

Trabajo Práctico N°1

[TA048] Redes
Cátedra Hamelin-Alvarez
Curso martes (Lopez/Horn)
Primer cuatrimestre del 2025

Alumno	Padrón	Email
Lorenzo Sebastián Ahumada	106780	lahumada@fi.uba.ar
Edgar Matías Campillay	106691	ecampillay@fi.uba.ar
Lara Daniela Converso	107632	lconverso@fi.uba.ar
Hernán de San Vicente	108800	hsanvicente@fi.uba.ar
Tobías Emilio Serpa	108266	tserpa@fi.uba.ar

Índice

1. Introduccion	2
2. Hipótesis y suposiciones realizadas	2
2.1. Hipótesis	2
3. Supuestos	2
3.1. Stop-and-wait	2
3.2. Go-back-n	3
4. Implementación	3
4.1. Protocolo de comunicación	3
4.1.1. Handshake - Upload	3
4.1.2. Handshake - Download	4
4.2. Validación de protocolo	5
4.3. Estructura y manejo de paquetes	5
4.4. Protocolo de recuperación de errores	7
4.5. Stop-and-wait	7
4.5.1. Detalles de implementación	7
4.6. Go-back-n	9
4.6.1. Descripción	9
4.6.2. Detalles de implementación	9
5. Pruebas	11
5.1. Sin pérdida	11
5.2. Con pérdida del 10 %	14
6. Preguntas a responder	16
6.1. Describa la arquitectura Cliente-Servidor	16
6.2. ¿Cuál es la función de un protocolo de capa de aplicación?	17
6.3. Detalle el protocolo de aplicación desarrollado en este trabajo	17
6.4. 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?	17
6.4.1. UDP	17
6.4.2. TCP	18
7. Dificultades encontradas	18
8. Conclusiones	19
9. Anexo: Fragmentación IPv4	19
9.1. Topología utilizada	19
9.2. Implementación	20
9.2.1. Tráfico TCP	20
9.3. Tráfico UDP	21
9.3.1. Cambio de cantidad de tráfico durante el cambio de MTU	21

1. Introduccion

El presente trabajo muestra la creación de una herramienta de carga y descarga de archivos entre cliente y servidor. Implementando un protocolo nuevo de capa de aplicación se asegura una transferencia de datos confiable (RDT), sobre una conexión UDP.

Para la implementación del protocolo de capa de aplicación se utilizaron 2 métodos de recuperación de pérdida de paquetes:

1. Stop-and-wait, el cual básicamente consiste en el envío secuencial de paquetes hasta el final.
2. Go-back-n, a diferencia del anterior, con GBN puedo enviar varios paquetes consecutivos al mismo tiempo.

2. Hipótesis y suposiciones realizadas

2.1. Hipótesis

- El archivo a enviar no puede superar los 10MB (10.000.000 Bytes).
- El servidor sobrescribe archivos recibidos con el mismo nombre.
- El cliente sobrescribe el archivo si recibe uno que ya tiene.
- No hay límite en la cantidad de conexiones simultáneas a un servidor.
- Un servidor solo puede establecer conexión con un cliente si comparten el mismo modo de recuperación de errores.
- El tiempo de espera sin respuesta por parte del sender durante una operación para determinar la desconexión es de 5 segundos.

3. Supuestos

- El servidor inicia en una carpeta predeterminada por el comando, por lo tanto los archivos que se buscarán en el caso de download deben encontrarse dentro de esta misma.

3.1. Stop-and-wait

- Timeouts de 0.5 segundos para realizar reenvios.
- Se realizaran 5 reintentos antes de asegurar la conexión como muerta.

3.2. Go-back-n

- La ventana soporta hasta 10 paquetes.
- Cada paquete puede ser reenviado hasta 3 veces inclusive antes de determinar la desconexión del servidor.
- Cada paquete tiene un timeout de 4 segundos antes de reenviarse junto con su ventana.

4. Implementación

4.1. Protocolo de comunicación

Antes de iniciar una transferencia de archivos, ya sea de subida (upload) o descarga (download), se realiza un proceso de triple handshake entre el cliente y el servidor. Este proceso implica tres pasos de comunicación entre ambos extremos y tiene como objetivo validar las condiciones necesarias para realizar la transferencia de manera segura y coherente.

4.1.1. Handshake - Upload

El proceso consta de tres pasos, lo que puede considerarse un triple handshake lógico:

Solicitud inicial del cliente El cliente localiza el archivo a subir, obtiene su tamaño y prepara un paquete con el flag **UPLOAD**, cuyo payload contiene el nombre del archivo y su tamaño en bytes, en el formato:

```
<file_name>\r\n<file_size>
```

1. Este paquete es enviado al servidor a través de sockets UDP.

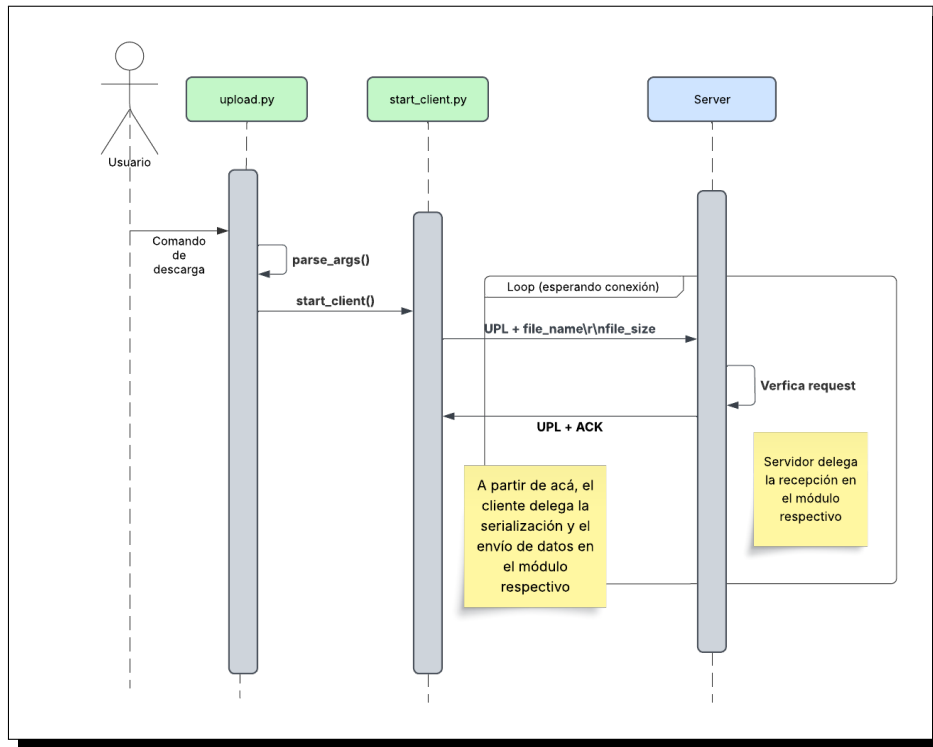
Respuesta del servidor El servidor recibe la solicitud, analiza si el archivo puede ser recibido (verifica el tamaño contra los límites de almacenamiento). En función de esto:

1. Si no hay espacio disponible, responde con un paquete **UPLOAD** sin **ACK**, rechazando la solicitud.
2. Si no hay espacio disponible, responde con un paquete **UPLOAD** sin **ACK**, rechazando la solicitud.

Confirmación implícita: El cliente interpreta la respuesta del servidor:

- Si es un **UPLOAD** junto con un **ACK**, da por finalizado el handshake y procede a transferir el archivo usando el protocolo elegido (sw o gbn).

- Si no recibe ACK, aborta la operación.



4.1.2. Handshake - Download

También en este caso hay tres pasos de comunicación que conforman un triple handshake:

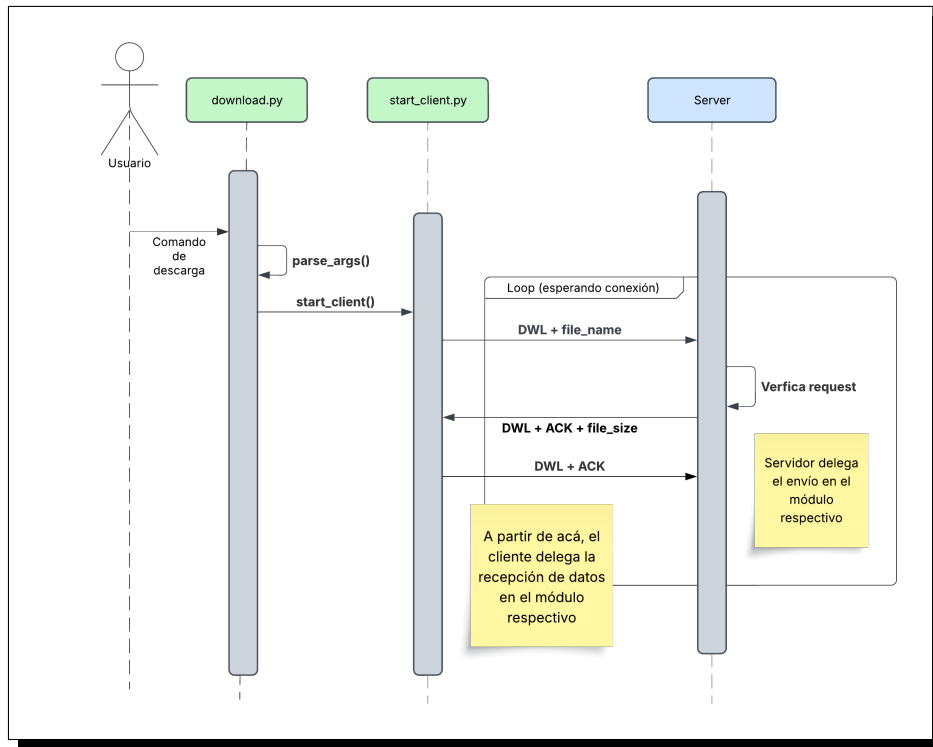
Solicitud inicial del cliente El cliente envía un paquete con el flag **DOWNLOAD** y el nombre del archivo que desea descargar como payload. Respuesta del servidor:

- El servidor verifica si el archivo existe:
 - Si no está disponible, se lanza un error.
 - Si el archivo existe, el servidor responde con un paquete **DOWNLOAD + ACK** cuyo payload contiene el tamaño del archivo en 4 bytes.

Confirmación del cliente El cliente evalúa si tiene espacio en disco para recibir el archivo:

- Si no hay suficiente espacio, responde con un paquete **DOWNLOAD** (sin **ACK**) y aborta la transferencia.
- Si sí hay espacio, responde con un paquete **DOWNLOAD + ACK**, confirmando que está listo para recibir los datos.

Una vez completado este tercer paso, el servidor procede a enviar el archivo usando el protocolo acordado.



4.2. Validación de protocolo

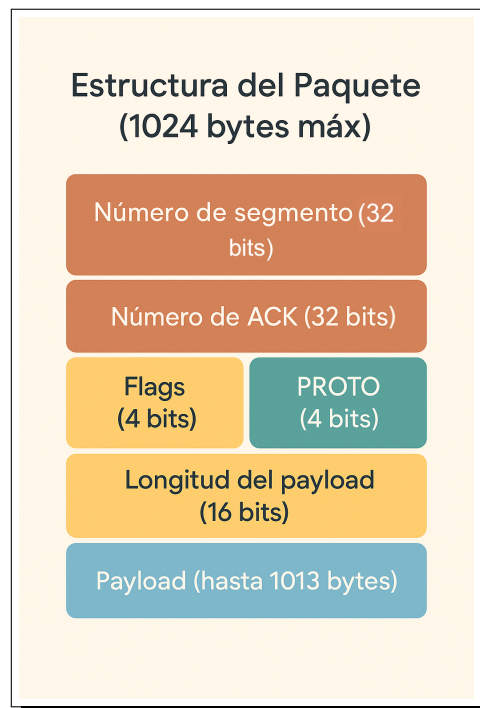
En ambos tipos de handshake, se valida que el protocolo solicitado por el cliente coincida con el configurado en el servidor. Si hay una discrepancia, se lanza un error y se aborta la operación por parte del cliente, el servidor sigue funcionando con normalidad.

4.3. Estructura y manejo de paquetes

La estructura del paquete utilizada en este proyecto está diseñada para garantizar una transferencia eficiente y confiable de datos entre nodos de la red. Cada paquete tiene un tamaño máximo de 1024 bytes (MMU) y está compuesto por varios campos que encapsulan tanto la información de control como los datos a transferir. La estructura del paquete es la siguiente:

- Número de segmento (32 bits): Identifica de manera única el paquete dentro de la secuencia de transmisión.
- Número de ACK (32 bits): Indica el número de segmento que el receptor está reconociendo como recibido correctamente.
- Flags (8 bits): Contiene indicadores de control, como:
 - UPL (Upload): Señala una solicitud de subida.
 - DWL (Download): Señala una solicitud de descarga.
 - ACK (Acknowledgment): Indica el reconocimiento de un paquete.

- **FIN** (Final): Marca el último paquete de una transmisión.
- **PROTO** (Protocolo): Especifica el protocolo de recuperación de errores (Stop-and-Wait o Go-Back-N).
- Longitud del payload (16 bits): Define el tamaño en bytes de los datos contenidos en el payload.
- Payload (hasta 1013 bytes): Contiene los datos reales que se están transfiriendo.



Para trabajar con esta estructura, el proyecto incluye herramientas específicas que permiten la manipulación y análisis de los paquetes:

- **Serialización:** La clase `PacketGenerator` y la función `serialize` en el módulo `serializer.py` permiten crear paquetes a partir de datos de entrada. Estas herramientas gestionan automáticamente los números de segmento, flags y otros campos necesarios para la transmisión.
- **Deserialización:** La función `deserialize` en el módulo `deserializer.py` permite reconstruir archivos a partir de los paquetes recibidos, asegurando que los datos se ensamblen correctamente en el lado receptor.
- **Extracción de datos:** Funciones como `get_seg_num`, `get_ack_num`, `get_payload`, y `get_protocol` en el módulo `serializer.py` permiten extraer información específica de los paquetes, como el número de segmento, el número de ACK, el payload, y el protocolo utilizado.

- Manejo de flags: Herramientas como `get_upl`, `get_dwl`, `get_ack`, y `get_fin` permiten verificar el estado de los flags en un paquete, facilitando la toma de decisiones durante la transmisión.

Estas herramientas están diseñadas para integrarse con los protocolos de recuperación de errores implementados (Stop-and-Wait y Go-Back-N), asegurando una comunicación robusta y eficiente en la red.

4.4. Protocolo de recuperación de errores

Se implementó un manejador de conexiones nuevas que se encarga de inicializar un “procesador” de requests de clientes cada vez que un cliente nuevo inicia una conexión. Cada procesador trabaja sobre un hilo de ejecución diferente y se encarga de procesar los mensajes de sus propios clientes y responder con el mensaje correspondiente.

Las conexiones con sockets UDP en python, a diferencia de la versión TCP, trabajan sobre un solo socket para la entrada y salida de mensajes, por lo que no se puede simplemente asignar un socket a cada procesador y que cada uno maneje su propia conexión, sino que surgió la necesidad de implementar un “router” de mensajes para que cada hilo de cliente procese solo los mensajes que se supone que debe procesar y nada más. Ésto se hizo utilizando un diccionario que usa como clave la dirección (IP + HOST) desde donde el cliente envía sus mensajes, y como valor una queue a través de la cual el manejador mete mensajes y el hilo correspondiente los obtiene. En caso de no existir un registro de la dirección recibida en el diccionario, se crea y se pone a correr el procesador en un nuevo hilo con su propia queue.

Cada procesador, previo a iniciar la transferencia de datos pedida por el cliente, realiza un triple-handshake de requests que valida que lo que se quiere hacer es viable (si es una operación válida, si se sigue el protocolo correcto, si hay suficiente espacio en el server, etc). Una vez completado este ida y vuelta de mensajes se inicia la transferencia con el protocolo correspondiente.

4.5. Stop-and-wait

El protocolo stop and wait permite el envío de paquetes de forma individual, asegurando que los paquetes lleguen ordenados. Luego de que un paquete es enviado, se espera por su correspondiente ack, y no se envía ningún paquete hasta la recepción de dicho ack.

Si un ack no es recibido, se envía el paquete nuevamente, y en caso de recibir un paquete fuera de orden, se envía el ack del último paquete que fue recibido correctamente.

4.5.1. Detalles de implementación

Cliente - Upload Cuando el cliente actúa como emisor, en el caso de una subida, implementa una lógica de reintentos para cada paquete.

Cada paquete se envía y se espera un ACK correspondiente. Si el ACK no llega en el tiempo estipulado (TIMEOUT), se reintenta el envío hasta un máximo de intentos (MAX_RETRIES).

Solo cuando el ACK recibido corresponde exactamente con el número de secuencia del paquete enviado se considera que ese paquete fue correctamente recibido.

Esto asegura que cada paquete sea confirmado individualmente, reduciendo la posibilidad de pérdida de datos. Además se corta la transferencia de paquetes en caso de que no haya respuesta luego de varios intentos, evitando así bloqueos indefinidos.

Cliente - Download Cuando el cliente actúa como receptor, en el caso de una bajada, se implementa un mecanismo que permite garantizar la recepción ordenada de los paquetes enviados por el servidor. Aplicando la lógica del protocolo, se espera que los paquetes lleguen en el orden correcto según su número de secuencia.

El cliente mantiene un número de secuencia esperado (`expected_seg_num`), que se incrementa solo cuando el paquete recibido coincide con el valor esperado. Si el paquete no está en orden (ya sea duplicado o adelantado), se ignora y se responde al servidor con un ACK del último paquete recibido correctamente.

De esta manera garantizamos que se descarten aquellos paquetes fuera de orden y se mantenga el orden de los datos.

En caso de que no se reciban paquetes dentro del tiempo establecido (TIMEOUT), se asume que la conexión con el servidor se ha perdido.

Servidor - Upload Para la subida de archivos (es decir, cuando el servidor actúa como receptor), se implementa un mecanismo que permite garantizar la recepción ordenada de los paquetes enviados por el cliente.

La lógica sigue el protocolo Stop-and-Wait y espera que los paquetes lleguen en el orden correcto según su número de secuencia.

El servidor mantiene un número de secuencia esperado (`expected_seg_num`), que se incrementa solo cuando el paquete recibido coincide con el valor esperado. Si el paquete no está en orden (ya sea duplicado o adelantado), se ignora y se responde al cliente con un ACK del último paquete recibido correctamente.

Este comportamiento garantiza que:

- Se descarten paquetes fuera de orden (ya sea porque llegaron adelantados o duplicados).
- Se mantiene el orden de los datos.
- El cliente puede retransmitir el paquete faltante cuando no reciba el ACK esperado.

Se utiliza un generador de paquetes que:

- Responde con un ACK por cada paquete recibido en orden.
- Marca el ACK como FIN si el paquete recibido también contenía dicha marca, indicando el final de la transferencia.
- En caso de que no se reciban paquetes dentro del tiempo establecido (TIMEOUT), se asume que la conexión con el cliente se ha perdido.

Servidor - Download Cuando el servidor actúa como emisor (en una descarga), implementa una lógica de reintentos para cada paquete.

Cada paquete se envía y se espera un ACK correspondiente. Si el ACK no llega en el tiempo estipulado (TIMEOUT), se reintenta el envío hasta un máximo de intentos (MAX_RETRIES).

Solo cuando el ACK recibido corresponde exactamente con el número de secuencia del paquete enviado se considera que ese paquete fue correctamente recibido.

Esto asegura que:

Cada paquete sea confirmado individualmente.

- Se reduzca la posibilidad de pérdida de datos.
- Se corte la transferencia si no hay respuesta después de varios intentos, para evitar bloqueos indefinidos.
- Este comportamiento es compatible con la lógica del cliente, que ignora paquetes fuera de orden, asegurando una transferencia secuencial y confiable.

4.6. Go-back-n

4.6.1. Descripción

El protocolo Go-Back-N, como mencionado anteriormente, permite el envío de paquetes de manera simultánea respetando, el orden de los mismos y el tamaño de la ventana (la cantidad total de paquetes a enviar sin ser reconocidos).

Cuando el paquete más viejo sin ser reconocido llega a timeout, se vuelven a enviar no solo el paquete asumido como perdido, sino todo los paquetes consecutivos pertenecientes a la ventana.

GBN vela por el orden de los paquetes recibidos manteniendo un trackeo del número esperado en el ACK recibido, a partir del número de segmento del paquete más viejo enviado, y comparándolos. En caso contrario ignora el paquete.

El trackeo de ACKs es útil porque provee al mecanismo de **cumulative acknowledgment**, por lo tanto, reconocer paquete $k+1$, significa que de $0...k$ inclusive fueron recibidos exitosamente y en orden.

4.6.2. Detalles de implementación

Cliente - Subida Para implementar la subida (envío de paquetes) del lado del cliente se utilizó la técnica conocida como sliding window. Donde:

- Referencia al inicio de la ventana (**floor**): Corresponde al paquete más viejo que todavía no recibió ACK.
- Referencia al final de la ventana (**roof**): El último paquete enviado.
- Tamaño de la ventana (**WINDOW_SIZE**): Diferencia entre piso y techo, nunca superior a el límite definido, 10.

Manejados por un bucle que verifica que se envíen todos los elementos de la lista (`floor < len(packets)`), donde la recepción de un ACK asociado al `floor`, aumenta este en 1. Y el envío de paquetes, aumenta `roof` en 1. En caso de producirse un timeout se reinicia el valor de `roof = floor`, asegurando así que en la próxima iteración se realice el envío de la ventana nuevamente.

Cliente - Download A diferencia de la subida, la implementación de la descarga (recepción de paquetes) de datos es bastante simple, consiste en trackear el próximo paquete esperado a partir del número de secuencia del paquete (bytes 0 al 3 inclusive) recibido. Si el trackeo no coincide, se descarta el paquete ignorándolo.

Si se da el caso de recibir un paquete fuera de orden:

- Adelantado: Se ignora, el sender eventualmente notará la falta de ack, y lo volverá a reenviar.
- Atrasado: Si corresponde al paquete más viejo sin reconocer, se reconoce y se avanza. Si es un duplicado de un paquete ya recibido, solo se ignora.

Servidor - Upload El servidor recibe los datos enviados por el cliente usando el esquema Go-Back-N (GBN) implementado de forma secuencial.

Para eso el servidor mantiene un contador `expected_seq`, que indica el número de secuencia del siguiente paquete esperado (comienza en 1), y por cada paquete recibido:

- Si el número de secuencia coincide con el esperado:
 - Se guarda el paquete.
 - Se responde con un ACK que confirma el número recibido.
 - Se incrementa `expected_seq`.
- Si no coincide (duplicado o fuera de orden):
 - Se descarta el paquete.
 - Se reenvía el último ACK válido (correspondiente al último paquete correctamente recibido).

Este proceso se repite hasta recibir un paquete marcado como FIN, indicando el final de la transmisión.

Servidor - Download El servidor envía los datos al cliente utilizando la técnica Go-Back-N con sliding window: Se usan dos índices:

- `base`: el número de secuencia del primer paquete no confirmado (piso).
- `next_seqnum`: el número del próximo paquete a enviar (techo).

El servidor envía paquetes mientras haya espacio en la ventana (`next_seqnum < base + WINDOW_SIZE`).

Al recibir un ACK con número de secuencia mayor o igual al base, actualiza base y reinicia el temporizador.

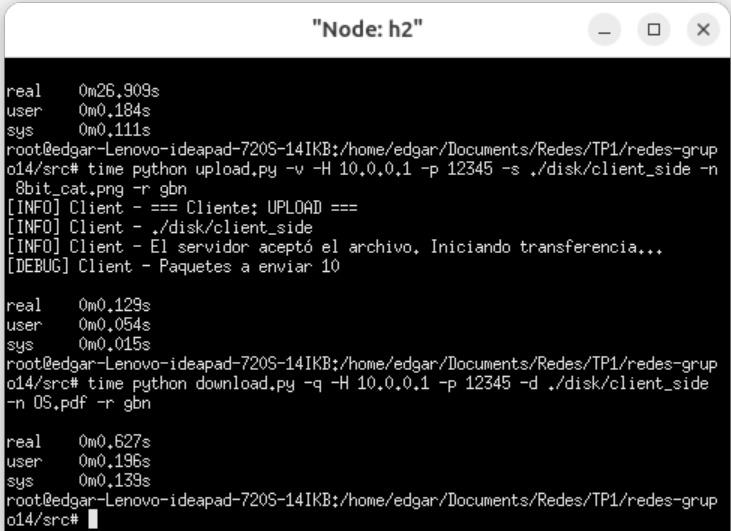
Si vence el temporizador del paquete base (no llega ACK en `TIMEOUT` segundos), se reenvían todos los paquetes en la ventana actual, desde base hasta `next_seqnum - 1`.

5. Pruebas

Para realizar las pruebas se utilizó una topología lineal, que incluye `h1 - s1 - s2 - h2` (similar a la del anexo).

Se utilizaron 2 archivos, uno pequeño `<1MB` (`8bit_cat.png`). y otro grande `>5MB` (`OS.pdf`)

5.1. Sin pérdida



```
"Node: h2"

real    0m26.909s
user    0m0.184s
sys     0m0.111s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n 8bit_cat.png -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 10

real    0m0.129s
user    0m0.054s
sys     0m0.015s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python download.py -q -H 10.0.0.1 -p 12345 -d ./disk/client_side -n OS.pdf -r gbn

real    0m0.627s
user    0m0.196s
sys     0m0.139s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```

"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grup
o14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n
OS.pdf -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589

real    0m26.909s
user    0m0.184s
sys     0m0.111s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grup
o14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n
8bit_cat.png -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 10

real    0m0.129s
user    0m0.054s
sys     0m0.015s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grup
o14/src#

```

```

"Node: h2"
o14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n
OS.pdf -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589
[ERROR] Client - Error: server hasn't sent any ack for packet 6

real    0m6.128s
user    0m0.078s
sys     0m0.031s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grup
o14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n
OS.pdf -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589

real    0m26.909s
user    0m0.184s
sys     0m0.111s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grup
o14/src#

```

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python download.py -q -H 10.0.0.1 -p 12345 -d ./disk/client_side -n 8bit_cat.png -r sw
real    0m0.094s
user    0m0.069s
sys     0m0.009s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python download.py -q -H 10.0.0.1 -p 12345 -d ./disk/client_side -n OS.pdf -r sw
real    0m0.636s
user    0m0.209s
sys     0m0.136s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n 8bit_cat.png -r sw
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 10

real    0m0.088s
user    0m0.067s
sys      0m0.013s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n OS.pdf -r sw
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589

real    0m0.602s
user    0m0.174s
sys      0m0.143s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

5.2. Con pérdida del 10%

Se agregó entre el switch 1 y switch 2 una pérdida del 10%. Y se lo reintentó con el archivo grande.

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python download.py -q -H 10.0.0.1 -p 12345 -d ./disk/client_side -n OS.pdf -r sw
real    0m0.615s
user    0m0.197s
sys     0m0.147s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n OS.pdf -r sw
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589
real    0m0.631s
user    0m0.171s
sys     0m0.149s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```



```
"Node: h2"
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python download.py -q -H 10.0.0.1 -p 12345 -d ./disk/client_side -n OS.pdf -r gbn

real    0m0.637s
user    0m0.206s
sys     0m0.132s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

```
"Node: h2"
o14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n OS.pdf -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589
[ERROR] Client - Error: server hasn't sent any ack for packet 4

real    0m6.126s
user    0m0.086s
sys     0m0.020s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src# time python upload.py -v -H 10.0.0.1 -p 12345 -s ./disk/client_side -n OS.pdf -r gbn
[INFO] Client - === Cliente: UPLOAD ===
[INFO] Client - ./disk/client_side
[INFO] Client - El servidor aceptó el archivo. Iniciando transferencia...
[DEBUG] Client - Paquetes a enviar 5589

real    0m27.090s
user    0m0.194s
sys     0m0.107s
root@edgar-Lenovo-ideapad-720S-14IKB:/home/edgar/Documents/Redes/TP1/redes-grupo14/src#
```

6. Preguntas a responder

6.1. Describa la arquitectura Cliente-Servidor

En una arquitectura Cliente-Servidor, un host llamado *Servidor* recibe solicitudes de otros nodos denominados *Clientes*. La comunicación es iniciada por el cliente,

mientras que el servidor se encarga de responder a dichas solicitudes.

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

La capa de aplicación tiene como objetivo principal permitir el desarrollo de aplicaciones capaces de ejecutarse en distintos hosts y comunicarse entre sí a través de una red. Esta capa proporciona los servicios necesarios para que las aplicaciones puedan intercambiar información.

6.3. Detalle el protocolo de aplicación desarrollado en este trabajo

El protocolo desarrollado se encuentra explicado en la sección 4.1.

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

La capa de transporte del modelo TCP/IP ofrece dos protocolos fundamentales: UDP (User Datagram Protocol) y TCP (Transmission Control Protocol). Cada uno brinda distintos servicios, con características que los hacen más adecuados para determinados escenarios.

6.4.1. UDP

UDP es un protocolo más simple y liviano que TCP. Sus principales características y servicios son:

- Chequeo de errores mediante un checksum, aunque no garantiza la corrección ni reenvío: el paquete puede ser descartado o pasado a la aplicación con advertencia.
- Comunicación proceso a proceso (process-to-process), facilitando el intercambio de datos entre aplicaciones específicas en distintos hosts.
- No requiere establecimiento previo de conexión.
- No mantiene estado de conexión.
- Tiene una carga de encabezado (header overhead) muy baja: solo 8 bytes.

Es apropiado utilizar UDP cuando:

- Se necesita baja latencia y se puede tolerar cierta pérdida de datos.
- Se desea un mayor control sobre cuándo y qué datos se entregan a la aplicación.

- Se quiere evitar la sobrecarga de mantener conexiones.

Ejemplos de uso: servicios como DNS, streaming de video o audio en tiempo real, VoIP y juegos en línea, donde pequeñas pérdidas de datos son aceptables.

6.4.2. TCP

TCP, a diferencia de UDP, ofrece una serie de servicios adicionales que permiten una comunicación confiable:

- Transferencia confiable de datos (Reliable Data Transfer): garantiza la entrega sin errores, en orden y sin duplicados.
- Control de flujo: evita que el emisor sature al receptor, ajustando la tasa de envío a la capacidad de lectura del receptor.
- Control de congestión: regula el tráfico en función de la congestión actual de la red, adaptando la velocidad de transmisión.

Aunque comparte con UDP el hecho de que no proporciona seguridad ni garantiza tiempos de transmisión, TCP es ideal para situaciones donde la integridad y el orden de los datos son esenciales.

7. Dificultades encontradas

- Definir las topologías a usar durante las pruebas con mininet y las maneras de generar la pérdida entre switches y hosts.
- Generar abstracción entre protocolos.
- Durante el desarrollo se intentaron implementar los módulos sin definir un paquete final lo cual al momento de la unión resultó en problemas de comunicación entre cliente y servidor.
- Armado del servidor listener, para la implementación del servidor se contemplaron inicialmente dos alternativas:
 - Un thread y socket por cada cliente, simulando una comunicación tipo TCP.
 - Un thread por cliente y un único socket de recepción con el uso de una cola de mensajería para cada cliente conectado. Implementación final.
- Cierre de operaciones, durante el desarrollo se contemplaron dos alternativas:
 - Enviar un paquete vacío FIN al final del envío para avisar al otro lado de la conexión que debe terminar.
 - Enviar el último paquete de la transmisión con el flag FIN, es útil porque al final de la función el tanto el cliente como el servidor asumen por finalizada la conexión.

- GBN, para implementar el protocolo de recuperación de errores se contemplaron 2 alternativas (ambas implementando sliding window technique):
 - 2 threads, listener y receiver.
 - Único thread, un bucle que envíe la ventana y revise las condiciones en cada iteración. Implementación final y común.

8. Conclusiones

A partir del desarrollo del proyecto se pudieron obtener los siguientes aprendizajes y resultados:

- Se profundizó el entendimiento sobre el uso de *sockets* UDP y la arquitectura Cliente-Servidor, reforzando conocimientos previos adquiridos.
- Se logró implementar un protocolo confiable mediante dos variantes: *Stop and Wait* y *Go Back N*. Ambas versiones permitieron realizar correctamente transferencias de archivos, mostrando una integración funcional entre protocolo y aplicación.
- Se observaron diferencias en el rendimiento entre *Stop and Wait* y *Go Back N*, especialmente a medida que aumentaba la pérdida de paquetes, donde *Go Back N* demostró ser notablemente más eficiente.
- Se identificó que el protocolo *Go Back N* podría beneficiarse de futuras mejoras, como conservar paquetes recibidos fuera de orden en lugar de descartarlos, optimizando así su desempeño general.

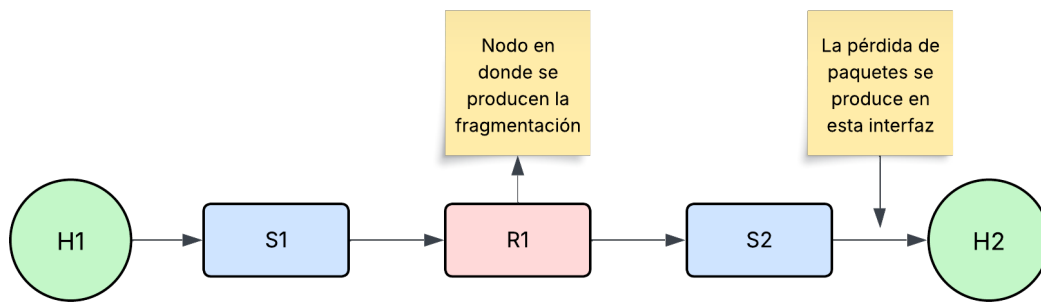
9. Anexo: Fragmentación IPv4

9.1. Topología utilizada

Para realizar la experiencia se simuló una red lineal compuesta de:

1. 2 hosts, fuente y destino;
2. 2 switches,
3. 1 router.

El objetivo de la misma es producir tanto la fragmentación como la pérdidas de paquetes en ambos tipos de protocolos (TCP/UDP) y documentarlos usando la herramienta Wireshark.



Para reducir el MTU de una de las interfaces se ejecutó:

```
ifconfig r1-eth2 mtu <bytes_size>
```

Para forzar la pérdida de paquetes en la interfaz entre el segundo switch y el último nodo se ejecutó:

```
tc qdisc add dev s2-eth2 root netem loss 10%
```

Para desactivar el flag DF, que permite la fragmentación en ambos host, se ejecutó:

```
sysctl -w net.ipv4.ip_no_pmtu_disc=
```

9.2. Implementación

9.2.1. Tráfico TCP

Para generar tráfico entre los hosts se realizaron los siguientes pasos:

- Iniciar el servidor en el host destino, H2 y dejarlo escuchando

```
h2 iperf -s &
```

- Iniciar la conexión TCP entre ambos hosts, donde el packet size debe ser mayor al tamaño del MTU.

```
h1 iperf -c h2 -l <packet_size>
```

Con un MTU de 600 y un tamaño de paquetes de 800 se consiguieron los siguientes resultados:

40	0.007387980	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=857f)
41	0.007391668	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=857f)
42	0.007393776	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=0, ID=8580) [Reassembled in #44]
43	0.007395045	10.0.0.2	10.0.0.6	IPv4	610	Fragmented IP protocol (proto=TCP 6, off=576, ID=8580) [Reassembled in #44]
44	0.007396293	10.0.0.2	10.0.0.6	TCP	362	[TCP Previous segment not captured] 44940 → 5001 [PSH, ACK] Seq=15281 Ack=1

Dónde se puede ver la fragmentación:

- 2 paquetes pertenecientes a un mismo ID,
- Cada uno de los paquetes fragmentados con un offset correspondientes al MTU establecido previamente.

Y la pérdida de uno de estos, **TCP Previous segment not captured**.

TCP al proveer de RDT al transporte de datos, vuelve a enviar los paquetes. Se puede ver en Wireshark en la descripción de los mismos con los paquetes retransmitidos.

227	12.953169361	10.0.0.2	10.0.0.6	TCP	1516	[TCP Retransmission] 44074 → 5001 [ACK] Seq=1513 Ack=1 Win=42496 Len=1448 TSval=928143669 TSecr=909808151
230	12.953170856	10.0.0.2	10.0.0.6	TCP	364	[TCP Retransmission] 44074 → 5001 [ACK] Seq=1513 Ack=1 Win=42496 Len=1448 TSval=928143669 TSecr=909808151
232	12.953249818	10.0.0.2	10.0.0.6	TCP	1516	[TCP Retransmission] 44074 → 5001 [ACK] Seq=2961 Ack=1 Win=42496 Len=1448 TSval=928143669 TSecr=909808151

9.3. Tráfico UDP

Mismo procedimiento anterior, cambiamos únicamente los flags de los comandos para forzar tráfico UDP:

```
h2 iperf -s -u &
h1 iperf -c h2 -u -l <packet_size>
```

Nuevamente con un MTU de 1500 y un tamaño de paquetes de 2000 se consiguieron los siguientes resultados:

1220	29.9025336...	10.0.0.2	10.0.0.6	IPv4	562	Fragmented IP protocol (proto=UDP 17, off=1480, ID=4f94)
1221	29.9142539...	10.0.0.2	10.0.0.6	IPv4	562	Fragmented IP protocol (proto=UDP 17, off=1480, ID=4f96)
1222	29.9372239...	10.0.0.2	10.0.0.6	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=4f9c)
1223	29.9459177...	10.0.0.2	10.0.0.6	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=4f9e) [Reassembled in #1224]
1224	29.9459244...	10.0.0.2	10.0.0.6	UDP	562	48442 → 5001 Len=2000

Donde la fragmentación se ve a través de los paquetes 1220, 21 y 22. Y la pérdida al ser UDP y no contar con un mecanismo de reenvío sólo se puede observar notando la falta de Reassembled packets al final de la comunicación.

9.3.1. Cambio de cantidad de tráfico durante el cambio de MTU

Con un MTU de 1500 y un tamaño de paquete 2000. Se enviaron en toda la red un total de 1178 paquetes (contando fragmentados).

Con un MTU reducido de 750 y el mismo tamaño de paquete 2000. Se enviaron un total de 1215 paquetes (contando fragmentados).