andres	106858	pdealbera@fi.uba.ar
DIEM, Walter Gabriel	105618	wdiem@fi.uba.ar

Índice

1. Introducción	3
2. Hipótesis y suposiciones realizadas	3
3. Implementación	4
3.1. Topología propia LinearEnds	4
3.2. Especificación del protocolo Stop-and-Wait	4
3.2.1. General	4
3.2.2. Handshake	5
3.2.3. Etapa de configuración y Transferencia	6
3.2.4. Cierre	7
3.2.5. Ciclo de Vida de Upload	8
3.2.6. Análisis del Ciclo de Vida de Upload de una transferencia con Stop-and-Wait	9
3.2.7. Ciclo de Vida de Download	10
3.2.8. Análisis del Ciclo de Vida de Download de una transferencia con Stop-and-Wait	11
3.3. Especificación del protocolo Go-Back-N	12
3.3.1. General	12
3.3.2. Ciclo de Vida de Upload	13
3.3.3. Análisis del Ciclo de Vida de Upload de una transferencia con Go-Back-N .	14
3.3.4. Ciclo de Vida de Download	15
3.3.5. Análisis del Ciclo de Vida de Download de una transferencia con Go-Back-N	15
4. Pruebas	16
4.1. Casos de error	16
4.2. Stop & Wait	17
4.2.1. Stop & Wait sin pérdida de paquetes	18
4.2.2. Stop & Wait con pérdida de paquetes del 10 %	22
4.3. Go Back N	24
4.4. Pruebas automatizadas	26
5. Preguntas a Responder	27
5.1. Describa la arquitectura Cliente-Servidor.	27
5.1.1. Características	27
5.1.2. Ventajas	27
5.1.3. Desventajas	27
5.2. ¿Cuál es la función de un protocolo de capa de aplicación?	27
5.3. Detalle el protocolo de aplicación desarrollado en este trabajo.	28
5.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP.	28
5.4.1. ¿Qué servicios proveen dichos protocolos?	28
5.4.2. ¿Cuáles son sus características?	28
5.4.3. ¿Cuándo es apropiado utilizar cada uno?	29
6. Anexo: Fragmentación IPv4	29
6.1. Consideraciones iniciales	29
6.2. Análisis	30
6.2.1. Topología	30
6.2.2. Proceso de fragmentación	30
6.2.3. Funcionamiento de TCP ante la pérdida de un fragmento	31
6.2.4. Funcionamiento de UDP ante la pérdida de un fragmento	32
6.2.5. Aumento de tráfico al reducirse el MTU mínimo de la red.	34

7. Dificultades Encontradas	35
7.1. Mininet	35
7.2. Protocolo SAW	35
7.3. Protocolo GBN	35
7.4. Plugin de Wireshark	35
8. Conclusión	35
9. Bibliografía	37

1. Introducción

En el presente trabajo práctico se desarrolla una aplicación de red para la transferencia de archivos, basada en una arquitectura cliente-servidor, donde un servidor puede atender las operaciones de múltiples clientes de manera concurrente, con el objetivo principal de implementar mecanismos de transferencia de datos confiable. Para ello, se trabajará sobre la capa de transporte, utilizando específicamente el protocolo UDP, lo cual requiere diseñar e implementar soluciones personalizadas que garanticen la entrega de datos confiable (RDT - Reliable Data Transfer).

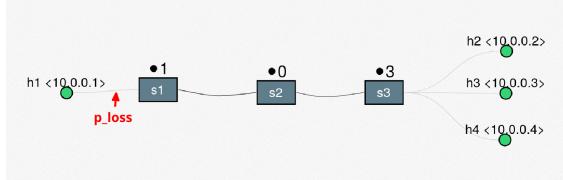
En este marco, se explorarán los principios del concepto de RDT, implementando dos de sus variantes: Stop-and-Wait (SAW) y Go-Back-N (GBN). La comunicación entre procesos se llevará a cabo mediante la interfaz de sockets de Python, y se utilizará la herramienta Mininet para simular diferentes condiciones de red y evaluar el comportamiento de las implementaciones, incluyendo escenarios con pérdida de paquetes.

2. Hipótesis y suposiciones realizadas

- La carga/descarga no va a conservar la metadata del archivo. Es decir, si yo descargo un archivo, ese archivo va a tener metadata como si yo hubiera creado el archivo desde cero usando ‘touch archivo’.
- Si el cliente utiliza otro protocolo para comunicarse con el server, el server debe rechazar este pedido, esta condición se la denominará **protocol mismatch**. El header tendrá un campo dedicado a la versión del protocolo.
- El argumento de FILENAME será opcional, en caso de no estar, se utiliza el nombre original del archivo.
- Por simplicidad, vamos a guardar todos los archivos en DIRPATH sin ningún nivel de subdirectorios.
- Si no se provee un DIRPATH para el storage del servidor, se utiliza el directorio actual.
- Se verifica que hayan 100 megas disponibles en el disco para realizar un upload al server.
- Si se cancela la carga o la descarga sin haberse finalizado correctamente, el archivo se borra.
- Si no se provee puerto por defecto se usa el 7777.
- Si no se provee un host por defecto será 127.0.0.1.
- El cliente hace un early bind con cualquier puerto UDP abierto disponible para uso justo antes de enviar el primer mensaje.
- Tanto el server como el cliente finalizan la conexión después de 30 segundos sin respuesta. También la conexión puede ser finalizada por exceder 300 retransmisiones del mismo paquete de manera continua, esta es una consideración de fácil cambio (dado que está definido en un archivo de constantes).
- Se eliminan los archivos corruptos no enviados/recibidos de manera completa.

3. Implementación

3.1. Topología propia LinearEnds

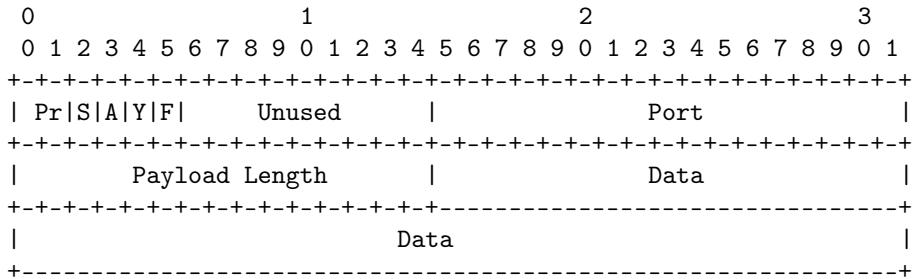


La topología diseñada es una red lineal con 1 host servidor (izquierda) conectado a 3 switches en serie cuyo último switch esta conectado a **n** hosts clientes (derecha). El primer enlace (el conectado entre el servidor y el primer switch) tiene configurado un packet loss del 10 % configurado de forma simétrica 5 % en cada extremo del enlace.

3.2. Especificación del protocolo Stop-and-Wait

3.2.1. General

Formato de mensaje SaW:



Donde:

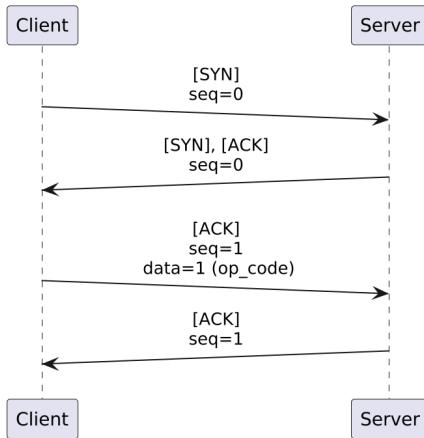
- **Pr:** son los bits que determinan el protocolo (0 para SAW y 1 para GBN)
- **S:** es el bit field que representa el sequence number, puede ser 0 o 1.
- **A:** es el flag booleano que representa si el mensaje es un ACK o no.
- **Y:** es el flag booleano que representa si el mensaje es SYN o no.
- **F:** es el flag booleano que representa si el mensaje es FIN o no.
- **Unused:** es una porción de bits que no se usa y queda como padding para completar el tamaño de un short unsigned int (2 bytes).
- **Port:** el puerto con el que el receptor del mensaje debe comunicarse, se usa para el cambio de puerto.
- **Payload length:** short unsigned int que representa el largo de la data que contiene el paquete.
- **Data:** información binaria (bytes) que son los datos a transportar.

El header tiene un tamaño de 6 bytes.

Tamaño máximo de payload

El tamaño máximo de payload permitido por el protocolo está basado en el MTU establecido en el estándar Ethernet, es decir, 1500 bytes. Teniendo en cuenta que esto debería ser lo máximo que transporta un paquete de IP, se toman estos 1500 bytes como base, se le resta el tamaño de un header máximo de IP (60 bytes), lo mismo con el header de UDP (8 bytes) y se le resta el tamaño de header del protocolo (6 bytes). Quedando así un payload máximo de 1426 bytes.

3.2.2. Handshake



Este handshake se usa para inicialización de recursos del servidor y check de protocolo (que no haya protocol mismatch).

Se usa la nomenclatura S para mencionar al servidor y C para el cliente.

Mensajes para caso Download y caso Upload:

1. C → S: con flag SYN para declarar una solicitud de conexión y el protocolo

Flujo normal (mismo protocolo):

2. S → C: con flag de SYN y ACK para declarar que se acepta la conexión y el puerto donde se va a escuchar el resto.
3. C → S: con flag ACK al mismo welcoming socket.

Flujo de error (distinto protocolo):

2. S → C: con flag FIN para denegar la conexión por usar un protocolo distinto (protocol mismatch).

Se hace una transferencia de puerto para que el welcoming socket se encargue solamente de establecer conexiones y el nuevo puerto maneje la transferencia de datos del archivo. El último ACK de parte del cliente asegura que se recibió el puerto donde se tiene que comunicar y es seguro hacer el cambio de socket.

3.2.3. Etapa de configuración y Transferencia

El cliente ya sabe que tiene que comunicarse con el nuevo puerto.

Se envía primero la configuración para saber si la operación es válida, teniendo en cuenta casos de error, y luego se hace la transferencia.

Mensajes para caso Download y caso Upload:

1. C → S: se declara la operación (OP), que puede ser download (1) o upload (2)
2. S → C: ACK de la operación

Continuación de mensajes para caso Download:

3. Mensaje 3 C → S: filename

Flujo Normal:

4. S → C: ACK + comienzo de datos (piggybacked)
5. C → S: ACK
6. S → C: continuacion de datos

Flujo de error (no existe un archivo con ese nombre):

4. S → C: FIN, se termina la conexión

Continuación de mensajes para caso Upload:

3. C → S: filename

Flujo de error (ya existe un archivo con ese nombre):

4. S → C: FIN, se termina la conexión

Flujo normal:

4. S → C: ACK
5. C → S: filesize

Flujo de error (archivo es más grande que el tamaño máximo o [TODO] no hay más espacio en disco):

6. S → C: FIN, se termina la conexión

Flujo normal:

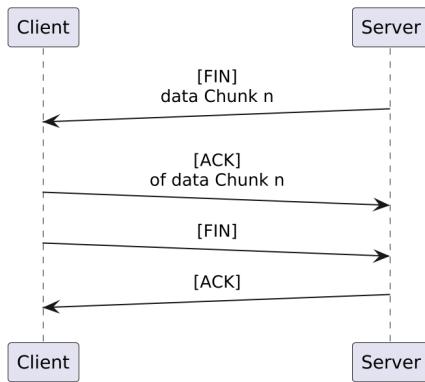
6. S → C: ACK
7. C → S: comienzo de datos
8. S → C: ACK
9. C → S: continuacion de datos

3.2.4. Cierre

El flag FIN va piggybacked con la última data para que sea más eficiente. El receptor confirma con un ACK seguido de un FIN para que el emisor sepa que le llegó la información, y por si este se pierde está el último ACK para confirmar el cierre de parte del emisor.

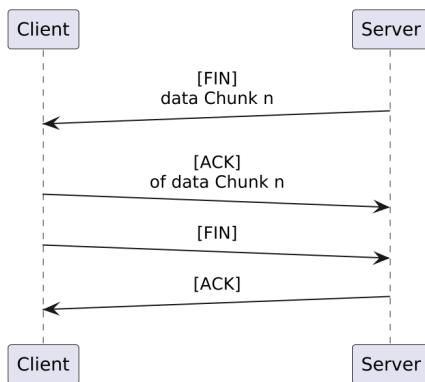
Esto garantiza que se realice el esfuerzo máximo por parte del cliente y el servidor para, en primer lugar, asegurar la recepción del último chunk y, en segundo lugar, que ambos actores sepan que la conexión se va a cerrar y poder conseguir un graceful shutdown. En el peor de los casos se puede perder el último ACK, pero cuando eso pase, se hace el esfuerzo máximo para finalizar la conexión (hasta timeout o exceso de retransmissions).

Mensajes para caso Download



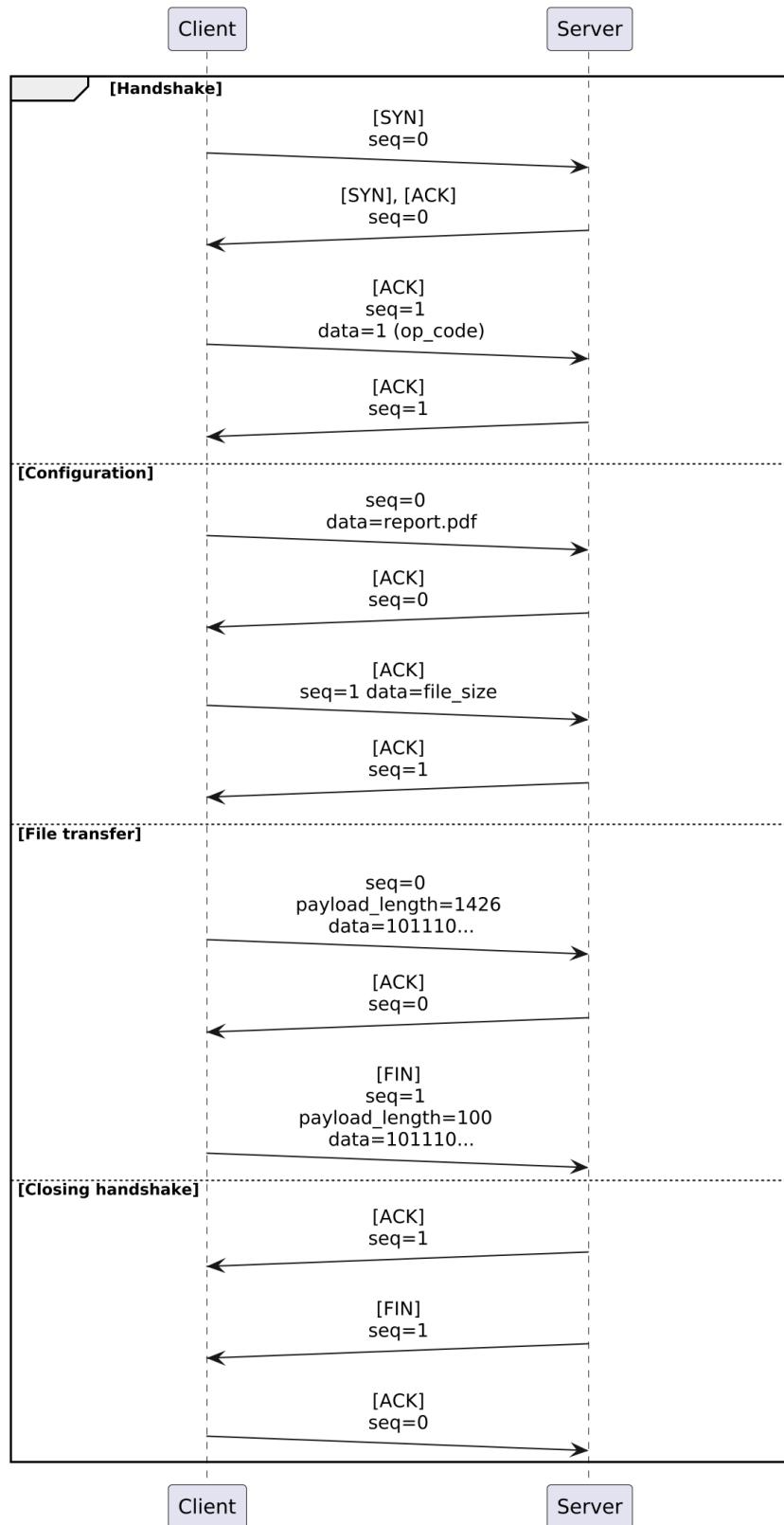
1. S → C: última data, va piggybacked el flag FIN
2. C → S: ACK
3. C → S: FIN
4. S → C: ACK

Mensajes para caso Upload:



1. C → S: última data, va piggybacked el flag FIN
2. S → C: ACK
3. S → C: FIN
4. C → S: ACK

3.2.5. Ciclo de Vida de Upload



3.2.6. Análisis del Ciclo de Vida de Upload de una transferencia con Stop-and-Wait

En esta transferencia se utiliza el protocolo **Stop-and-Wait**, que asegura la entrega de datos mediante la espera de una confirmación (ACK) por cada paquete enviado antes de continuar. El archivo transferido es ‘report.pdf’ y tiene un tamaño total de 1526 bytes.

1. Establecimiento de la conexión (Handshake):

- El **cliente** inicia la conexión enviando un paquete con el flag SYN y seq=0.
- El **servidor** responde con un paquete con flags SYN, ACK y seq=0.
- El **cliente** confirma la recepción con un paquete ACK con seq=1 y datos que indican el código de operación data=2 (upload) al welcoming socket.
- El **servidor** responde con un ACK (seq=1) para confirmar la configuración de operación desde el socket al que la conexión fue transferida.

2. Configuración:

- El **cliente** envía un paquete con seq=0 y data=report.pdf, indicando el nombre del archivo como payload.
- El **servidor** responde con un paquete ACK (seq=0) para confirmar la recepción del nombre.
- El **cliente** envía otro paquete con seq=1 y data=file_size, indicando el tamaño del archivo como payload.
- El **servidor** responde con un ACK (seq=1).

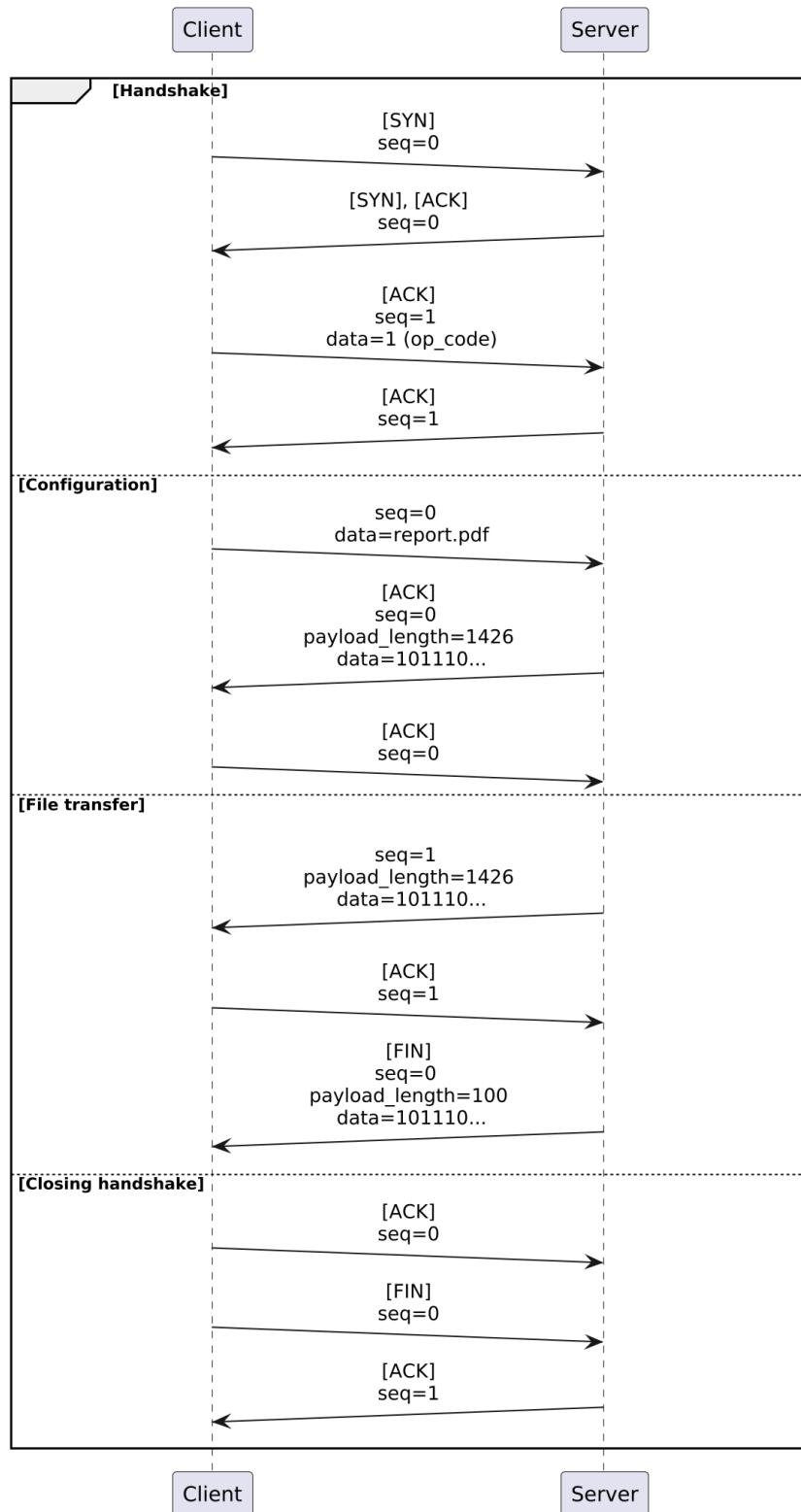
3. Transferencia del archivo:

- El **cliente** envía un chunk de datos con seq=0, payload_length=1426, y datos binarios (data=101110...).
- El **servidor** responde con un ACK (seq=0), permitiendo al cliente continuar.
- El **cliente** envía un segundo y último paquete con FIN, seq=1, payload_length=100, y más datos (data=101110...), indicando además el fin de la transferencia.

4. Cierre de la conexión (Closing handshake):

- El **servidor** confirma la recepción del último paquete con un ACK (seq=1).
- Luego, el **servidor** también inicia su cierre con un paquete FIN (seq=1).
- Finalmente, el **cliente** responde con un último ACK (seq=0), completando el cierre de la conexión.

3.2.7. Ciclo de Vida de Download



3.2.8. Análisis del Ciclo de Vida de Download de una transferencia con Stop-and-Wait

Esta transferencia ilustra el funcionamiento del protocolo Stop-and-Wait en una operación de descarga de archivos. El archivo solicitado por el cliente es report.pdf, con un tamaño total de 3000 bytes, dividido en fragmentos de máximo 1426 bytes. Cada paquete de datos enviado requiere una confirmación antes de que el servidor continúe con el siguiente.

1. Establecimiento de la conexión (Handshake):

- El cliente inicia la conexión con un paquete SYN con **seq=0**.
- El servidor responde con un paquete que incluye los flags SYN y ACK (**seq=0**).
- El cliente confirma el establecimiento de la conexión con un ACK (**seq=1**) e indica en el cuerpo del mensaje el código de operación (**data=1**), correspondiente a una descarga.
- El servidor responde con otro ACK (**seq=1**), confirmando la recepción del código de operación.

2. Configuración:

- El cliente solicita el archivo enviando un paquete con **seq=0** y **data=report.pdf** como payload.
- El servidor, en lugar de enviar sólo un ACK, responde directamente con un ACK con el primer chunk de datos: **seq=0**, **payload_length=1426**, y **data=101110....**
- El cliente confirma la recepción de este primer fragmento con un ACK (**seq=0**).

3. Transferencia del archivo:

- El servidor envía el segundo fragmento: **seq=1**, **payload_length=1426**, y **data=101110....**
- El cliente responde con un ACK (**seq=1**).

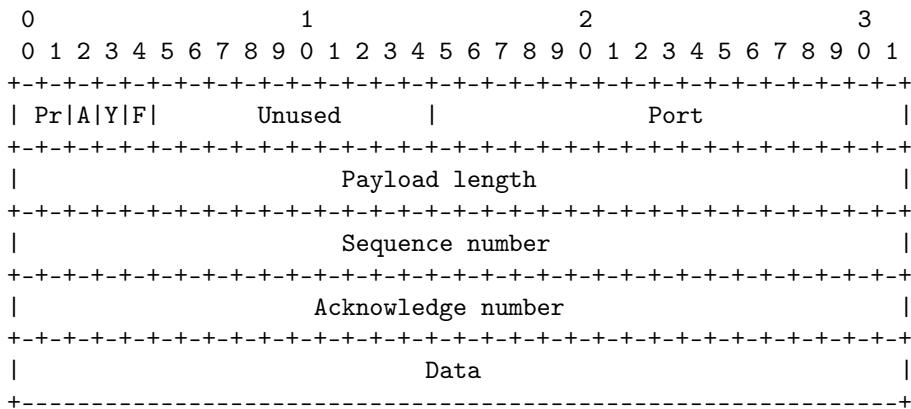
4. Cierre de la conexión (Closing handshake):

- El servidor, en el último chunk de datos indica el fin de la transmisión con un paquete FIN: **seq=0**, **payload_length=100**, y **data=101110....**
- El servidor confirma la recepción del último chunk con un ACK (**seq=0**).
- El cliente responde iniciando su propio cierre con un paquete FIN (**seq=0**).
- Finalmente, el cliente envía un último ACK (**seq=1**), completando el cierre de la conexión.

3.3. Especificación del protocolo Go-Back-N

3.3.1. General

Formato de mensaje GBN:



Donde:

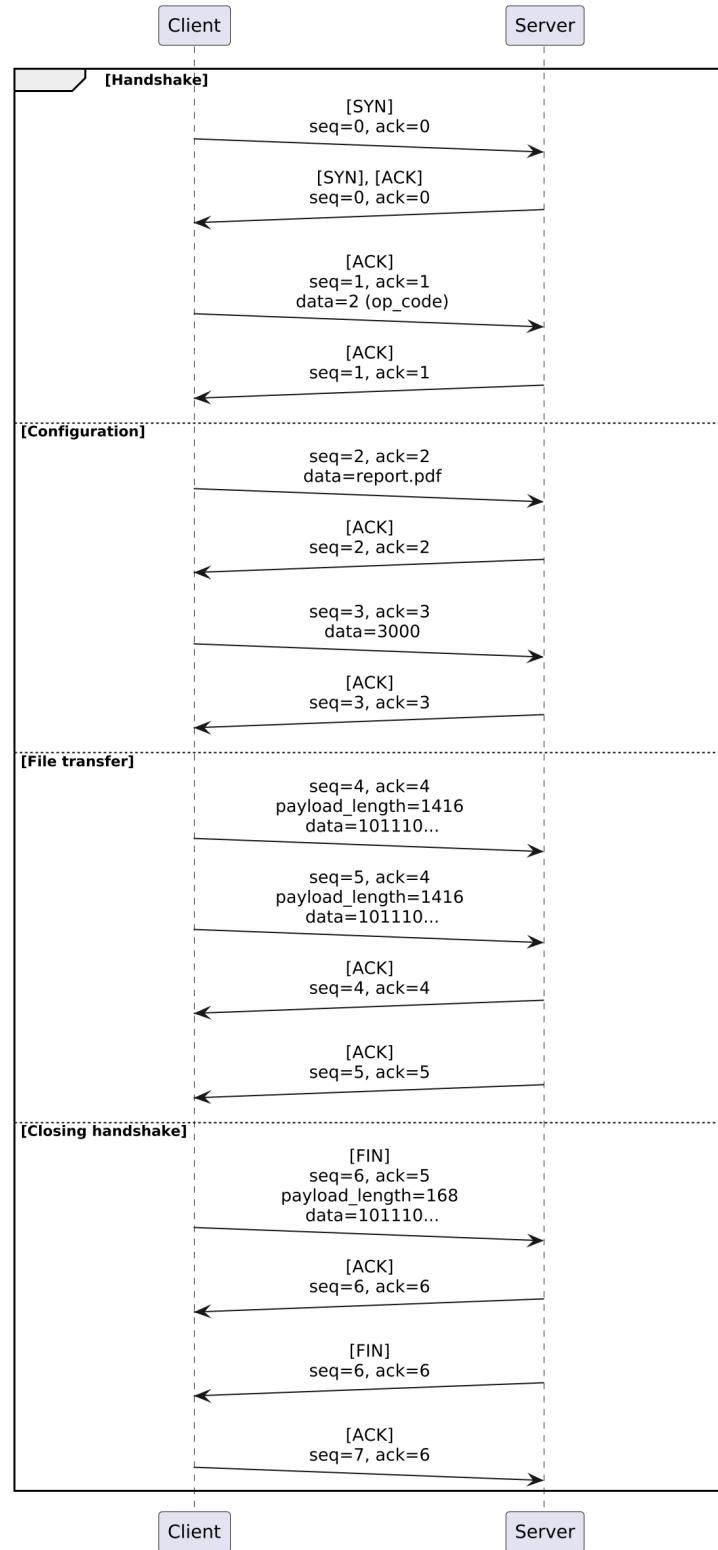
- **Pr:** son los bits que determinan el protocolo (0 para SAW y 1 para GBN)
 - **A:** es el flag booleano que representa si el mensaje es un ACK o no.
 - **Y:** es el flag booleano que representa si el mensaje es SYN o no.
 - **F:** es el flag booleano que representa si el mensaje es FIN o no.
 - **Unused:** es una porción de bits que no se usa y queda como padding para completar el tamaño de un short unsigned int (2 bytes).
 - **Port:** el puerto con el que el receptor del mensaje debe comunicarse, se usa para el cambio de puerto.
 - **Payload length:** unsigned int (4 bytes) que representa el largo de la data que contiene el paquete.
 - **Sequence number:** unsigned int (4 bytes) que representa el número de secuencia del paquete.
 - **Acknowledge number:** unsigned int (4 bytes) que representa el número de acknowledge, usado principalmente para saber cuál fue el último paquete que se recibió.
 - **Data:** información binaria (bytes) que son los datos a transportar.

El header tiene un tamaño de 16 bytes.

Tamaño máximo de payload

El tamaño se determina análogamente a SAW, sólo que esta vez se le resta el tamaño de header del protocolo (16 bytes). Quedando así un payload máximo de 1416 bytes.

3.3.2. Ciclo de Vida de Upload



Go-Back-N protocol with window of
2 packets. Client uploads file
(report.pdf of size 3000 bytes) to server

3.3.3. Análisis del Ciclo de Vida de Upload de una transferencia con Go-Back-N

Se observa el comportamiento de una transferencia de archivos con una ventana de tamaño 2. En este caso, el cliente sube un archivo (report.pdf) de 3000 bytes al servidor.

1. Establecimiento de la conexión (Handshake):

El cliente inicia la conexión enviando un paquete con el flag **SYN**, con **seq=0** y **ack=0**.

El servidor responde con un paquete con flags **SYN** y **ACK**, manteniendo los mismos valores de seq y ack.

El cliente confirma la recepción enviando un paquete **ACK** con **seq=1** y **ack=1**, incluyendo en datos la configuración del código de operación (en este caso de subida) **data=2** (**opcode**).

El servidor responde con un **ACK** para confirmar la recepción del mensaje de configuración (**seq=1, ack=1**).

2. Configuración:

El cliente envía un paquete con **seq=2, ack=2** y **data=report.pdf**, indicando el nombre del archivo a subir.

El servidor responde con un paquete **ACK** (**seq=2, ack=2**) para confirmar la petición.

El cliente envía un paquete con **seq=3, ack=3** y **data=3000**, informando el tamaño total del archivo en bytes.

El servidor confirma la recepción de esta información con otro **ACK** (**seq=3, ack=3**).

3. Transferencia del archivo:

El cliente envía el primer chunk de datos con **seq=4, ack=4, payload_length=1416** y **data=101110**.

A continuación, envía el segundo chunk con **seq=5, ack=4, payload_length=1416** y **data=101110**.

El servidor confirma la recepción del primer chunk enviando un **ACK** (**seq=4, ack=4**).

Luego, confirma el segundo chunk con otro **ACK** (**seq=5, ack=5**).

4. Cierre de la conexión (Closing handshake):

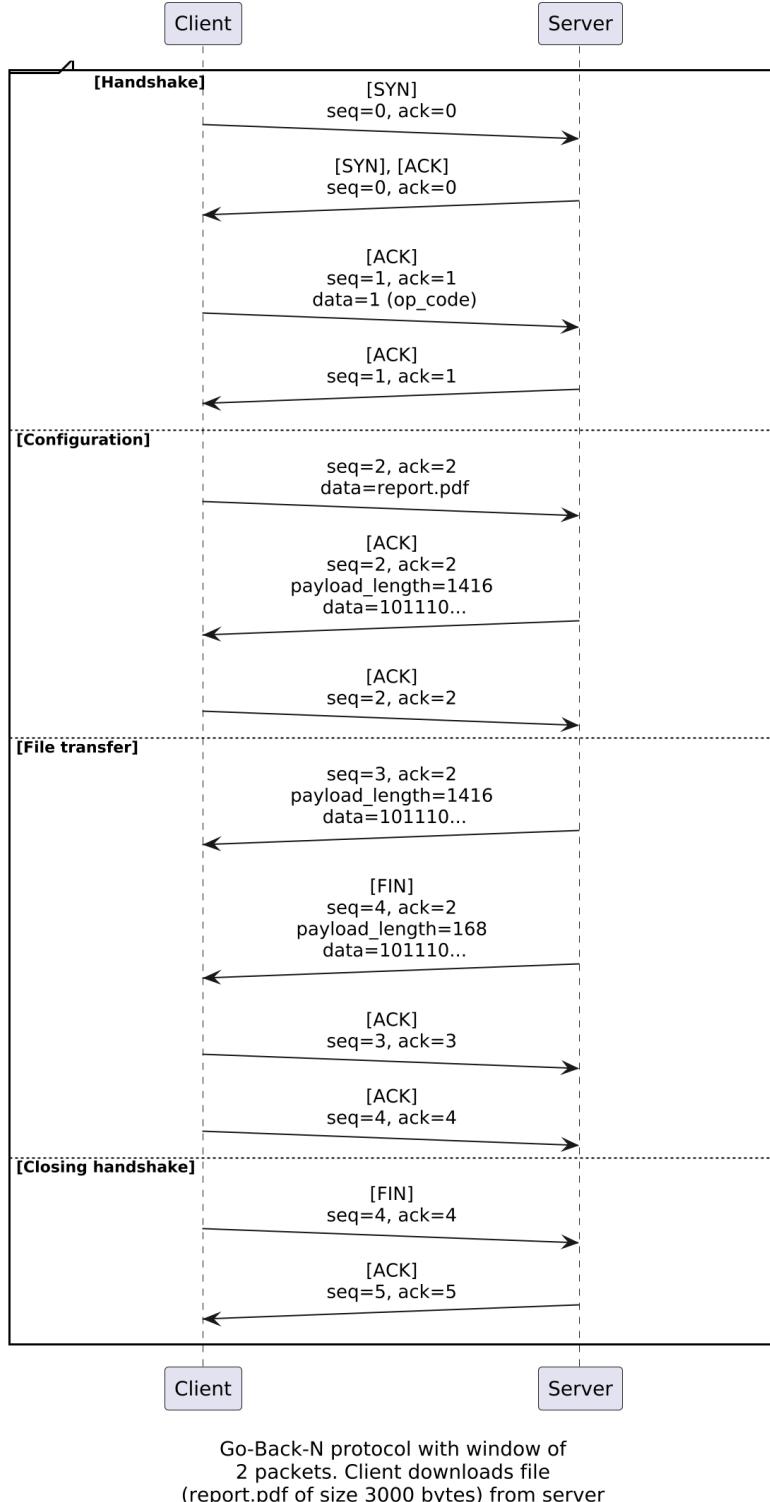
El cliente inicia el cierre enviando un paquete **FIN** con **seq=6, ack=5, payload_length=168** y **data=101110**.

El servidor responde con un **ACK** final (**seq=6, ack=6**).

El servidor envía a su vez un paquete **FIN** (**seq=6, ack=6**) para cerrar su lado de la comunicación.

Finalmente, el cliente completa el cierre con un último **ACK** (**seq=7, ack=6**).

3.3.4. Ciclo de Vida de Download



3.3.5. Análisis del Ciclo de Vida de Download de una transferencia con Go-Back-N

Se observa el comportamiento de una transferencia de archivos con una ventana de tamaño 2. En este caso, el cliente descarga un archivo ('report.pdf') de 3000 bytes desde el servidor.

1. Establecimiento de la conexión (Handshake):

- El cliente inicia la conexión enviando un paquete con el flag **SYN**, con **seq=0** y **ack=0**.
- El servidor responde con un paquete con flags **SYN** y **ACK** manteniendo los mismos valores de **seq** y **ack**.
- El cliente confirma la recepción enviando un paquete **ACK** con **seq=1** y **ack=1**, incluyendo en datos la configuración del código de operación (en este caso de descarga) **data=1** (**op_code**).
- El servidor responde con un **ACK** para confirmar la recepción del mensaje de configuración.

2. Configuración:

- El cliente envía un paquete con **seq=2**, **ack=2** y **data=report.pdf**, indicando el nombre del archivo a descargar.
- El servidor responde con un paquete de datos con **seq=2**, **ack=2**, una **size** de 1416 bytes y los primeros bits del archivo.
- El cliente confirma la recepción con un **ACK** correspondiente.

3. Transferencia del archivo:

- El servidor envía el segundo **chunk** de datos (**seq=3**, **ack=2**), también de 1416 bytes.
- Posteriormente, se envía un paquete con el flag **FIN** (**seq=4**, **ack=2**, **payload_length=168**), marcando el fin de la transferencia.
- El cliente responde con dos **ACK**, uno para cada paquete recibido correctamente: **seq=3**, **ack=3** y **seq=4**, **ack=4**.

4. Cierre de la conexión (Closing handshake):

- El cliente envía un **FIN** para finalizar su lado de la comunicación (**seq=4**, **ack=4**).
- El servidor responde con un **ACK** final (**seq=5**, **ack=5**), completando el cierre de la conexión de manera ordenada.

4. Pruebas

Se presentan capturas de diferentes casos de uso de la aplicación.

4.1. Casos de error

- Protocol Mismatch

```

"Node: h1"                               "Node: h2"
root@vicky-Lenovo-ideapad-330-15IKB:/home/vicky/Escritorio/TP1-Rede
s# e
./src/start-server.py -H 10.0.0.1 -s ./tmp/server -v -r saw
[17:40:38] [INFO] Server started
[17:40:38] [DEBUG] Protocol: saw
[17:40:38] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.0.1:7777)
[17:40:42] [INFO] [ACCEP] Rejecting client Address(10.0.1.1:54700) due to protocol mismatch, expected saw
[17:40:42] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.0.1:7777)
s# e
./src/upload.py -H 10.0.0.1 -s ./tmp/client/descarga.pdf -v -r gbn
[17:40:42] [DEBUG] Running on Address(0.0.0.0:54700)
[17:40:42] [INFO] Client started for upload
[17:40:42] [DEBUG] Protocol: gbn
[17:40:42] [DEBUG] UDP socket ready
[17:40:42] [DEBUG] Starting handshake
[17:40:42] [DEBUG] Requesting connection to Address(10.0.0.1:7777)
[17:40:42] [ERROR] Protocol mismatch
[17:40:42] [ERROR] Connection refused from server
[17:40:42] [INFO] Stopping
[17:40:42] [INFO] Press Enter to finish
  
```

Figura 1: Ejemplo de protocol mismatch.

En caso de que un cliente intente conectarse con un servidor utilizando un protocolo diferente al suyo, el servidor lo rechazará. En la imagen se puede observar un ejemplo en el que un servidor que utiliza Stop & Wait rechaza a un cliente que hace una petición con Go Back N.

- Archivo ya existente

```

"Node: h1"                               "Node: h2"
root@vicky-Lenovo-ideapad-330-15IKB:/home/vicky/Escritorio/TP1-Rede s# e
/src/start-server.py -H 10.0.0.1 -s ./tmp/server -v -r saw           root@vicky-Lenovo-ideapad-330-15IKB:/home/vicky/Escritorio/TP1-Rede s# e
[17:37:31] [INFO] Server started          /src/upload.py -H 10.0.0.1 -s ./tmp/client/descarga.pdf -v -r saw
[17:37:31] [DEBUG] Protocol: saw         [17:38:33] [DEBUG] Running on Address(0.0.0.0:41162)
[17:37:31] [DEBUG] Waiting for connection on Address(10.0.0.1:7777) [17:38:33] [INFO] Client started for upload
[17:38:33] [DEBUG] Accepting connection for Address(10.0.0.1:41162) [17:38:33] [DEBUG] Protocol: saw
[17:38:33] [DEBUG] Transferred to Address(10.0.0.1:52755)          [17:38:33] [DEBUG] UDP socket ready
[17:38:33] [DEBUG] Handshake completed    [17:38:33] [DEBUG] Starting handshake
[17:38:33] [DEBUG] Processing operation intention [17:38:33] [DEBUG] Requesting connection to Address(10.0.0.1:7777)
[17:38:33] [DEBUG] Waiting for connection on Address(10.0.0.1:52755) [17:38:33] [DEBUG] Connection request accepted
[17:38:33] [DEBUG] Operation is: UPLOAD [17:38:33] [DEBUG] Completing handshake with Address(10.0.0.1:7777)
[17:38:33] [DEBUG] Confirming operation [17:38:33] [DEBUG] Sending operation intention
[17:38:33] [DEBUG] Validating filename   [17:38:33] [DEBUG] Waiting for operation confirmation
[17:38:33] [WARN] [CONN:52755] Filename received invalid [17:38:33] [DEBUG] Operation accepted
[17:38:33] [ERROR] [CONN:52755] Client 10.0.1.1:41162 shutting down [17:38:33] [DEBUG] Connection established
due to file 'descarga.pdf' already existing in the server          [17:38:33] [DEBUG] Informing filename: descarga.pdf
[17:38:33] [DEBUG] Initiating connection close [17:38:33] [DEBUG] Waiting for filename confirmation
[17:38:34] [DEBUG] Received connection finalization from client [17:38:33] [DEBUG] Filename confirmation failed
[17:38:34] [INFO] [CONN:52755] Connection closed [17:38:33] [ERROR] File in server already exists
[17:38:34] [INFO] Stopping [17:38:33] [DEBUG] Connection finalization received. Confirming it
[17:38:34] [INFO] Press Enter to finish [17:38:33] [DEBUG] Sending own connection finalization

```

Figura 2: Ejemplo de upload de un archivo que ya existe en el servidor.

Para ambos protocolos, si el cliente intenta subir un archivo que el servidor ya tiene, se rechaza.

- Descarga de un archivo que no existe

```

"Node: h1"                               "Node: h2"
root@nacho:/home/nachotower/repositories/redes/TP1-Redes# ./src/ start-server.py -H 10.0.0.1 -v -s ./tmp/server/ -r saw           root@nacho:/home/nachotower/repositories/redes/TP1-Redes# ./sr c/download.py -H 10.0.0.1 -v -d ./tmp/client/no_existe.txt -n
[20:50:28] [INFO] Server started          no_existe.txt -r saw
[20:50:28] [DEBUG] Protocol: saw         [20:50:35] [DEBUG] Running on Address(0.0.0.0:37304)
[20:50:28] [DEBUG] Waiting for connection on Address(10.0.0.1:7777) [20:50:35] [DEBUG] Location to save downloaded file: ./tmp/client/no_existe.txt
[20:50:35] [DEBUG] Accepting connection for Address(10.0.0.1:37304) [20:50:35] [INFO] Client started for upload
[20:50:35] [DEBUG] Transferred to Address(10.0.0.1:44672)          [20:50:35] [DEBUG] Protocol: saw
[20:50:35] [DEBUG] Handshake completed    [20:50:35] [DEBUG] UDP socket ready
[20:50:35] [DEBUG] Processing operation intention [20:50:35] [DEBUG] Starting handshake
[20:50:35] [DEBUG] Operation is: DOWNLOAD [20:50:35] [DEBUG] Requesting connection to Address(10.0.0.1:7777)
[20:50:35] [DEBUG] Confirming operation [20:50:35] [DEBUG] Connection request accepted
[20:50:35] [DEBUG] Waiting for connection on Address(10.0.0.1:44672) [20:50:35] [DEBUG] Completing handshake with Address(10.0.0.1:7777)
[20:50:35] [DEBUG] Validating filename   [20:50:35] [DEBUG] Connection established
[20:50:35] [DEBUG] I/O error occurred: [Errno 2] No such file or [20:50:35] [DEBUG] Informing filename to download: no_existe.txt
directory: './tmp/server/no_existe.txt' [20:50:35] [DEBUG] Waiting for filename confirmation
[20:50:35] [WARN] [CONN:44672] Filename received invalid [20:50:35] [DEBUG] Operation accepted
[20:50:35] [ERROR] [CONN:44672] Client 10.0.1.1:37304 shutdowned [20:50:35] [DEBUG] Connection finalized received. Confirming it
due to file 'no_existe.txt' not existing in server for download [20:50:35] [DEBUG] Sending own connection finalization
[20:50:35] [DEBUG] Initiating connection close [20:50:35] [INFO] Connection closed
[20:50:35] [DEBUG] Received connection finalization from client [20:50:35] [DEBUG] File ./tmp/client/no_existe.txt is corrupted
[20:50:35] [INFO] [CONN:44672] Connection closed or incomplete. Removing file
[20:50:35] [INFO] Stopping [20:50:35] [INFO] Press Enter to finish
[20:50:35] [INFO] Connection closed
[20:50:35] [WARN] File ./tmp/client/no_existe.txt is corrupted
[20:50:35] [INFO] Stopping
[20:50:35] [INFO] Press Enter to finish

```

Figura 3: Ejemplo de intento de descarga de un archivo que no existe.

Para ambos protocolos, si el cliente intenta descargar un archivo que el servidor no posee, se rechaza.

4.2. Stop & Wait

Para mostrar el funcionamiento de Stop & Wait, mostraremos las capturas de las operaciones upload y download de un archivo pequeño de 5kB a modo de ejemplo. Primero sin pérdida de paquetes, y luego con una pérdida del 10% utilizando mininet.

4.2.1. Stop & Wait sin pérdida de paquetes

- Upload

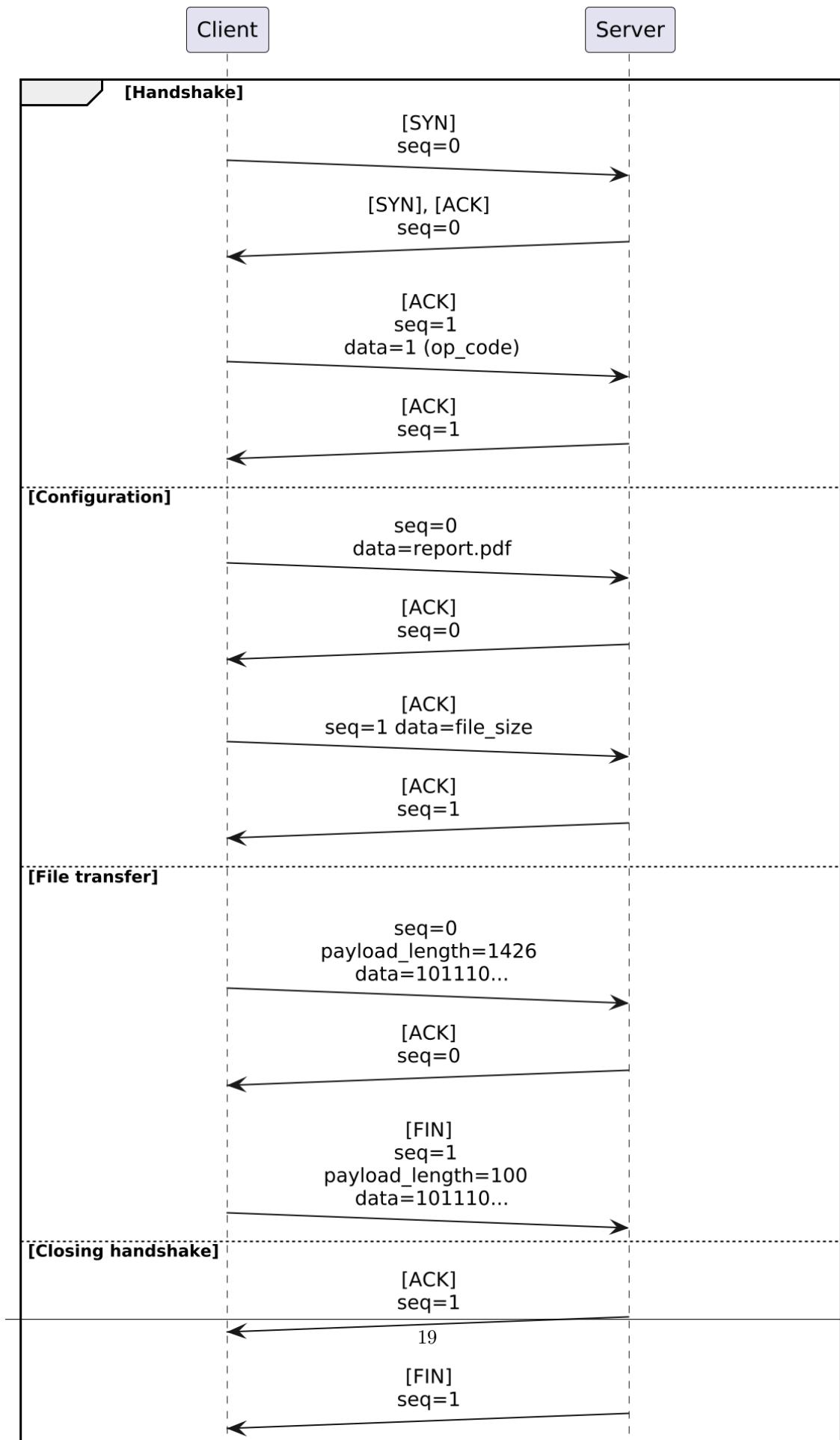


Figura 4: Captura de los logs de Upload con Stop and Wait.

No.	Time	Source	Destination	Protocol	Length	SQN	SAW	Info
3	0.001276884	10.0.1.1	10.0.0.1	SAW	48	0	(SAW)	[SYN]
4	0.001583021	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[SYN] [ACK]
5	0.004823484	10.0.1.1	10.0.0.1	SAW	50	1	(SAW)	[ACK] SEQ=1 Data(2)
6	0.006517231	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[ACK]
7	0.009580066	10.0.1.1	10.0.0.1	SAW	62	0	(SAW)	SEQ=0 Data(14)
8	0.010016107	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[ACK]
9	0.011770878	10.0.1.1	10.0.0.1	SAW	52	1	(SAW)	SEQ=1 Data(4)
10	0.012769219	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[ACK]
11	0.014334757	10.0.1.1	10.0.0.1	SAW	1474	0	(SAW)	SEQ=0 Data(1426)
12	0.015314610	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[ACK]
13	0.015678432	10.0.1.1	10.0.0.1	SAW	1474	1	(SAW)	SEQ=1 Data(1426)
14	0.016514919	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[ACK]
15	0.016722570	10.0.1.1	10.0.0.1	SAW	1474	0	(SAW)	SEQ=0 Data(1426)
16	0.017761222	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[ACK]
17	0.017989342	10.0.1.1	10.0.0.1	SAW	770	1	(SAW)	[FIN] SEQ=1 Data(722)
18	0.022569726	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[ACK]
19	0.022816041	10.0.1.1	10.0.0.1	SAW	48	0	(SAW)	[ACK]
20	0.023322291	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[FIN]

Figura 5: Captura de wireshark de Upload con Stop and Wait.

- Download

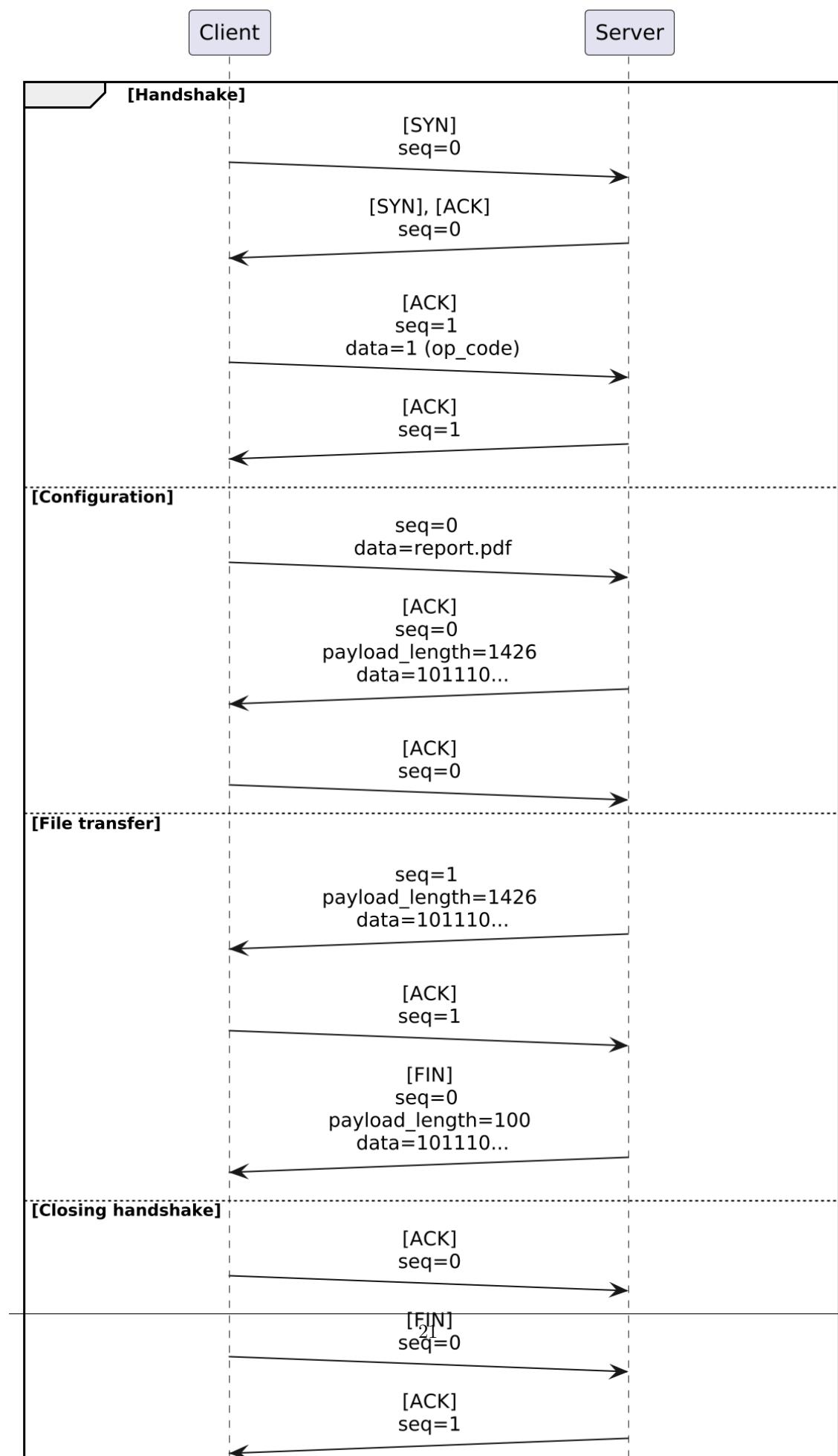


Figura 6: Captura de los logs de Download con Stop and Wait.

No.	Time	Source	Destination	Protocol	Length	SQN	SAW	Info
3	0.000889970	10.0.1.1	10.0.0.1	SAW	48	0	(SAW)	[SYN]
4	0.001228074	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[SYN] [ACK]
5	0.002993334	10.0.1.1	10.0.0.1	SAW	50	1	(SAW)	[ACK] SEQ=1 Data(2)
6	0.005225142	10.0.0.1	10.0.1.1	SAW	48	1	(SAW)	[ACK]
7	0.007731745	10.0.1.1	10.0.0.1	SAW	62	0	(SAW)	SEQ=0 Data(14)
8	0.008911627	10.0.0.1	10.0.1.1	SAW	1474	0	(SAW)	[ACK] SEQ=0 Data(1426)
9	0.010986390	10.0.1.1	10.0.0.1	SAW	48	0	(SAW)	[ACK]
10	0.011167391	10.0.0.1	10.0.1.1	SAW	1474	1	(SAW)	SEQ=1 Data(1426)
11	0.012880075	10.0.1.1	10.0.0.1	SAW	48	1	(SAW)	[ACK]
12	0.013667967	10.0.0.1	10.0.1.1	SAW	1474	0	(SAW)	SEQ=0 Data(1426)
13	0.014523036	10.0.1.1	10.0.0.1	SAW	48	0	(SAW)	[ACK]
14	0.016145495	10.0.0.1	10.0.1.1	SAW	770	1	(SAW)	[FIN] SEQ=1 Data(722)
15	0.017319266	10.0.1.1	10.0.0.1	SAW	48	1	(SAW)	[ACK]
16	0.017797019	10.0.0.1	10.0.1.1	SAW	48	0	(SAW)	[ACK]
17	0.020146182	10.0.1.1	10.0.0.1	SAW	48	1	(SAW)	[FIN]

Figura 7: Captura de wireshark de Download con Stop and Wait.

4.2.2. Stop & Wait con pérdida de paquetes del 10%

- Upload

```

stir                                     "Node:h1"
[20:39:31] [INFO] Server started
[20:39:31] [DEBUG] Protocol: saw
[20:39:31] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.1.7777)
[20:40:34] [DEBUG] [ACCEP] Accepting connection for Address(10.0.1.45696)
[20:40:34] [WARN] [ACCEP] Retransmission from re-listen attempt number 0
[20:40:34] [WARN] [ACCEP] Re-listening attempt attempt number 1.
Due to cause: The message is not an ACK
[20:40:34] [DEBUG] [ACCEP] Transferred to Address(10.0.0.1:48053)
[20:40:34] [DEBUG] [ACCEP] Handshake completed
[20:40:34] [DEBUG] [CONN:48053] Processing operation intention
[20:40:34] [DEBUG] [CONN:48053] Operation is: UPLOAD
[20:40:34] [DEBUG] [CONN:48053] Confirming operation
[20:40:34] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.1.7777)
[20:40:34] [DEBUG] [CONN:48053] Validating filename
[20:40:34] [DEBUG] [CONN:48053] Filename received valid: prueba.txt
[20:40:34] [DEBUG] [CONN:48053] Filesize received valid: 5001 bytes
[20:40:34] [DEBUG] [CONN:48053] Ready to receive from Address(10.0.1.45696)
[20:40:34] [DEBUG] [CONN:48053] Received chunk 1
[20:40:34] [DEBUG] [CONN:48053] Received chunk 2
[20:40:34] [DEBUG] [CONN:48053] Received chunk 3
[20:40:34] [DEBUG] [CONN:48053] Received chunk 4
[20:40:34] [DEBUG] [CONN:48053] Finished receiving file
[20:40:34] [DEBUG] [CONN:48053] Connection finalization received.
Confirming it
[20:40:34] [DEBUG] [CONN:48053] Sending own connection finalization
[20:40:34] [INFO] [CONN:48053] Connection closed
[20:40:34] [INFO] [CONN:48053] Upload completed from client 10.0.1.45696

```

```

root@nacho:/home/nachotower/repositories/redes/TP1-Redes# ./src/upload.py -H 10.0.0.1 -s tmp/client/prueba.txt -n prueba.txt -v -r saw
[20:40:34] [DEBUG] Running on Address(0.0.0.0:45696)
[20:40:34] [INFO] Client started for upload
[20:40:34] [DEBUG] Protocol: saw
[20:40:34] [DEBUG] UDP socket ready
[20:40:34] [DEBUG] Starting handshake
[20:40:34] [DEBUG] Requesting connection to Address(10.0.0.1:7777)
[20:40:34] [WARN] Retransmission from timeout attempt number 1
[20:40:34] [DEBUG] Connection request accepted
[20:40:34] [DEBUG] Completing handshake with Address(10.0.0.1:7777)
[20:40:34] [DEBUG] Sending operation intention
[20:40:34] [DEBUG] Waiting for operation confirmation
[20:40:34] [DEBUG] Operation accepted
[20:40:34] [DEBUG] Connection established
[20:40:34] [DEBUG] Informing filename: prueba.txt
[20:40:34] [DEBUG] Waiting for filename confirmation
[20:40:34] [DEBUG] Informing filesize: 5001 bytes
[20:40:34] [DEBUG] Waiting for filesize confirmation
[20:40:34] [INFO] Sending file prueba.txt of 0.00 MB
[20:40:34] [DEBUG] Sending chunk 1/4 of size 1.39 KB
[20:40:34] [DEBUG] Waiting confirmation for chunk 1/4
[20:40:34] [DEBUG] Sending chunk 2/4 of size 1.39 KB
[20:40:34] [DEBUG] Waiting confirmation for chunk 2/4
[20:40:34] [DEBUG] Sending chunk 3/4 of size 1.39 KB
[20:40:34] [DEBUG] Waiting confirmation for chunk 3/4
[20:40:34] [DEBUG] Sending chunk 4/4 of size 0.71 KB
[20:40:34] [DEBUG] Waiting confirmation for chunk 4/4
[20:40:34] [DEBUG] Waiting for confirmation of last packet
[20:40:34] [INFO] Upload completed
[20:40:34] [DEBUG] Received connection finalization from server
[20:40:34] [INFO] Stopping
[20:40:34] [INFO] Press Enter to finish

```

Figura 8: Captura de los logs de Upload con Stop and Wait con pérdida del 10 por ciento.

No.	Time	Source	Destination	Protocol	Length	Sequence Number	Info
L	24 25.155850803	fe80::7c...	ff02::fb	SAW	107	0 (SAW)	SEQ=0 Data(2)
L	20 0.020370512	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
L	19 0.020245871	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
L	18 0.020113175	10.0.1.1	10.0.0.1	SAW	771	1 (SAW)	[FIN] SEQ=1 Data(723)
L	17 0.019939198	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
L	16 0.019878521	10.0.1.1	10.0.0.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
L	15 0.019252480	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
L	14 0.019201061	10.0.1.1	10.0.0.1	SAW	1474	1 (SAW)	SEQ=1 Data(1426)
L	13 0.018416223	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
L	12 0.018379953	10.0.1.1	10.0.0.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
L	11 0.017579385	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
L	10 0.017522946	10.0.1.1	10.0.0.1	SAW	52	1 (SAW)	SEQ=1 Data(4)
L	9 0.0167577245	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
L	8 0.016646180	10.0.1.1	10.0.0.1	SAW	58	0 (SAW)	SEQ=0 Data(10)
L	7 0.015772511	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
L	6 0.015455347	10.0.1.1	10.0.0.1	SAW	50	1 (SAW)	[ACK] SEQ=1 Data(2)
L	5 0.014395087	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[SYN] [ACK]
L	4 0.014238364	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[SYN]
L	3 0.000403139	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[SYN]

Figura 9: Captura de wireshark de Upload con Stop and Wait con pérdida del 10 por ciento.

- #### ■ Download

```
"Node: h1"                                         "Node: h2"
[20:45:32] [INFO] Server started                  [20:45:06] [WARN] Retransmission from timeout attempt number 28
[20:45:32] [DEBUG] Protocol: saw                [20:45:06] [WARN] Retransmission from timeout attempt number 28
[20:45:32] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.0.1:7777) [root@nacho: /home/nachotower/repositories/redes/TP1-Redes# ./src/download.py -H 10.0.0.1 -v -d ./tmp/client/prueba.txt -n prueba.txt -r saw
[20:45:33] [DEBUG] [ACCEP] Accepting connection for Address(10.0.0.1:34641)
[20:45:33] [DEBUG] [ACCEP] Transferred to Address(10.0.0.1:54020) [20:45:33] [DEBUG] Running on Address(0.0.0.0:34641)
[20:45:33] [DEBUG] [ACCEP] Handshake completed [20:45:33] [DEBUG] Location to save downloaded file: ./tmp/client/prueba.txt
[20:45:33] [DEBUG] [CONN:54020] Processing operation intention [20:45:33] [INFO] Client started for upload
[20:45:33] [DEBUG] [CONN:54020] Operation is: DOWNLOAD [20:45:33] [DEBUG] Protocol: saw
[20:45:33] [DEBUG] [CONN:54020] Confirming operation [20:45:33] [DEBUG] UDP socket ready
[20:45:33] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.0.1:7777) [20:45:33] [DEBUG] Starting handshake
[20:45:33] [DEBUG] [CONN:54020] Validating filename [20:45:33] [DEBUG] Requesting connection to Address(10.0.0.1:7777)
[20:45:33] [WARN] [ACCEP] Retransmission from timeout attempt number 1 [20:45:33] [DEBUG] Connection request accepted
[20:45:33] [DEBUG] The message is not a SYN [20:45:33] [DEBUG] Completing handshake with Address(10.0.0.1:7777)
[20:45:33] [DEBUG] [ACCEP] Waiting for connection on Address(10.0.0.1:7777)
[20:45:33] [DEBUG] [CONN:54020] Filename received valid [20:45:33] [DEBUG] Sending operation intention
[20:45:33] [DEBUG] [CONN:54020] Ready to transmit to Address(10.0.0.1:34641) [20:45:33] [DEBUG] Waiting for operation confirmation
[20:45:33] [INFO] [CONN:54020] Sending file prueba.txt of 0.00 MB [20:45:33] [WARN] Retransmission from timeout attempt number 1
[20:45:33] [DEBUG] [CONN:54020] Sending chunk 1/4 of size 1.39 KB [20:45:33] [DEBUG] Operation accepted
[20:45:33] [DEBUG] [CONN:54020] Waiting confirmation for chunk 1 [20:45:33] [DEBUG] Connection established
[20:45:33] [DEBUG] [CONN:54020] Sending chunk 2/4 of size 1.39 KB [20:45:33] [DEBUG] Informing filename to download: prueba.txt
[20:45:33] [DEBUG] [CONN:54020] Waiting confirmation for chunk 2 [20:45:33] [DEBUG] Waiting for filename confirmation
[20:45:33] [DEBUG] [CONN:54020] Sending chunk 3/4 of size 1.39 KB [20:45:33] [DEBUG] Received chunk 1
[20:45:33] [DEBUG] [CONN:54020] Waiting confirmation for chunk 3 [20:45:33] [DEBUG] Received chunk 2
[20:45:33] [WARN] [ACCEP] Retransmission from timeout attempt number 1 [20:45:33] [DEBUG] Received chunk 3
[20:45:33] [DEBUG] [CONN:54020] Sending chunk 4/4 of size 0.71 KB [20:45:33] [DEBUG] Received chunk 4
[20:45:33] [DEBUG] [CONN:54020] Waiting for confirmation of last [20:45:33] [INFO] Download completed
packet [20:45:33] [INFO] [CONN:54020] File transfer complete [20:45:33] [DEBUG] Connection finalization received. Confirming it
[20:45:33] [DEBUG] [CONN:54020] Received connection finalization [20:45:33] [DEBUG] Sending own connection finalization
from server [20:45:33] [INFO] [CONN:54020] Download completed to client 10.0.0.1:34641 [20:45:33] [INFO] Connection closed
[20:45:33] [INFO] Stopping [20:45:33] [INFO] Press Enter to finish
```

Figura 10: Captura de los logs de Download con Stop and Wait con pérdida del 10 por ciento.

No.	Time	Source	Destination	Protocol	Length	Sequence Number	Info
1	0.000000000	10.0.0.1	10.0.0.1	SAW	48	0 (SAW)	[SYN]
2	0.000191429	10.0.0.1	10.0.0.1	SAW	48	0 (SAW)	[SYN] [ACK]
3	0.001184676	10.0.0.1	10.0.0.1	SAW	50	1 (SAW)	[ACK] SEQ=1 Data(2)
4	0.016704079	10.0.0.1	10.0.0.1	SAW	48	1 (SAW)	[ACK]
5	0.017188553	10.0.0.1	10.0.0.1	SAW	50	1 (SAW)	[ACK] SEQ=1 Data(2)
6	0.017878794	10.0.0.1	10.0.0.1	SAW	58	0 (SAW)	SEQ=0 Data(10)
7	0.018049912	10.0.0.1	10.0.0.1	SAW	1474	0 (SAW)	[ACK] SEQ=0 Data(1426)
8	0.018646994	10.0.0.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
9	0.018709220	10.0.0.1	10.0.0.1	SAW	1474	1 (SAW)	SEQ=1 Data(1426)
10	0.019245550	10.0.0.1	10.0.0.1	SAW	48	1 (SAW)	[ACK]
11	0.034486607	10.0.0.1	10.0.0.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
12	0.034897281	10.0.0.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
13	0.035028357	10.0.0.1	10.0.0.1	SAW	771	1 (SAW)	[FIN] SEQ=1 Data(723)
14	0.035193803	10.0.0.1	10.0.0.1	SAW	48	1 (SAW)	[ACK]
15	0.035240939	10.0.0.1	10.0.0.1	SAW	48	1 (SAW)	[FIN]
16	0.035312033	10.0.0.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
24	109.247021842	fe80::7c..	ff02::fb	SAW	107	0 (SAW)	SEQ=0 Data(2)

Figura 11: Captura de wireshark de Download con Stop and Wait con pérdida del 10 por ciento.

4.3. Go Back N

Para mostrar el funcionamiento de Go Back N, mostraremos las capturas de las operaciones upload y download del mismo archivo de 5kB. Para este caso, mostraremos únicamente el caso con pérdida de paquetes, ya que el escenario con pérdida resulta muy similar al de Stop & Wait debido a que la ventana que utilizamos es mayor a la cantidad de paquetes que posee este archivo.

- Upload

```

"Node:h1"                               "Node:h2"
[21:01:28] [WARN] [CONN:47917] Re-listening attempt attempt
    number 1. Due to cause: A packet with invalid sequence num
ber was received
[21:01:28] [DEBUG] [CONN:47917] Expected seq 3 got 2
[21:01:28] [WARN] [ACCEP] Retransmission from re-listen att
empt number 1
[21:01:28] [WARN] [CONN:47917] Re-listening attempt attempt
    number 2. Due to cause: A packet with invalid sequence num
ber was received
[21:01:28] [DEBUG] [CONN:47917] Filesize received valid: 50
01 bytes
[21:01:28] [DEBUG] [CONN:47917] Ready to receive from Addr
ess 10.0.1.1:37124
[21:01:28] [DEBUG] [CONN:47917] Beginning file reception in
    GBN manner
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Found invalid sequence numbe
r, expected seq 4
[21:01:28] [WARN] [CONN:47917] Received chunk 1.
[21:01:28] [DEBUG] [CONN:47917] Received chunk 2.
[21:01:28] [DEBUG] [CONN:47917] Received chunk 3.
[21:01:28] [DEBUG] [CONN:47917] Received chunk 4.
[21:01:28] [DEBUG] [CONN:47917] Finished receiving file
    received. Confirming it
[21:01:28] [DEBUG] [CONN:47917] Connection finalization rec
eived. Confirming it
[21:01:28] [INFO] [CONN:47917] Connection closed
[21:01:28] [INFO] [CONN:47917] Upload completed from client
    10.0.1.1:37124
r 2
[21:01:28] [DEBUG] Informing filesize: 5001 bytes
[21:01:28] [DEBUG] Waiting for filesize confirmation
[21:01:28] [DEBUG] Sending file 'prueba.txt' with window si
ze of 10 packets
[21:01:28] [DEBUG] Sending chunk 1/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 2/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 3/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 4/4 of size 0.74 KB.
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Retransmission is needed
[21:01:28] [DEBUG] Reseting next sequence number to base pa
cket number 1
[21:01:28] [DEBUG] Sending chunk 1/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 2/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 3/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 4/4 of size 0.74 KB.
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Retransmission is needed
[21:01:28] [DEBUG] Reseting next sequence number to base pa
cket number 1
[21:01:28] [DEBUG] Sending chunk 1/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 2/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 3/4 of size 1.38 KB.
[21:01:28] [DEBUG] Sending chunk 4/4 of size 0.74 KB.
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Received ack of packet 1
[21:01:28] [DEBUG] Waiting for confirmation of last packet
[21:01:28] [INFO] Upload completed
[21:01:28] [DEBUG] Received connection finalization from se
rver
[21:01:28] [INFO] Stopping
[21:01:28] [INFO] Press Enter to finish

```

Figura 12: Captura de los logs de Upload con Go Back N.

No.	Time	Source	Destination	Protocol	Length	Sequen	Info
1	0.0000000000	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[SYN] SEQ=0 ACK=0
2	0.000799898	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[SYN] SEQ=0 ACK=0
3	0.001491764	10.0.1.1	10.0.0.1	GBN	60	(GBN)	[ACK] SEQ=1 ACK=1 Data(2)
4	0.002193460	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=1 ACK=1
5	0.003152690	10.0.1.1	10.0.0.1	GBN	68	(GBN)	SEQ=2 ACK=2 Data(10)
6	0.019190671	10.0.1.1	10.0.0.1	GBN	68	(GBN)	SEQ=2 ACK=2 Data(10)
7	0.034261124	10.0.1.1	10.0.0.1	GBN	68	(GBN)	SEQ=2 ACK=2 Data(10)
8	0.034505123	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=2 ACK=2
9	0.034742097	10.0.1.1	10.0.0.1	GBN	62	(GBN)	SEQ=3 ACK=3 Data(4)
10	0.034893714	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=3 ACK=3
11	0.035107103	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=4 ACK=4 Data(1416)
12	0.035164185	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=5 ACK=4 Data(1416)
13	0.035243059	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=6 ACK=4 Data(1416)
14	0.035286915	10.0.1.1	10.0.0.1	GBN	811	(GBN)	[FIN] SEQ=7 ACK=4 Data(753)
15	0.035289410	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
16	0.035392141	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
17	0.035476597	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
18	0.045743825	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=4 ACK=4 Data(1416)
19	0.045800055	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=5 ACK=4 Data(1416)
20	0.045868299	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=6 ACK=4 Data(1416)
21	0.045917224	10.0.1.1	10.0.0.1	GBN	811	(GBN)	[FIN] SEQ=7 ACK=4 Data(753)
22	0.045949838	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
23	0.046035847	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
24	0.046106486	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
25	0.0563372050	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=4 ACK=4 Data(1416)
26	0.056439492	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=5 ACK=4 Data(1416)
27	0.056464140	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
28	0.056501433	10.0.1.1	10.0.0.1	GBN	1474	(GBN)	SEQ=6 ACK=4 Data(1416)

Figura 13: Captura de wireshark de Upload con Go Back N.

- Download

```
"Node: h1"                               "Node: h2"
.00 MB
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 1/4 of size 1
[21:05:34] [DEBUG] [CONN:39263] Waiting confirmation for ch
unk 1
[21:05:34] [DEBUG] [CONN:39263] Sending file 'prueba.txt' w
ith window size of 10 packets
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 2/4 of size 1
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 3/4 of size 1
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 4/4 of size 0
[21:05:34] [DEBUG] [CONN:39263] Detected previous ACK
[21:05:34] [DEBUG] [CONN:39263] Accumulated 0.00017380714416
503906 before retransmission needed. Totalling 0.0
[21:05:34] [WARN] [CONN:39263] Detected previous ACK
[21:05:34] [DEBUG] [CONN:39263] Accumulated 1.47819519042968
75e-05 before retransmission needed. Totalling 0.0001738071
4416503906
[21:05:34] [DEBUG] [CONN:39263] Retransmission is needed
[21:05:34] [DEBUG] [CONN:39263] Reseting next sequence numb
er to base packet number 2
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 2/4 of size 1
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 3/4 of size 1
[21:05:34] [DEBUG] [CONN:39263] Sending chunk 4/4 of size 0
[21:05:34] [DEBUG] [CONN:39263] Received ack of packet 2
[21:05:34] [DEBUG] [CONN:39263] Received ack of packet 3
[21:05:34] [DEBUG] [CONN:39263] Received ack of packet 4
[21:05:34] [DEBUG] [CONN:39263] Waiting for confirmation of
last packet
[21:05:34] [INFO] [CONN:39263] File transfer complete
[21:05:34] [DEBUG] [CONN:39263] Received connection finaliz
ation from client
[21:05:34] [INFO] [CONN:39263] Download completed to client
10.0.1.1:47581
[21:05:34] [DEBUG] [CONN:39263] Location to save downloaded file: ./tmp/
client/prueba.txt
[21:05:34] [INFO] Client started for upload
[21:05:34] [DEBUG] Protocol: gbn
[21:05:34] [DEBUG] UDP socket ready
[21:05:34] [DEBUG] Starting handshake
[21:05:34] [DEBUG] Requesting connection to Address(10.0.0.
1:7777)
[21:05:34] [DEBUG] Connection request accepted
[21:05:34] [DEBUG] Completing handshake with Address(10.0.
0.1:7777)
[21:05:34] [DEBUG] Sending operation intention
[21:05:34] [DEBUG] Waiting for operation confirmation
[21:05:34] [DEBUG] Operation accepted
[21:05:34] [DEBUG] Connection established
[21:05:34] [DEBUG] Informing filename to download: prueba.t
xt
[21:05:34] [DEBUG] Waiting for filename confirmation
[21:05:34] [DEBUG] Ready to receive from Address(10.0.0.1:3
9263)
[21:05:34] [DEBUG] Received chunk 1.
[21:05:34] [DEBUG] Beginning file reception in GBN manner
[21:05:34] [WARN] Found invalid sequence number, expected s
eq 3, got 5
[21:05:34] [DEBUG] Resending last ACK
[21:05:34] [WARN] Found invalid sequence number, expected s
eq 3, got 4
[21:05:34] [DEBUG] Resending last ACK
[21:05:34] [DEBUG] Received chunk 2.
[21:05:34] [DEBUG] Received chunk 3.
[21:05:34] [DEBUG] Received chunk 4.
[21:05:34] [INFO] Download completed
[21:05:34] [DEBUG] Connection finalization received. Confir
ming it
[21:05:34] [DEBUG] Sending own connection finalization
[21:05:34] [INFO] Connection closed
[21:05:34] [INFO] Stopping
[21:05:34] [INFO] Press Enter to finish
```

Figura 14: Captura de los logs de Download con Go Back N.

No.	Time	Source	Destination	Protocol	Length	Sequen	Info
1	0.000000000	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[SYN] SEQ=0 ACK=0
2	0.000746210	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[SYN] SEQ=0 ACK=0
3	0.001922470	10.0.1.1	10.0.0.1	GBN	60	(GBN)	[ACK] SEQ=1 ACK=1 Data(2)
4	0.002891274	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=1 ACK=1
5	0.004513416	10.0.1.1	10.0.0.1	GBN	68	(GBN)	SEQ=2 ACK=2 Data(10)
6	0.005034185	10.0.0.1	10.0.1.1	GBN	1474	(GBN)	[ACK] SEQ=2 ACK=2 Data(1416)
7	0.005571916	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=2 ACK=2
8	0.005977449	10.0.0.1	10.0.1.1	GBN	1474	(GBN)	SEQ=3 ACK=3 Data(1416)
9	0.005979183	10.0.0.1	10.0.1.1	GBN	1474	(GBN)	SEQ=4 ACK=3 Data(1416)
10	0.006017848	10.0.0.1	10.0.1.1	GBN	811	(GBN)	[FIN] SEQ=5 ACK=3 Data(753)
11	0.006124908	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=2 ACK=2
12	0.006156359	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=2 ACK=2
13	0.016495767	10.0.0.1	10.0.1.1	GBN	1474	(GBN)	SEQ=3 ACK=3 Data(1416)
14	0.016554853	10.0.0.1	10.0.1.1	GBN	1474	(GBN)	SEQ=4 ACK=3 Data(1416)
15	0.016599660	10.0.0.1	10.0.1.1	GBN	811	(GBN)	[FIN] SEQ=5 ACK=3 Data(753)
16	0.016607967	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=3 ACK=3
17	0.016675308	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=4 ACK=4
18	0.016803128	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[ACK] SEQ=5 ACK=5
19	0.016832706	10.0.1.1	10.0.0.1	GBN	58	(GBN)	[FIN] SEQ=5 ACK=5
20	0.016933042	10.0.0.1	10.0.1.1	GBN	58	(GBN)	[ACK] SEQ=6 ACK=5
21	0.0174104740	00:00:00	00:00:00	ARP	42	00:00:00	00:00:00:00:00:00

Figura 15: Captura de wireshark de Download con Go Back N.

4.4. Pruebas automatizadas

Si bien se detallaron ejemplos manuales donde se probaron las características del protocolo de aplicación desarrollado, también se desarrollaron pruebas automatizadas con *pytest*, siendo ventajoso el hecho de que Mininet está desarrollo en Python y ofrece una API.

Se desarrollaron pruebas de integración que testean:

- Upload y download correcto.
- Caso download cuando la ubicación de guardado del archivo a descargar ya existe.
- Caso download cuando el archivo a descargar no existe en el servidor.
- Caso upload cuando la ubicación del archivo a subir no existe.
- Caso download cuando el archivo a subir ya existe en el servidor con el nombre proveído.
- Caso de comunicación fallida por protocol mismatch entre servidor y clientes.
- Caso de descargas y subida múltiples y simultáneas.

Todas estas pruebas se ejecutan, cada una, bajo las siguientes condiciones:

- Protocolo SAW con 0 % de packet loss.
- Protocolo SAW con 10 % de packet loss.
- Protocolo SAW con 40 % de packet loss.
- Protocolo GBN con 0 % de packet loss.
- Protocolo GBN con 10 % de packet loss.
- Protocolo GBN con 40 % de packet loss.

Estas pruebas generan un alto consumo de CPU y poseen una cota de tiempo (establecida para que en el CI/CD no consuma recursos de más) por lo que requieren de hardware adecuado.

5. Preguntas a Responder

5.1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es una de las arquitecturas más comunes. En esta arquitectura hay un *host (end system)* llamado *server* que está siempre encendido y pasivamente escucha *requests* de otros *hosts* llamados *clients* que son agentes activos que inician la comunicación con el *server*.

Un ejemplo de esta arquitectura es una aplicación Web donde hay un *Web server* que escucha *requests* de navegadores web. El navegador web es el cliente que inicia la comunicación y el *Web server* es el servidor que responde a los *requests*. Estos mensajes tienen el formato de Capa de Aplicación HTTP.

5.1.1. Características

- Los clientes son agentes activos que inician la comunicación.
- Los clientes no se comunican entre sí.
- Los clientes no necesitan estar encendidos todo el tiempo ni tener una IP fija.
- Los servidores son pasivos y siempre están encendidos.
- Los servidores **deben** tener una IP fija bien conocida (*well-known IP address*) que se puede resolver con un nombre de dominio DNS (*domain name*).
- Los servidores pueden tener múltiples clientes conectados al mismo tiempo.

5.1.2. Ventajas

- Diseño simple usando protocolos sin estado como HTTP donde el servidor no necesita mantener información sobre clientes ya que se puede guardar información del cliente en *cookies* del cliente y estos se transmiten en *headers* HTTP.
- Puede soportar un gran número de clientes.

5.1.3. Desventajas

- Un solo punto de falla. Si el servidor se cae, el servicio se cae.
- El servidor debe estar encendido todo el tiempo.
- Gran costo para escalar, ya que a medida de que el servicio tiene más usuarios, el servidor debe también aumentar su capacidad de procesar más clientes.

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

Un protocolo de capa de aplicación especifica cómo los procesos de una aplicación, que se ejecutan en diferentes sistemas finales, intercambian mensajes entre sí. Este tipo de protocolo define:

- Los tipos de mensajes que se envían, como mensajes de solicitud y de respuesta.
- La sintaxis de los mensajes, es decir, la estructura de los campos dentro de cada mensaje y cómo se separan o identifican esos campos.
- La semántica de los campos, indicando qué significa la información contenida en cada uno.
- Las reglas de comunicación, que establecen cuándo un proceso debe enviar un mensaje y cómo debe reaccionar al recibir uno.

En resumen, los protocolos de capa de aplicación aseguran que las aplicaciones puedan comunicarse correctamente y coordinarse en la red, haciendo posible servicios como el correo electrónico, la web o la transferencia de archivos.

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

Este apartado fue detallado y explicado en la sección **Implementación**.

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

5.4.1. ¿Qué servicios proveen dichos protocolos?

Ambos protocolos proveen los siguientes servicios:

- **Multiplexación/Demultiplexación:** son los mecanismos que permiten extender el servicio de entrega de IP entre dos end systems a un servicio de entrega entre dos procesos que se ejecutan en esos sistemas. Dichos mecanismos permiten identificar a qué proceso pertenece cada segmento recibido.
- **Chequeo de integridad:** se verifica que no haya errores en los datos mediante un campo de checksum en los headers de ambos protocolos.

UDP no realiza ninguna otra función extra. Por lo tanto, su servicio es:

- **No confiable:** no garantiza que la entrega de los paquetes sea exitosa, ni tampoco que lleguen en orden.
- **Sin conexión:** cada paquete datagrama se envía de manera independiente, sin garantías de que el receptor esté listo o incluso disponible.

Por su parte, TCP ofrece las siguientes funcionalidades adicionales:

- **Orientado a la conexión:** antes de que un proceso de aplicación pueda comenzar a enviar datos a otro, ambos procesos deben comunicarse entre sí; es decir, deben enviarse algunos segmentos preliminares para establecer los parámetros de la transferencia de datos subsiguiente. Se trata de una conexión lógica con un estado en común que reside en TCP de los hosts.
- **Transferencia de datos confiable:** garantiza la entrega, el orden y la no corrupción de los datos. Esto lo logra mediante timers, números de secuencia y ACKs (flags que indican que un paquete fue entregado correctamente).
- **Control de congestión:** gestiona que no se saturen los enlaces. Es más bien un servicio para la red.
- **Control de flujo:** para eliminar la posibilidad de que el remitente desborde el buffer del receptor. Hace coincidir la velocidad a la que el remitente envía con la velocidad a la que la aplicación receptora lee.

5.4.2. ¿Cuáles son sus características?

Algunas de las características de UDP son las siguientes:

- **Pequeño overhead de header por paquete:** UDP posee un header pequeño (8 bytes) en comparación con TCP (20 bytes)

- **Sin estado de conexión:** UDP no mantiene un estado de conexión en los end systems, por lo que no rastrea ningún parámetro. Por esta razón, un servidor dedicado a una aplicación específica generalmente puede admitir muchos más clientes activos cuando la aplicación se ejecuta mediante UDP en lugar de TCP.
- **Sin retraso por conexión:** UDP no induce ningún retraso para establecer una conexión, a diferencia de TCP que posee un handshake de tres pasos.

Por su parte, TCP posee las siguientes características:

- **Full-duplex:** dada una conexión TCP entre dos hosts, digamos A y B, la información puede fluir de A a B al mismo tiempo que fluye información de B a A.
- **Conexión point-to-point:** la conexión de TCP únicamente se puede establecer entre un único remitente y un único receptor, no admite multicasting.
- **Three-Way Handshake:** para establecer la conexión mencionada anteriormente se realiza un procedimiento donde se envían tres segmentos.

5.4.3. ¿Cuándo es apropiado utilizar cada uno?

Ninguno de estos protocolos es mejor que el otro. Para decidir cuál de ellos utilizar, se deben tener en cuenta las necesidades de la aplicación. Debido a las características mencionadas anteriormente, UDP resulta más apropiado para aplicaciones que requieran mayor velocidad sin que sea tan sensible a algunas pérdidas de paquetes, por ejemplo plataformas de streaming, y si se tiene un servidor dedicado a una aplicación específica que necesita poder admitir muchos más clientes activos. Por otro lado, TCP es más ventajoso para las aplicaciones que necesitan un transporte confiable de los datos. Algunos ejemplos son el email y la web.

6. Anexo: Fragmentación IPv4

6.1. Consideraciones iniciales

Se propuso inicialmente una red de topología lineal que una a:

- Un host servidor
- Tres switches conectados en serie
- Un host cliente

Donde haya pérdida de paquetes en el enlace próximo al host receptor y que el switch central tenga la capacidad de fragmentar paquetes que excedan el MTU de alguna de sus interfaces.

Sin embargo, los switches en mininet no tienen la capacidad de fragmentar paquetes, por lo que forzar este fenómeno causaría una pérdida total de todo paquete de tamaño mayor al MTU. Conceptualmente un switch, al ser un elemento de Link Layer, no debe tener la capacidad de fragmentar paquetes de otra capa de red. La solución propuesta fue sustituir ese switch central por un routercentral, un tipo de nodo de mininet similar a los hosts pero que además cuenta con la capacidad de hacer forwarding de paquetes IPv4.

Otro de los conflictos presentados es la comunicación bidireccional. Reducir el MTU de alguna interfaz de un router implica que se reduce sobre el enlace de esta interfaz. Y si este enlace está conectado a un router y a un switch, la comunicación andará bien en sentido router → switch pero se perderán los paquetes en el sentido switch → router.

Por lo tanto, si se desea una tráfico bidireccional de paquetes excedentes al MTU de alguna de las interfaces de la ruta esta NO debe estar vinculada a un switch, de lo contrario se perderá todo paquete que excede el MTU reducido. Dicho esto, se decidió arbitrariamente que basta con tener comunicación unidireccional (C → S) para poder provocar el fenómeno de fragmentación de paquetes.

6.2. Análisis

6.2.1. Topología

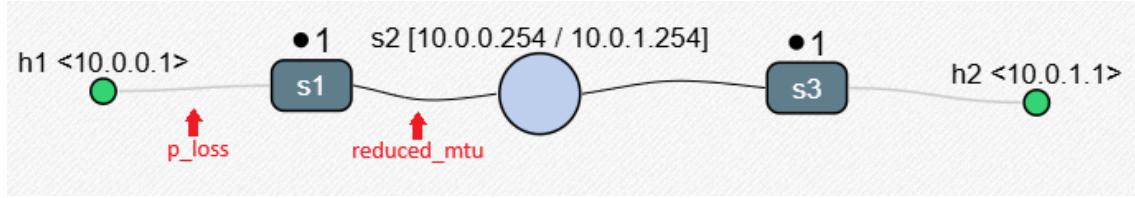


Figura 16: Red con un servidor, dos switches, un router y un cliente

Se define una pérdida de 20 % de paquetes sobre el enlace próximo al servidor, un MTU de 800 en la interfaz saliente del router (s2-eth0) y un tamaño de paquetes de 1000B siendo ambos valores elegidos arbitrariamente, basta con generar tráfico de paquetes de tamaño superior al MTU reducido para estudiar este fenómeno.

6.2.2. Proceso de fragmentación

Habiendo definido la red como se mencionó anteriormente, se genera tráfico de cliente a servidor usando iperf y se capturan los paquetes recibidos y enviados por las interfaces del router (s2) con Wireshark para estudiar este fenómeno. Por ejemplo:

No.	Time	Source	Destination	Protocol	Length	Info
15	1.447252	10.0.1.1	10.0.0.1	TCP	74	51126 - 5001 [SYN] Seq=0 Win=43000 Len=0 MSS=1000 SACK_PERM TSval=3000069702 TSecr=0 WS=512
16	1.454771	10.0.0.1	10.0.1.1	TCP	74	5001 - 51126 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2148938881 TSecr=3000069702 WS=512
17	1.455599	10.0.1.1	10.0.0.1	TCP	66	51126 - 5001 [ACK] Seq=1 Ack=1 Win=43008 Len=0 TSval=3000069717 TSecr=2148938881
18	1.455594	10.0.1.1	10.0.0.1	TCP	126	51126 - 5001 [PSH, ACK] Seq=1 Ack=1 Win=43008 Len=60 TSval=3000069717 TSecr=2148938881
19	1.455598	10.0.1.1	10.0.0.1	TCP	2042	51126 - 5001 [PSH, ACK] Seq=51 Ack=1 Win=43008 Len=1976 TSval=3000069717 TSecr=2148938881
20	1.456842	10.0.1.1	10.0.0.1	TCP	2042	51126 - 5001 [PSH, ACK] Seq=2037 Ack=1 Win=43088 Len=1976 TSval=3000069717 TSecr=2148938881
21	1.456866	10.0.1.1	10.0.0.1	TCP	2042	51126 - 5001 [PSH, ACK] Seq=4013 Ack=1 Win=43088 Len=1976 TSval=3000069717 TSecr=2148938881
22	1.456989	10.0.1.1	10.0.0.1	TCP	2042	51126 - 5001 [PSH, ACK] Seq=5989 Ack=1 Win=43088 Len=1976 TSval=3000069717 TSecr=2148938881
23	1.456111	10.0.1.1	10.0.0.1	TCP	1054	51126 - 5001 [ACK] Seq=7985 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
24	1.458500	10.0.0.1	10.0.1.1	TCP	66	5001 - 51126 [ACK] Seq=61 Ack=61 Win=0 TSval=2148938884 TSecr=3000069717

Figura 17: Paquetes capturados en interfaz de entrada del router (s2-eth1)

No.	Time	Source	Destination	Protocol	Length	Info
17	1.772126	10.0.1.1	10.0.0.1	TCP	74	51126 - 5001 [SYN] Seq=0 Win=43000 Len=0 MSS=1000 SACK_PERM TSval=3000069702 TSecr=0 WS=512
18	1.775532	10.0.0.1	10.0.1.1	TCP	74	5001 - 51126 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2148938881 TSecr=3000069702 WS=512
19	1.776303	10.0.1.1	10.0.0.1	TCP	66	51126 - 5001 [ACK] Seq=1 Ack=1 Win=43008 Len=0 TSval=3000069717 TSecr=2148938881
20	1.776379	10.0.1.1	10.0.0.1	TCP	126	51126 - 5001 [PSH, ACK] Seq=1 Ack=1 Win=43008 Len=60 TSval=3000069717 TSecr=2148938881
21	1.776787	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1706) [Reassembled in #22]
22	1.776788	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [ACK] Seq=61 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
23	1.776791	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1707) [Reassembled in #24]
24	1.776792	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [PSH, ACK] Seq=1044 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
25	1.776822	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1708) [Reassembled in #26]
26	1.776827	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [ACK] Seq=2037 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
27	1.776822	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1709) [Reassembled in #28]
28	1.776830	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [PSH, ACK] Seq=3025 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
29	1.776849	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1708) [Reassembled in #30]
30	1.776850	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [ACK] Seq=4913 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
31	1.776852	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1706) [Reassembled in #32]
32	1.776853	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [PSH, ACK] Seq=5001 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881
33	1.776872	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1706) [Reassembled in #34]
34	1.776874	10.0.1.1	10.0.0.1	TCP	278	51126 - 5001 [ACK] Seq=5989 Ack=1 Win=43008 Len=988 TSval=3000069717 TSecr=2148938881

Figura 18: Paquetes capturados en interfaz de salida del router (s2-eth0)

Una vez establecida la conexión TCP, los paquetes que superan el MTU de la interfaz de salida son fragmentados por el router. Cada fragmento se encapsula en su propio datagrama IPv4 (con el mismo Identification y flags MF/Offset adecuados) y se envía por la red.

No.	Time	Source	Destination	Protocol	Length	Info
22	3.906561	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol
• 23	3.906564	10.0.1.1	10.0.0.1	TCP	278	35848 → 5001 [PSH, ACK]

▶ Frame 22: 810 bytes on wire (6480 bits), 810 bytes captured (6480 bits)
 ▶ Ethernet II, Src: b6:63:93:6e:0e:ef (b6:63:93:6e:0e:ef), Dst: f6:50:85:00:ad:58 (f6:50:85:00:ad:58)
 ▶ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.0.1
 0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
 ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
 Total Length: 796
 Identification: 0x529e (21150)
 ▶ 001. = Flags: 0x1, More fragments
 0.... = Reserved bit: Not set
 .0. = Don't fragment: Not set
 ..1. = More fragments: Set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 63
 Protocol: TCP (6)
 Header Checksum: 0xf13c [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 10.0.1.1
 Destination Address: 10.0.0.1
 [Reassembled IPv4 in frame: 23]

Figura 19: Secuencia de un paquete fragmentado (1) a detalle

▶ Frame 23: 278 bytes on wire (2224 bits), 278 bytes captured (2224 bits)
▶ Ethernet II, Src: b6:63:93:6e:0e:ef (b6:63:93:6e:0e:ef), Dst: f6:50:85:00:ad:58 (f6:50:85:00:ad:58)
▶ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.0.1 0100 = Version: 4 0101 = Header Length: 20 bytes (5) ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT) Total Length: 264 Identification: 0x529e (21150) ▶ 000. = Flags: 0x0 0.... = Reserved bit: Not set .0. = Don't fragment: Not set ..0. = More fragments: Not set ...0 0000 0110 0001 = Fragment Offset: 776 Time to Live: 63 Protocol: TCP (6) Header Checksum: 0x12f0 [validation disabled] [Header checksum status: Unverified] Source Address: 10.0.1.1 Destination Address: 10.0.0.1 ▶ [2 IPv4 Fragments (1020 bytes): #22(776), #23(244)] [Frame: 22, payload: 0-775 (776 bytes)] [Frame: 23, payload: 776-1019 (244 bytes)] [Fragment count: 2] [Reassembled IPv4 length: 1020] [Reassembled IPv4 data [truncated]: 8c0813893d8882d4be5838668018005480e200000101080afdefda921e]
▶ Transmission Control Protocol, Src Port: 35848, Dst Port: 5001, Seq: 1049, Ack: 1, Len: 988

Figura 20: Secuencia de un paquete fragmentado (2) a detalle

Wireshark agrupa los fragmentos bajo un mismo “Reassembled in #” basándose en el Identification y los offsets; tras inspeccionar la columna de secuencia TCP, se puede mapear fácilmente cada fragmento al segmento original. Si un segmento requiere más de dos fragmentos, se observará varios “Reassembled in #” con offsets crecientes. Sin embargo, cabe destacar que el reensamblado ocurre únicamente en el host destino, no en el router. Esto debido a que si cada router tuviese que reensamblar paquetes sería un proceso muy costoso y afectaría al rendimiento de la red.

Sumando las longitudes de los fragmentos observamos que el router termina transmitiendo más datos que los recibidos originalmente: esto se debe a que cada fragmento es un paquete en sí mismo, y por cada fragmento generado es un nuevo header.

6.2.3. Funcionamiento de TCP ante la pérdida de un fragmento

TCP es un protocolo de transporte que maneja la retransmisión de datos tras pérdida y detección a partir de ACKs duplicados. Este fenómeno es muy simple de visualizar en Wireshark, donde se señala en la columna de información cuando un paquete corresponde a información

retransmitida.

Cuando un paquete TCP es demasiado grande para pasar por una interfaz con un MTU pequeño y no tiene activado el bit "Don't Fragment", el router lo fragmenta a nivel IP. Esto significa que el paquete IP original se divide en varios fragmentos IP, cada uno con su propio encabezado IP, pero todos parte del mismo datagrama original.

Los fragmentos no se reensamblan en los routers intermedios sino que el host de destino es quien reensambla todos los fragmentos en el paquete IP original antes de entregárselo al stack TCP.

No.	Time	Source	Destination	Protocol	Length	Info
75	1.463044	10.0.1.1	10.0.0.1	TCP	1094	51126 → 5001 [ACK] Seq=34644 Ack=1 Win=43098 Len=988 TSeqr=3009069724 TSeq=2148938881
76	1.463080	10.0.0.1	10.0.1.1	TCP	86	[TCP Dup ACK 69x1] 5001 → 51126 [ACK] Seq=29 Ack=23773 Win=23592 Len=0 TSeq=2148938891 TSeqr=3009069724 SLE=33653 SRE=34641 SLE=24761 SRE=28713
77	1.463105	10.0.1.1	10.0.0.1	TCP	1054	[TCP Fast Retransmission] 51126 → 5001 [ACK] Seq=23773 Ack=1 Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881
78	1.463119	10.0.1.1	10.0.0.1	TCP	1054	[TCP Fast Retransmission] 51126 → 5001 [PSH, ACK] Seq=28713 Ack=Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881
79	1.463133	10.0.1.1	10.0.0.1	TCP	78	5001 → 51126 [ACK] Seq=28 TSeq=2148938891 TSeqr=3009069724 SLE=297081 SRE=34641
80	1.463149	10.0.1.1	10.0.0.1	TCP	1054	[TCP Fast Retransmission] 51126 → 5001 [ACK] Seq=28713 Ack=Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881
81	1.463188	10.0.1.1	10.0.0.1	TCP	3939	51126 → 5001 [PSH, ACK] Seq=35629 Ack=1 Win=43098 Len=2964 TSeq=2148938891 TSeqr=3009069724
82	1.463252	10.0.0.1	10.0.1.1	TCP	86	[TCP Dup ACK 79x1] 5001 → 51126 [ACK] Seq=29 Ack=28713 Win=10944 Len=0 TSeq=2148938891 TSeqr=3009069724 SLE=35629 SRE=36617 SLE=29701 SRE=34641
83	1.463355	10.0.0.1	10.0.1.1	TCP	86	[TCP Dup ACK 79x2] 5001 → 51126 [ACK] Seq=29 Ack=28713 Win=10944 Len=0 TSeq=2148938891 TSeqr=3009069724 SLE=35629 SRE=37065 SLE=29701 SRE=34641
84	1.463380	10.0.1.1	10.0.0.1	TCP	1054	[TCP Fast Retransmission] 51126 → 5001 [PSH, ACK] Seq=28713 Ack=1 Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881
85	1.463412	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=36593 Ack=1 Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881
86	1.463442	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=39581 Ack=1 Win=43098 Len=988 TSeq=3009069724 TSeqr=2148938881

Figura 21: Retransmisión TCP vista en el router

Es decir, si se pierde un solo fragmento, el paquete IP no puede ser reensamblado, y por lo tanto no llega al protocolo TCP. El sistema operativo desecha todos los fragmentos recibidos de ese datagrama después de un tiempo de espera (timeout).

Desde el punto de vista de TCP nunca ve los fragmentos individuales. Solo sabe que no recibió un segmento TCP completo porque fue descartado en capa IP.

De esta forma, TCP en el servidor no responde porque no tiene nada que confirmar y TCP en el cliente eventualmente detecta la pérdida (por timeout o falta de ACK) y retransmite el segmento TCP completo.

Capturando el output del comando iperf (ejecutado durante 3 segundos) se puede observar el resultado final de la comunicación en cliente y servidor:

```

1 -----
2 Client connecting to 10.0.0.1, TCP port 5001
3 MSS req size 1000 bytes (per TCP_MAXSEG)
4 TCP window size: 85.3 KByte (default)
5 -----
6 [ 1] local 10.0.1.1 port 51126 connected with 10.0.0.1 port 5001 (icwnd/mss/intt=9/988/13827)
7 [ ID] Interval      Transfer     Bandwidth
8 [ 1] 0.0000-1.0000 sec   237 KBytes   1.94 Mbits/sec
9 [ 1] 1.0000-2.0000 sec   0.000 Bytes   0.000 bits/sec
10 [ 1] 2.0000-3.0000 sec   0.000 Bytes   0.000 bits/sec
11 [ 1] 0.0000-5.4206 sec   237 KBytes   359 Kbytes/sec

```

Figura 22: Captura de iperf TCP desde cliente

```

Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)

[ 1] local 10.0.0.1 port 5001 connected with 10.0.1.1 port 51126 (icwnd/mss/intt=9/988/3107)
[ ID] Interval      Transfer     Bandwidth
[ 1] 0.0000-5.4168 sec   237 KBytes   359 Kbits/sec

```

Figura 23: Captura de iperf TCP desde servidor

6.2.4. Funcionamiento de UDP ante la pérdida de un fragmento

UDP no es un protocolo orientado a la transferencia confiable de datos, por lo que NO cuenta con ningún mecanismo tras pérdida de paquetes, solamente envía sin garantía de recepción.

En estas capturas de Wireshark se evidencia como ocurre el fenómeno de fragmentación, donde al solo fragmentar en 2 un paquete UDP en Wireshark se capturará cerca del doble de paquetes

en cada interfaz (o más según la relación tamaño/MTU):

No.	Time	Source	Destination	Protocol	Length	Info
	2 1.684906	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	3 1.687054	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	4 1.687099	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	5 1.687119	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	6 1.687396	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	7 1.687482	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	8 1.688443	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	9 1.689349	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
	10 1.689726	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000

Figura 24: Paquetes capturados en interfaz de entrada del router (s2-eth1)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=f0f9) [Reassembled in #2]
2	0.000007	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
3	0.002141	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=f0fa) [Reassembled in #4]
4	0.002145	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
5	0.002165	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=f0fb) [Reassembled in #6]
6	0.002167	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
7	0.002185	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=f0fc) [Reassembled in #8]
8	0.002187	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
9	0.002474	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=f0fd) [Reassembled in #10]
10	0.002477	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000

Figura 25: Paquetes capturados en interfaz de salida del router (s2-eth0)

Sin embargo, al haber pérdida de paquetes cercano al servidor, esta no puede observarse capturando sobre el router sino sobre el servidor, donde se observaría que tantos paquetes se pierden a partir de que tantos envió el cliente. Otra forma más sencilla de observar este fenómeno es con el output del comando iperf ejecutado sobre UDP, donde explícitamente señala que cantidad de paquetes se perdieron.

```

Client connecting to 10.0.0.1, UDP port 5001
Sending 1000 byte datagrams, IPG target: 800.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)

[ 1] local 10.0.1.1 port 56856 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer     Bandwidth
[ 1] 0.0000-1.0000 sec  1.19 MBytes  10.0 Mbits/sec
[ 1] 1.0000-2.0000 sec  1.19 MBytes  9.97 Mbits/sec
[ 1] 2.0000-3.0000 sec  1.20 MBytes  10.0 Mbits/sec
[ 1] 0.0000-3.0011 sec  3.58 MBytes  10.0 Mbits/sec
[ 1] Sent 3754 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer     Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-2.9970 sec  2.92 MBytes  8.18 Mbits/sec   0.062 ms 688/3753 (18%)
[ 1] 0.0000-2.9970 sec  3 datagrams received out-of-order

```

Figura 26: Captura de iperf UDP desde cliente

```

1 -----
2 Server listening on UDP port 5001
3 UDP buffer size: 208 KByte (default)
4 -----
5 [ 1] local 10.0.0.1 port 5001 connected with 10.0.1.1 port 56856
6 [ ID] Interval      Transfer     Bandwidth      Jitter    Lost/Total Datagrams
7 [ 1] 0.0000-2.9970 sec  2.92 MBytes  8.18 Mbits/sec  0.062 ms 688/3753 (18%)
8 [ 1] 0.0000-2.9970 sec  3 datagrams received out-of-order

```

Figura 27: Captura de iperf UDP desde servidor
Donde se observa una pérdida en torno al 20 % definido inicialmente.

6.2.5. Aumento de tráfico al reducirse el MTU mínimo de la red.

Por último, el fenómeno más sencillo de comprobar. Como se mencionó anteriormente, ya que el router fragmenta paquetes de ambos protocolos de transporte en función de su tamaño y el MTU de la interfaz de salida, siempre va a enviar más paquetes de los que recibe, por lo que se observa un aumento notorio del tráfico en la red. Para esta experiencia, se observa un aumento trabajando con UDP:

No.	Time	Source	Destination	Protocol	Length	Info
3752	4.682403	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
3753	4.683427	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
3754	4.684459	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
3755	4.685006	10.0.1.1	10.0.0.1	UDP	1042	56856 → 5001 Len=1000
3756	4.704161	10.0.0.1	10.0.1.1	UDP	170	5001 → 56856 Len=128

Figura 28: Paquetes capturados en interfaz de entrada del router (s2-eth1)

No.	Time	Source	Destination	Protocol	Length	Info
7503	2.999542	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=ffa0) [Reassembled in #7504]
7504	2.999547	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
7505	3.000083	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=UDP 17, off=0, ID=ffa1) [Reassembled in #7506]
7506	3.000086	10.0.1.1	10.0.0.1	UDP	266	56856 → 5001 Len=1000
7507	3.019206	10.0.0.1	10.0.1.1	UDP	170	5001 → 56856 Len=128

Figura 29: Paquetes capturados en interfaz de salida del router (s2-eth0)

Y se observa también un aumento en el tráfico de la red trabajando con TCP:

No.	Time	Source	Destination	Protocol	Length	Info
513	6.864242	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=239157 Ack=29 Win=43908 Tsvl=3000075125 TSscr=2148944292
514	6.866287	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=240145 Ack=29 Win=43908 Tsvl=3000075125 TSscr=2148944292
515	6.868332	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=241130 Ack=29 Win=43908 Tsvl=2148944296 Tscr=241130 SLE=241133
516	6.868357	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=241130 Ack=29 Win=43908 Tsvl=2148944296 Tscr=241130 SLE=241133
517	6.868362	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=241130 Ack=29 Win=43908 Tsvl=3000075129 TSscr=2148944296
518	6.868384	10.0.1.1	10.0.0.1	TCP	88	[TCP Dup ACK 515#1] 5001 → 51126 [ACK] Seq=29 Ack=239157 Win=397312 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110 SRE=241133
519	6.868393	10.0.1.1	10.0.0.1	TCP	1054	[TCP Retransmission] 51126 → 5001 [ACK] Seq=29 Ack=239157 Win=397312 Tsvl=2148944296 TSscr=3000075129 SLE=242121 SRE=243110
520	6.872765	10.0.1.1	10.0.0.1	TCP	1054	[TCP Retransmission] 51126 → 5001 [ACK] Seq=241133 Ack=29 Win=397312 Tsvl=2148944296 TSscr=3000075134 SLE=242121 SRE=243110
521	6.872767	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [ACK] Seq=29 Ack=243110 Win=396288 Tsvl=2148944301 TSscr=3000075134 SLE=242121 SRE=243110
522	6.872748	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [ACK] Seq=29 Ack=243110 Win=396288 Tsvl=2148944301 TSscr=3000075134 SLE=242121 SRE=243110
523	6.876082	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [FIN, ACK] Seq=29 Ack=243110 Win=397824 Tsvl=2148944304 TSscr=3000075134
524	6.876187	10.0.1.1	10.0.0.1	TCP	66	51126 → 5001 [ACK] Seq=243110 Ack=29 Win=43908 Tsvl=3000075137 TSscr=2148944304

Figura 30: Paquetes capturados en interfaz de entrada del router (s2-eth1)

No.	Time	Source	Destination	Protocol	Length	Info
855	7.189087	10.0.1.1	10.0.0.1	TCP	278	51126 → 5001 [ACK] Seq=240145 Ack=29 Win=43908 Tsvl=3000075129 TSscr=2148944292
856	7.189117	10.0.1.1	10.0.0.1	TCP	78	5001 → 51126 [ACK] Seq=240157 Ack=29 Win=43908 Tsvl=2148944296 TSscr=3000075125 SLE=240145 SRE=241133
857	7.189140	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1846) [Reassembled in #858]
858	7.189141	10.0.1.1	10.0.0.1	TCP	278	51126 → 5001 [ACK] Seq=241133 Ack=29 Win=43908 Tsvl=3000075129 TSscr=2148944296
859	7.189145	10.0.1.1	10.0.0.1	TCP	810	[TCP Dup ACK 515#1] 5001 → 51126 [ACK] Seq=241133 Ack=29 Win=43908 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110
860	7.189147	10.0.1.1	10.0.0.1	TCP	278	51126 → 5001 [ACK] Seq=241133 Ack=29 Win=43908 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110
861	7.189162	10.0.1.1	10.0.0.1	TCP	88	[TCP Dup ACK 515#1] 5001 → 51126 [ACK] Seq=29 Ack=239157 Win=397312 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110 SRE=241133
862	7.189176	10.0.1.1	10.0.0.1	IPv4	810	Fragmented IP protocol (proto=TCP 6, off=0, ID=1846) [Reassembled in #863]
863	7.189177	10.0.1.1	10.0.0.1	TCP	278	[TCP Retransmission] 51126 → 5001 [ACK] Seq=241133 Ack=29 Win=43908 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110
864	7.189187	10.0.1.1	10.0.0.1	TCP	1054	51126 → 5001 [ACK] Seq=241133 Ack=29 Win=43908 Tsvl=2148944296 TSscr=3000075125 SLE=242121 SRE=243110
865	7.193475	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [ACK] Seq=243110 Ack=29 Win=396288 Tsvl=2148944301 TSscr=3000075134
866	7.193480	10.0.1.1	10.0.0.1	TCP	278	[TCP Retransmission] 51126 → 5001 [ACK] Seq=243110 Ack=29 Win=396288 Tsvl=2148944301 TSscr=3000075134
867	7.193525	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [ACK] Seq=243110 Ack=29 Win=396288 Tsvl=2148944301 TSscr=3000075134
868	7.196709	10.0.1.1	10.0.0.1	TCP	66	5001 → 51126 [FIN, ACK] Seq=29 Ack=243110 Win=397824 Tsvl=2148944304 TSscr=3000075134
869	7.196977	10.0.1.1	10.0.0.1	TCP	66	51126 → 5001 [ACK] Seq=243110 Ack=29 Win=43908 Tsvl=3000075137 TSscr=2148944304

Figura 31: Paquetes capturados en interfaz de salida del router (s2-eth0)

7. Dificultades Encontradas

7.1. Mininet

Inicialmente se tuvo que entender de manera profunda cómo funciona Mininet, para poder crear una topología modularizable en 3 apartados: cantidad de clientes, porcentaje de packet loss y reducción de MTU. Se tuvo que leer la documentación y documentación creada por la comunidad para lograr el cometido y crear una topología robusta.

A pesar de las dificultades quedamos satisfechos e incluso esto nos permitió compartir la topología con la feature de reducción de MTU con nuestros compañeros de manera pública a pedido del profesor, se puede encontrar esto en este link: [Linear ends topology with IP fragmentation](#).

7.2. Protocolo SAW

Teniendo sólida la teoría de cómo debería funcionar el protocolo nos encontramos con el problema de cómo, técnicamente, realizar la retransmisión, además de qué criterio tomar para espaciarlas y cuántas deberían haber.

Después de intentar con un enfoque poco escalable en términos de código (notamos que iba a haber mucha repetición de código) usando ciclos *while* en cada método que queríamos que retransmitiera, evolucionamos la idea a utilizar el patrón *decorator*, para manejar las veces que una recepción del socket se debería llevar a cabo, y un ciclo while genérico, aplicado a una abstracción del socket que hicimos (clase *SocketSAW*), para realizar las retransmisiones.

Este cambio de enfoque permitió desatascar progresivamente los problemas de debugging que nos fuimos encontrando a lo largo del TP, e.g. sequence number incorrecto, socket trabado en receive, entre otros.

7.3. Protocolo GBN

En términos de avance del proyecto, primero desarrollamos el protocolo SAW, por lo que para el comienzo del desarrollo del protocolo GBN ya teníamos el conocimiento de ejecución y el código ya armado para el SAW.

De esta manera, analizamos y tomamos la decisión de mantener las etapas de handshake, configuración y closing handshake de SAW, ya que esto no tenía sentido hacerlo de manera GBN, teniendo en cuenta los pocos paquetes que son, en comparación a un envío de archivo de miles de paquetes.

Esto llevó a realizar un manejo especial del primer y último chunk dentro de la base de código para asegurar el RDT, chunks que se podrían considerar como la "transición entre modos de comunicación.

Al comienzo del proyecto debatimos sobre si usar un mismo formato de header para ambos protocolos o si utilizar dos distintos. La decisión que tomamos fue utilizar un header apropiado para cada protocolo, de tal manera no arrastraríamos decisiones tomadas para GBN (como el ACK number, o el tamaño del sequence number) que no tienen sentido para SAW.

7.4. Plugin de Wireshark

Se encontraron dificultades para armar el plugin dinámico que reconozca el protocolo que se está utilizando y a partir de ello realizar el formateo del contenido del paquete, con lo cual, se adquirieron los conocimientos básicos del uso del lenguaje de programación Lua y también de la integración de dicho lenguaje con la herramienta Wireshark.

8. Conclusión

El desarrollo de este trabajo práctico permitió consolidar conceptos fundamentales sobre la comunicación entre procesos en redes de computadoras, especialmente aquellos vinculados a la

capa de transporte y a los mecanismos necesarios para lograr una transferencia de datos confiable (RDT) sobre el protocolo UDP. La implementación de los protocolos Stop-and-Wait y Go-Back-N brindó una comprensión más profunda de los desafíos involucrados en el diseño de mecanismos de retransmisión y confirmación.

Desde una perspectiva empírica, se comprobó que Go-Back-N ofrece un rendimiento superior al de Stop-and-Wait, dado que reduce la espera por confirmaciones al permitir el envío continuo de múltiples paquetes. Sin embargo, se evidenció la importancia de seleccionar adecuadamente el tamaño de ventana, ya que un valor grande puede generar que la retransmisión demore mucho.

Por otro lado, el uso de la interfaz de sockets en Python y la simulación de red mediante Mininet proporcionaron un entorno realista para evaluar el comportamiento de las implementaciones bajo condiciones adversas, como la pérdida de paquetes. En conjunto, el trabajo permitió articular teoría y práctica, fortaleciendo las habilidades necesarias para el diseño e implementación de protocolos de red personalizados.

9. Bibliografía

User Datagram Protocol RFC768 - IETF. Disponible en: <https://www.ietf.org/rfc/rfc768.txt>. Recuperado en Abril del 2025.

Transmission Control Protocol RFC793 - IETF. Disponible en: <https://www.ietf.org/rfc/rfc793.txt>. Recuperado en Abril del 2025.