



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

## TP N° 1: File Transfer

Alumnos

|                          |                    |         |
|--------------------------|--------------------|---------|
| Mundani Vegega, Ezequiel | emundani@fi.uba.ar | 102.312 |
| Sicca, Fabio             | fsicca@fi.uba.ar   | 104.892 |

ASIGNATURA

TA048 - Redes

9 de mayo de 2024

# Índice

|   |           |
|---|-----------|
| <b>1. Introducción</b>  | <b>2</b>  |
| <b>2. Hipótesis y suposiciones realizadas</b>                                 | <b>2</b>  |
| <b>3. Implementación</b>  | <b>3</b>  |
| 3.1. Programas . . . . .  | 3         |
| 3.2. Mensajes . . . . .   | 3         |
| 3.3. Handshake . . . . .  | 4         |
| 3.4. Protocolos implementados . . . . .                                       | 5         |
| 3.4.1. Protocolo Stop And Wait . . . . .                                      | 5         |
| 3.4.2. Protocolo Go Back N . . . . .  | 6         |
| <b>4. Ejemplos</b>  | <b>7</b>  |
| 4.1. Stop & Wait . . . . .  | 7         |
| 4.2. Go Back N . . . . .  | 7         |
| 4.3. Pruebas temporales . . . . .   | 8         |
| 4.4. Pruebas de verificación de datos . . . . .                               | 9         |
| <b>5. Preguntas a responder</b>   | <b>10</b> |
| 5.1. Describa la arquitectura Cliente-Servidor . . . . .                      | 10        |
| 5.2. ¿Cuál es la función de un protocolo de capa de aplicación? . . . . .     | 10        |
| 5.3. Detalle el protocolo de aplicación desarrollado en este trabajo. . . . . | 10        |
| 5.4. Descripción de TCP y UDP . . . . .                                       | 11        |
| <b>6. Dificultades encontradas</b>  | <b>11</b> |
| <b>7. Conclusiones</b>  | <b>12</b> |

## 1. Introducción

Este trabajo práctico consiste en la implementación de un protocolo de aplicación RDT (Reliable Data Transfer) para la transferencia de archivos entre múltiples clientes y un servidor. Para ello se hace uso de sockets, threads y el protocolo de capa de transporte UDP (User Datagram Protocol). Para lograr una efectiva y confiable transferencia de datos a partir del mismo se desarrolló la lógica de los protocolos “Stop And Wait” y “Go Back N” en la capa de aplicación que estará por arriba de la capa de transporte.

Para comprobar el correcto funcionamiento de los programas hechos, se enviarán archivos de manera local, primero directamente y luego utilizando la herramienta Comcast para simular pérdida de paquetes.

## 2. Hipótesis y suposiciones realizadas

- El número de secuencia máximo (siendo un entero de 4 bytes) es  $2^{32} - 1$ , por lo que el tamaño máximo de archivo soportado sería  $2^{32} \cdot \text{TAMAÑO\_PAYLOAD}$ . Suponiendo un payload de 1000 bytes, entonces el tamaño máximo sería aproximadamente 4294 GB.
- Si se sube un archivo al servidor con nombre igual a uno ya guardado en él, este se sobrescribe.
- Dado que UDP posee un checksum para corroborar la corrupción en los paquetes, no se hizo una implementación propia del mismo. El protocolo UDP por defecto tiene un checksum y ante la llegada de un paquete corrupto, este es descartado. Para esta implementación de protocolo de aplicación, un paquete corrupto y un paquete perdido son lo mismo, dado a que no llegan a la capa de aplicación. Podría darse el caso de una corrupción de múltiples bits, lo que haría que sí llegue un archivo corrupto, pero esta situación es tan poco probable que no se la considera.
- En caso de no existir el archivo a descargar del servidor o no existir el archivo a subir, los programas no se ejecutan.

### 3. Implementación

#### 3.1. Programas

Se desarrolló una arquitectura cliente servidor, en la cual el servidor es capaz de procesar múltiples conexiones simultáneamente. Existen tres programas: `start_server.py`, `upload.py` y `download.py`. Estos dos últimos podrían haber sido uno solo, pero la consigna pedía que estén explícitamente, la diferencia entre uno y otro es el tipo de transferencia con el que crean la conexión (*Upload* o *Download*). Los tres programas inician procesando sus argumentos por línea de comandos. Además, cada programa cuenta con un logging, cuya verbosidad es configurable por argumento.

Argumentos de `start_server.py`:

`-h|--help` para mostrar ayuda, `-v|--verbose` para aumentar verbosidad, `-q|--quiet` para disminuir la verbosidad, `-H|--host` para definir la dirección IP del servidor, `-p|--port` para definir el puerto *listener* y `-s|--storage` para definir dónde se guardarán/buscarán los archivos.

Argumentos de `upload.py`:

`-h|--help` para mostrar ayuda, `-v|--verbose` para aumentar verbosidad, `-q|--quiet` para disminuir la verbosidad, `-H|--host` para definir la dirección IP del servidor, `-p|--port` para definir el puerto *listener* del servidor, `-s|--src` para definir dónde está el archivo a subir, `-n|--name` para definir el archivo a subir y `-r` para definir el protocolo de comunicación.

Argumentos de `download.py`:

`-h|--help` para mostrar ayuda, `-v|--verbose` para aumentar verbosidad, `-q|--quiet` para disminuir la verbosidad, `-H|--host` para definir la dirección IP del servidor, `-p|--port` para definir el puerto *listener* del servidor, `-d|--dst` para definir dónde guardar el archivo a descargar, `-n|--name` para definir el archivo a descargar (debe existir en el servidor) y `-r` para definir el protocolo de comunicación.

#### `start_server.py`

En caso de haber procesado correctamente los argumentos, se crea un objeto `Server`, que tendrá los atributos de este y luego se crea un socket por el cual se escucharán nuevas conexiones.

#### `upload.py/download.py`

En caso de haber procesado correctamente los argumentos, se crea un objeto `Connection`, el cual tendrá todos los atributos referidos a la conexión que se intentará establecer. Luego se creará un thread *Idle Timeout* y un objeto `Client` con esta, y se intentará establecer la conexión con el server a partir de un *handshake*, el cual es similar al *3 way handshake* de TCP.

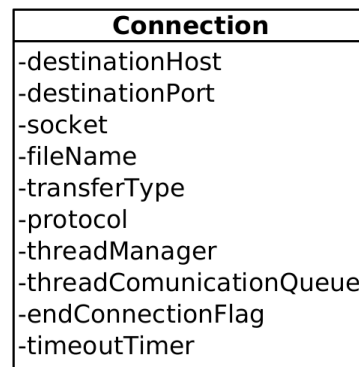


Figura 1: clase Connection

#### 3.2. Mensajes

Antes de seguir describiendo la lógica del *handshake* conviene ver los tipos de mensajes de los protocolos implementados. El tipo de cada uno es guardado en el *header* de cada mensaje. Cada distinto tipo de mensaje sabe codificarse y decodificarse.

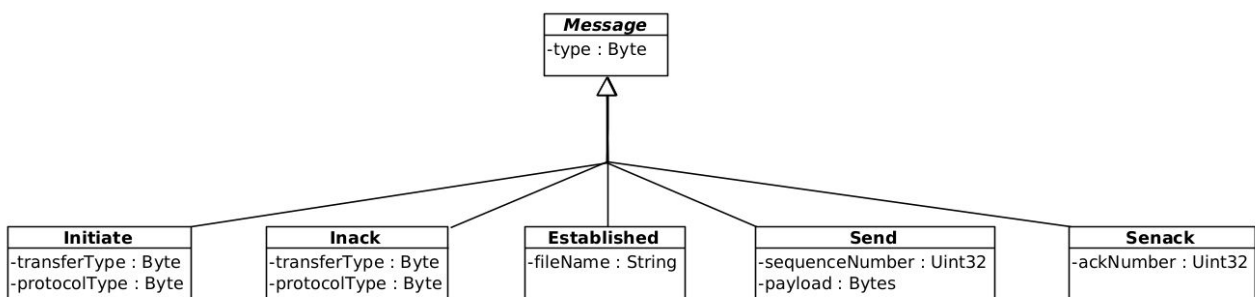


Figura 2: clase Message y sus hijos

### 3.3. Handshake

Teniendo ya al servidor y al cliente corriendo, se intentará realizar un 3 way handshake entre estos, el cual funciona de la siguiente manera: el cliente intentará establecer una conexión al socket listener del servidor enviando un mensaje *Initiate*, el cual contiene únicamente la información del tipo de transferencia y el protocolo. El server procesa ese *Initiate*, y si puede aceptar la conexión y no tiene una conexión con ese cliente todavía crea un nuevo socket dedicado exclusivamente para esa conexión. Desde este nuevo socket responde con un *Inack*, devolviendo también el tipo de transferencia y el protocolo, a modo de aceptación. Luego crea un thread *Idle Timeout*, para poder cerrar la conexión en caso de ser necesario.

Finalmente el cliente recibe este *Inack*, del cual obtiene la dirección creada para su comunicación con el servidor y finalmente envía un *Established*, con el nombre del archivo a subir/descargar. Desde el lado del cliente aquí termina el handshake y la conexión ha sido establecida. Del lado del server, una vez que llega el *Established* es que la conexión queda establecida.

En caso de perderse el *Initiate*, el Client simplemente lo seguirá reenviando. En caso de perderse el *Inack*, el Client seguirá enviando el *Initiate*, pero el servidor no aceptará esta conexión por ya tenerla guardada, por lo que eventualmente el Client hará timeout y se cerrará, teniendo que ejecutarse de nuevo. En caso de perderse el *Established*, tanto servidor como cliente terminarán haciendo timeout, siendo que el server estará a la espera del *Established* mientras recibe *Sends* (si es Downloader) o mientras los tiene que enviar (si es Uploader). Esto conlleva reiniciar al programa Client y reintentar la conexión.

Una vez realizado exitosamente el *handshake*, tanto Server como Client crean sus threads *Sender* y *Receiver*, dependiendo de qué protocolo utilicen y de su tipo de transferencia. Cabe aclarar que la lógica de estos depende solo del protocolo y el tipo de transferencia, no de si los implementan el servidor o el cliente.

Creados ya los threads, comienza la comunicación entre estos, utilizando el protocolo elegido.

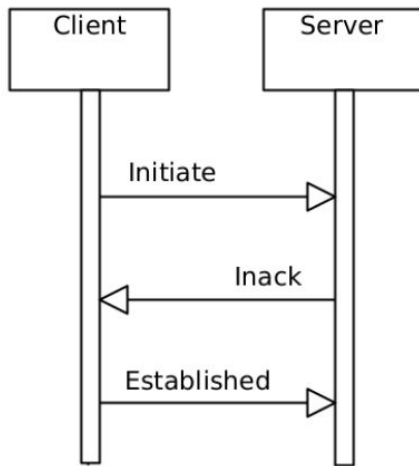


Figura 3: Handshake exitoso

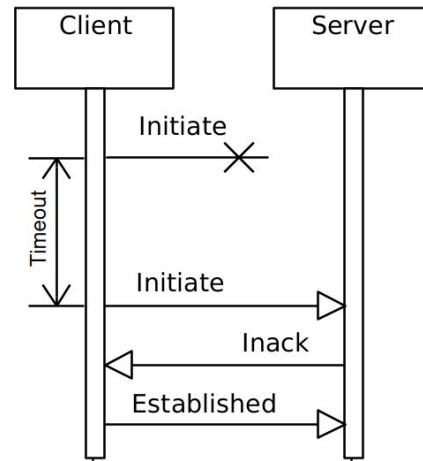


Figura 4: Pérdida del Initiate

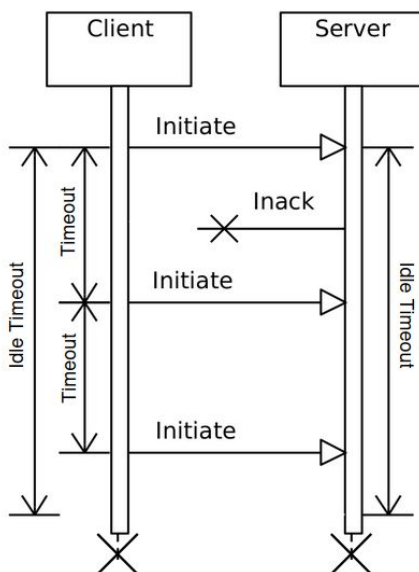


Figura 5: Pérdida del Inack

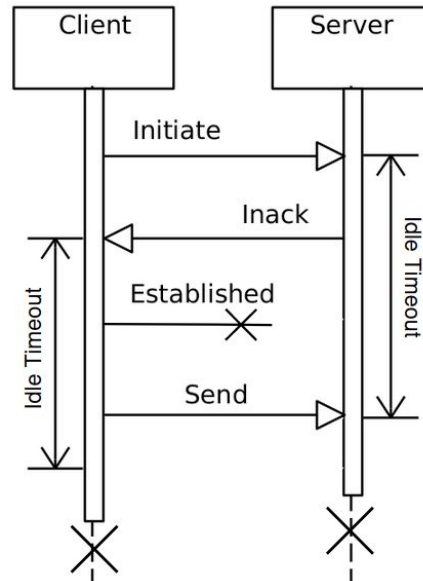


Figura 6: Pérdida del Established

### 3.4. Protocolos implementados

Independientemente del protocolo existen dos clases de mensajes a utilizar: Send y Senack. Send tiene un número de secuencia y un payload de longitud fija, en el caso de este TP se eligió 1000 bytes, debido a que se consideró que resulta útil para el ámbito académico poder analizar el envío de muchos paquetes, y tampoco está tantos órdenes de magnitud alejado del payload de TCP (64kB). Senack tiene el número de acknowledgement (ack number).

Cada conexión (ya sea del Server o del Client) tendrá un tipo de transferencia (Upload o Download), cada una con 3 threads: uno sender, uno receiver y uno para el *Idle Timeout*. El programa que sea Uploader irá leyendo el archivo y enviándolo de a trozos en Sends, mientras que el Downloader lo irá escribiendo a partir de los Sends que recibe.

Cabe aclarar que el *Idle Timeout* utilizado es de 10s y el *Timeout* (tiempo hasta el reenvío de Sends) es de 100ms.

#### 3.4.1. Protocolo Stop And Wait

##### Uploader:

Ya sea del Server o del Client, el thread sender envía un mensaje Send con un número de secuencia y espera a recibir un Senack con el mismo número de secuencia. Luego de un tiempo (timeout, establecido en el socket) se reenvía el Send. De esta manera se puede seguir manteniendo una conexión si se pierde tanto el Send como el Senack. En el thread receiver se escucha para obtener un mensaje, de ser obtenido este es enviado al thread Sender y procesado ahí. Si el Senack fue procesado correctamente (el ACK number es igual al número de secuencia del Send enviado), significa que el Send fue recibido correctamente y se envía el siguiente, hasta terminar de enviar el archivo entero (el cual es leído de a partes, en vez de ser cargado entero en memoria, permitiendo enviar archivos de gran tamaño). Una vez que se envió y se sabe de la recepción correcta de todas las partes del archivo (gracias a la recepción del último Senack), ambos threads del Uploader finalizan, terminando con la conexión. En el programa Server esto significa que se cierra la conexión abierta, pero el programa continúa su ejecución. En el programa Client esto significa la finalización del programa. En caso de no recibir ningún mensaje por parte del Downloader ocurre un *idle\_timeout* y finaliza la ejecución del programa.

##### Downloader:

Nuevamente, independientemente de si es en el Server o Client, el thread receiver escucha para recibir los Send. A partir de sus payload se va escribiendo el archivo a recibir. Con el sequence number del Send se crea un Senack, el cual es enviado por el thread sender. En caso de no recibir más archivos por parte del Uploader, eventualmente ocurre un *timeout* que finaliza la conexión (esto ocurre tanto en el escenario de haber recibido todo el archivo, que se haya caído el Uploader o que haya pasado demasiado tiempo hasta que lleguen los mensajes).

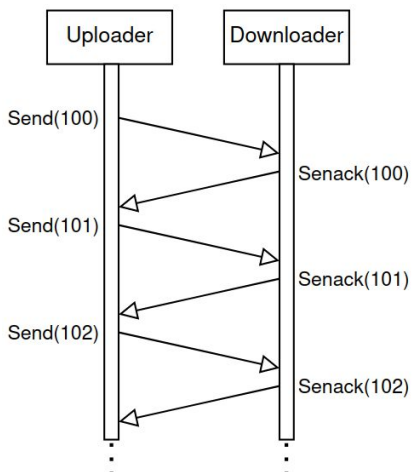


Figura 7: correcto funcionamiento.

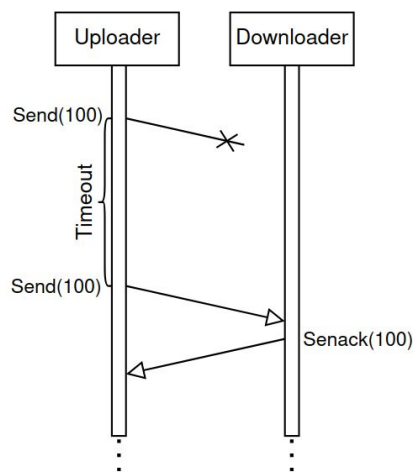


Figura 8: pérdida del Send.

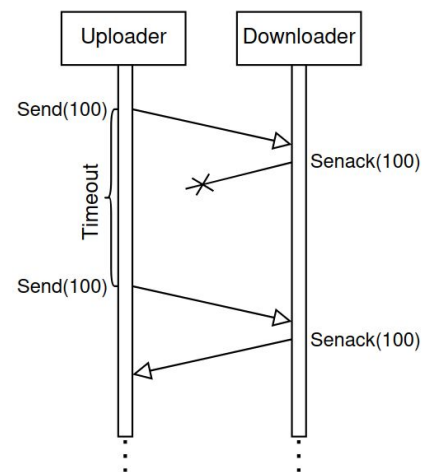


Figura 9: pérdida del Senack.

### 3.4.2. Protocolo Go Back N

Nuevamente un programa (cliente o servidor) será el Uploader y el otro el Downloader, son bastante análogas las implementaciones con el Stop And Wait, lo que cambia es la lógica de envío y recepción de Send y Senacks. En este caso el Uploader envía hasta una ventana definida de Sends (4 en esta implementación) y no se envía otro hasta no obtener el Senack del Send enviado con mayor antigüedad (el de menor número de secuencia). A medida que se van recibiendo los Senacks, del archivo más antiguo (el cual se va actualizando) se envían nuevos Send, teniendo siempre hasta el tamaño de ventana de Sends sin su Senack.

En caso de perderse un mensaje Send, el Downloader empezará a recibir Sends desordenados. Estos serán descartados, siendo que el Uploader los reenviará al reenviar toda la ventana de Sends que no hayan obtenido su Senack. Luego de cierto tiempo sin recibir el Senack esperado, ocurre el timeout de envío de ventana, el cual indica que se debe reenviar la ventana entera de Sends. Con todos los Senacks obtenidos se va guardando el archivo. En caso de obtener mensajes desordenados, el Downloader responde con un Senack pero con número de ACK del último Send recibido correctamente.

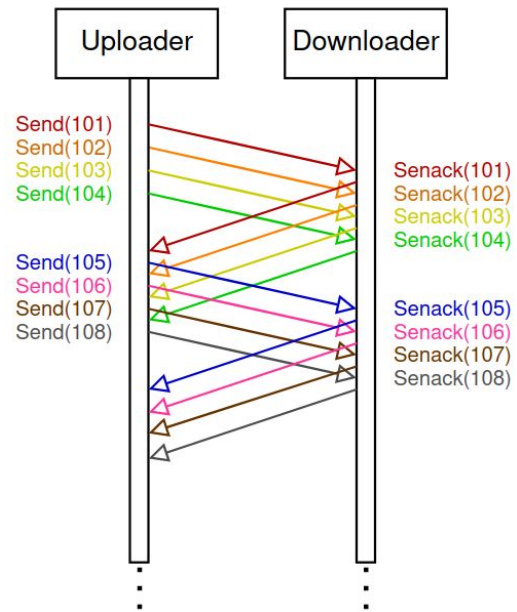


Figura 10: Go Back N sin pérdidas de paquetes.

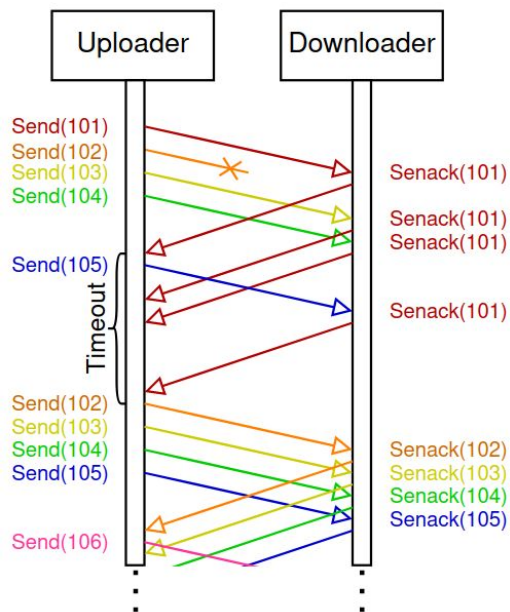


Figura 11: Go Back N con pérdida de Send.

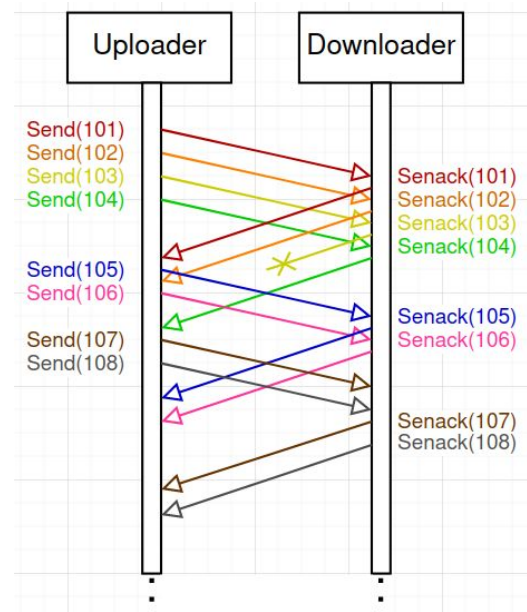
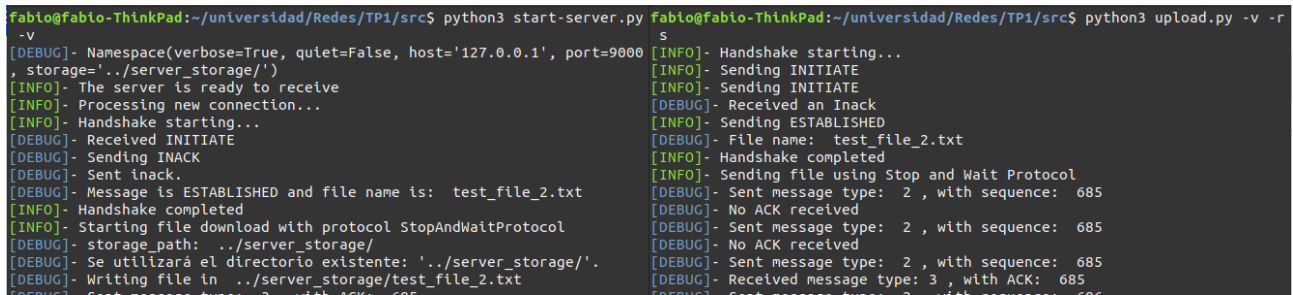


Figura 12: Go Back N con pérdida de Senack.

## 4. Ejemplos

### 4.1. Stop & Wait



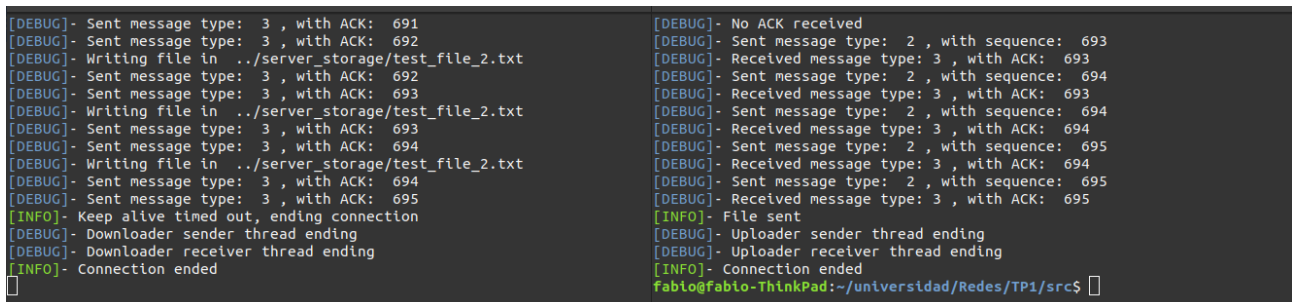
```

fabio@fabio-ThinkPad:~/universidad/Redes/TP1/src$ python3 start-server.py -v
[DEBUG]- Namespace(verbose=True, quiet=False, host='127.0.0.1', port=9000, storage='../server_storage/')
[INFO]- The server is ready to receive
[INFO]- Processing new connection...
[INFO]- Handshake starting...
[DEBUG]- Received INITIATE
[DEBUG]- Sending INACK
[DEBUG]- Sent inack.
[INFO]- Message is ESTABLISHED and file name is: test_file_2.txt
[INFO]- Handshake completed
[INFO]- Starting file download with protocol StopAndWaitProtocol
[DEBUG]- storage_path: ../server_storage/
[DEBUG]- Se utilizará el directorio existente: '../server_storage/'.
[DEBUG]- Writing file in ../server_storage/test_file_2.txt

fabio@fabio-ThinkPad:~/universidad/Redes/TP1/src$ python3 upload.py -v -r
[INFO]- Handshake starting...
[INFO]- Sending INITIATE
[INFO]- Sending INITIATE
[DEBUG]- Received an Inack
[INFO]- Sending ESTABLISHED
[DEBUG]- File name: test_file_2.txt
[INFO]- Handshake completed
[INFO]- Sending file using Stop and Wait Protocol
[DEBUG]- Sent message type: 2, with sequence: 685
[DEBUG]- No ACK received
[DEBUG]- Sent message type: 2, with sequence: 685
[DEBUG]- No ACK received
[DEBUG]- Sent message type: 2, with sequence: 685
[DEBUG]- Received message type: 3, with ACK: 685

```

Figura 13: Inicio de conexión con programa Upload usando Stop & Wait



```

[DEBUG]- Sent message type: 3, with ACK: 691
[DEBUG]- Sent message type: 3, with ACK: 692
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3, with ACK: 692
[DEBUG]- Sent message type: 3, with ACK: 693
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3, with ACK: 693
[DEBUG]- Sent message type: 3, with ACK: 694
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3, with ACK: 694
[DEBUG]- Sent message type: 3, with ACK: 695
[INFO]- Keep alive timed out, ending connection
[DEBUG]- Downloader sender thread ending
[DEBUG]- Downloader receiver thread ending
[INFO]- Connection ended

[DEBUG]- No ACK received
[DEBUG]- Sent message type: 2, with sequence: 693
[DEBUG]- Received message type: 3, with ACK: 693
[DEBUG]- Sent message type: 2, with sequence: 694
[DEBUG]- Received message type: 3, with ACK: 693
[DEBUG]- Sent message type: 2, with sequence: 694
[DEBUG]- Received message type: 3, with ACK: 694
[DEBUG]- Sent message type: 2, with sequence: 695
[DEBUG]- Received message type: 3, with ACK: 694
[DEBUG]- Sent message type: 2, with sequence: 695
[DEBUG]- Received message type: 3, with ACK: 695
[INFO]- File sent
[DEBUG]- Uploader sender thread ending
[DEBUG]- Uploader receiver thread ending
[INFO]- Connection ended
fabio@fabio-ThinkPad:~/universidad/Redes/TP1/src$

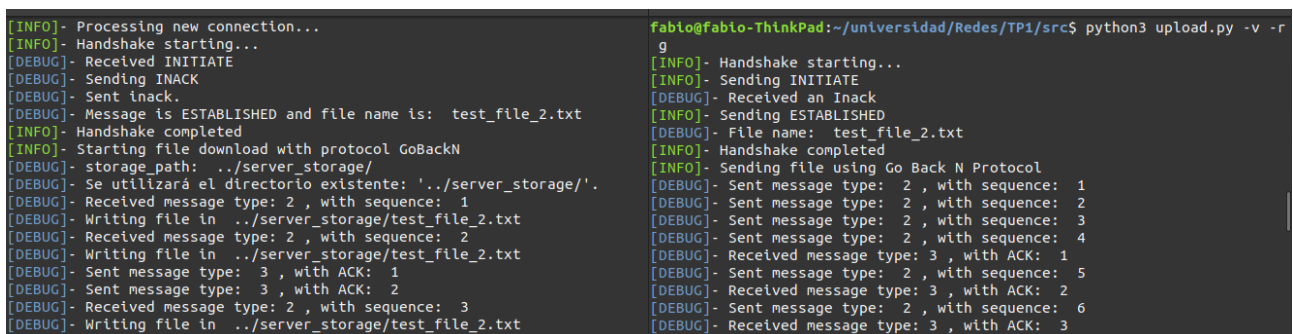
```

Figura 14: Fin de conexión Upload usando Stop & Wait

En este ejemplo se ejecuta el servidor y un cliente hace un upload con el protocolo Stop And Wait. Cosas interesantes a observar:

- El handshake al inicio entre el cliente y el servidor
- El cliente le envía el nombre del archivo al servidor para que lo guarde
- El servidor recibe 1005 bytes y escribe 1000 bytes en el archivo, los otros 5 son de la cabecera
- La escritura del archivo en el servidor cada vez que llega un paquete que no está repetido
- La pérdida de ACKS demostrada por la frase “No ACK received” y su posterior reenvío del paquete hacia el servidor (del lado del cliente, a la derecha)
- El funcionamiento del protocolo Stop And Wait que espera a obtener el ACK antes de mandar el siguiente paquete (o en su defecto lo reenvía si se produce un Time Out)
- El cierre de los hilos sender y receiver de ambos lados
- La finalización de la ejecución del lado del cliente, mientras que el servidor sigue a la espera de nuevas conexiones.

### 4.2. Go Back N



```

[INFO]- Processing new connection...
[INFO]- Handshake starting...
[DEBUG]- Received INITIATE
[DEBUG]- Sending INACK
[DEBUG]- Sent inack.
[INFO]- Message is ESTABLISHED and file name is: test_file_2.txt
[INFO]- Handshake completed
[INFO]- Starting file download with protocol GoBackN
[DEBUG]- storage_path: ../server_storage/
[DEBUG]- Se utilizará el directorio existente: '../server_storage/'.
[DEBUG]- Received message type: 2, with sequence: 1
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Received message type: 2, with sequence: 2
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3, with ACK: 1
[DEBUG]- Sent message type: 3, with ACK: 2
[DEBUG]- Received message type: 2, with sequence: 3
[DEBUG]- Writing file in ../server_storage/test_file_2.txt

fabio@fabio-ThinkPad:~/universidad/Redes/TP1/src$ python3 upload.py -v -r
[INFO]- Handshake starting...
[INFO]- Sending INITIATE
[INFO]- Sending INITIATE
[DEBUG]- Received an Inack
[INFO]- Sending ESTABLISHED
[DEBUG]- File name: test_file_2.txt
[INFO]- Handshake completed
[INFO]- Sending file using Go Back N Protocol
[DEBUG]- Sent message type: 2, with sequence: 1
[DEBUG]- Sent message type: 2, with sequence: 2
[DEBUG]- Sent message type: 2, with sequence: 3
[DEBUG]- Sent message type: 2, with sequence: 4
[DEBUG]- Received message type: 3, with ACK: 1
[DEBUG]- Sent message type: 2, with sequence: 5
[DEBUG]- Received message type: 3, with ACK: 2
[DEBUG]- Sent message type: 2, with sequence: 6
[DEBUG]- Received message type: 3, with ACK: 3

```

Figura 15: Inicio de conexión upload usando Go Back N



```

[DEBUG]- Sent message type: 3 , with ACK: 8
[DEBUG]- Sent message type: 3 , with ACK: 8
[DEBUG]- Received message type: 2 , with sequence: 11
[DEBUG]- Sent message type: 3 , with ACK: 8
[DEBUG]- Received message type: 2 , with sequence: 9
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Received message type: 2 , with sequence: 10
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3 , with ACK: 9
[DEBUG]- Sent message type: 3 , with ACK: 10
[DEBUG]- Received message type: 2 , with sequence: 11
[DEBUG]- Writing file in ../server_storage/test_file_2.txt
[DEBUG]- Sent message type: 3 , with ACK: 11
[INFO]- Keep alive timed out, ending connection
[DEBUG]- Downloader sender thread ending
[DEBUG]- Downloader receiver thread ending
[INFO]- Connection ended

[DEBUG]- Sent message type: 2 , with sequence: 11
[DEBUG]- Received message type: 3 , with ACK: 8
[DEBUG]- Received message type: 3 , with ACK: 8
[DEBUG]- Received message type: 3 , with ACK: 8
[DEBUG]- Resending Go Back N window
[DEBUG]- Sent message type: 2 , with sequence: 9
[DEBUG]- Sent message type: 2 , with sequence: 10
[DEBUG]- Sent message type: 2 , with sequence: 11
[DEBUG]- Received message type: 3 , with ACK: 9
[DEBUG]- Received message type: 3 , with ACK: 10
[DEBUG]- Resending Go Back N window
[DEBUG]- Sent message type: 2 , with sequence: 11
[DEBUG]- Received message type: 3 , with ACK: 11
[INFO]- File sent
[DEBUG]- Uploader sender thread ending
[DEBUG]- Uploader receiver thread ending
[INFO]- Connection ended
fabio@fabio-ThinkPad:~/universidad/Redes/TP1/src$

```

Figura 16: Fin de conexión upload usando Go Back N

Este caso se trata de un Upload usando Go Back N. Se pueden ver algunas similitudes con el caso anterior así como también algunas diferencias:

- La principal diferencia se ve en el funcionamiento del protocolo, que en este caso si se pierde un paquete, el servidor al recibir los siguientes paquetes los rechaza y envía acks repetidos del último paquete antes del perdido, por lo que el cliente (luego de un Timeout) vuelve a enviar toda la ventana de Sends.

### 4.3. Pruebas temporales

|      |     | 1,1 MB        |               |              | 5 MB          |               |              | 17,1 MB       |               |              |
|------|-----|---------------|---------------|--------------|---------------|---------------|--------------|---------------|---------------|--------------|
|      |     | $t_{real}[s]$ | $t_{user}[s]$ | $t_{sys}[s]$ | $t_{real}[s]$ | $t_{user}[s]$ | $t_{sys}[s]$ | $t_{real}[s]$ | $t_{user}[s]$ | $t_{sys}[s]$ |
| 0 %  | S&W | 0,343         | 0,125         | 0,054        | 1,137         | 0,485         | 0,242        | 3,540         | 1,528         | 0,867        |
|      | GBN | 0,408         | 0,177         | 0,148        | 1,383         | 0,825         | 0,546        | 4,497         | 2,844         | 1,855        |
| 10 % | S&W | 24,800        | 0,210         | 0,080        | 114,552       | 0,844         | 0,421        | 401,407       | 3,050         | 1,342        |
|      | GBN | 11,841        | 0,289         | 0,159        | 62,745        | 1,282         | 0,762        | 207,705       | 4,426         | 2,754        |

Tabla 1: Mediciones temporales ejecutando `time` antes de correr al cliente, para archivos de distintos tamaños, distintos protocolos de comunicación y distintos porcentajes de pérdida de paquetes.

Analizando la Tabla 1, se ve que hay una gran diferencia entre el tiempo que lleva correr un programa ( $t_{real}$ ) con y sin pérdida de paquetes. Analizando el tiempo de sistema ( $t_{sys}$ ) se ve que al tener pérdida de paquetes el programa no estaba constantemente procesando, por lo que se deduce que la mayoría del tiempo estaba a la espera del Senack, con lo que una manera de reducir el tiempo de ejecución sería reducir el *Timeout*.

Sin pérdida de paquetes, Stop & Wait es más rápido en todos los casos, probablemente debido a que sin pérdida de paquetes y con un RTT prácticamente nulo, la mayor parte del tiempo se está procesando. Como la lógica de Stop & Wait es más simple, tiene sentido que bajo estas condiciones ideales haya tardado un poco menos.

Con pérdida de paquetes al contrario, Go Back N tarda entre el 46 y 54 % de lo que hubiera tardado con Stop & Wait. Este resultado es lo esperado, siendo que en Stop & Wait la pérdida de un Send o Senack implican esperar un *Timeout*, mientras que en Go Back N solo la pérdida de un Send implica esperar el *Timeout*, ¡la mitad de los escenarios!

Puede apreciarse cierta linealidad entre el tiempo que tarda en transmitirse un archivo y su tamaño. Esto implicaría que las velocidades de transferencia sean bastante parecidas entre todos tiempos reales del mismo protocolo con misma pérdida de paquetes, lo cual se cumple. Principalmente con pérdida de paquetes, sin esta pareciera que cuanto más grande es el archivo mayor es la velocidad, lo que tiene sentido siendo que el tiempo dedicado a configurar al programa Client, procesar sus argumentos y realizar el *handshake* es mucho más considerable para archivos chicos.

|      |     | 1,1 MB    | 5 MB      | 17,1 MB   |
|------|-----|-----------|-----------|-----------|
|      |     | $v[MB/s]$ | $v[MB/s]$ | $v[MB/s]$ |
| 0 %  | S&W | 3,210     | 4,400     | 4,830     |
|      | GBN | 2,700     | 3,620     | 3,800     |
| 10 % | S&W | 0,044     | 0,044     | 0,043     |
|      | GBN | 0,093     | 0,080     | 0,082     |

Tabla 2: Velocidad de transferencia para distintos tipos de transferencia, protocolos y pérdida de paquetes.

#### 4.4. Pruebas de verificación de datos

Se utilizó Wireshark para capturar el envío de paquetes y verificar que sus estructuras y datos concuerdan con lo planificado por los integrantes del grupo. Además, se utilizaron mensajes de debugging integrados en el código para verificar también lo antes mencionado.

Para verificar su correcto funcionamiento con pérdida de paquetes se hizo uso de la herramienta Comcast. Por último, se enviaron archivos con diferentes extensiones (txt, pdf, jpg) para corroborar (con estos 2 últimos) que la pérdida de paquetes no afectara la comunicación, ya que de ser así el archivo quedaría corrupto sin posibilidad de abrirlo y visualizarlo.

|      | Time          | Source    | Destination | Protocol | Length | Info                   | SourcePort | DestPort |
|------|---------------|-----------|-------------|----------|--------|------------------------|------------|----------|
| 6555 | 194.153270570 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6556 | 194.153523055 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6557 | 194.153735809 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6558 | 194.153870111 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6559 | 194.154093354 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6560 | 194.154105676 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6561 | 194.354578109 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6562 | 194.354813744 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6563 | 194.355088899 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6564 | 194.355238972 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6565 | 194.355393084 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6566 | 194.355525367 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6567 | 194.455838331 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6568 | 194.456087936 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6569 | 194.556433560 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6570 | 194.556696845 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6571 | 194.556995179 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6572 | 194.557053482 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6573 | 194.557208325 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6574 | 194.557346108 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6575 | 194.657664770 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6576 | 194.657918125 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6577 | 194.658120969 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6578 | 194.658262872 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6579 | 194.658393764 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6580 | 194.658535397 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6581 | 194.658683230 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6582 | 194.658831252 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6583 | 194.658975845 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6584 | 194.659120448 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6585 | 194.659292501 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |
| 6586 | 194.659414624 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 53632 → 36609 Len=5    | 53632      | 36609    |
| 6587 | 194.659558386 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 36609 → 53632 Len=1005 | 36609      | 53632    |

Figura 17: Intercambio de Sends y Senacks con Stop & Wait. Se ve que los Sends tienen tamaño 1005 (fixed header + número de secuencia + payload) y los Senacks 5 (fixed header + ack number).

|      | Time         | Source    | Destination | Protocol | Length | Info                   | SourcePort | DestPort |
|------|--------------|-----------|-------------|----------|--------|------------------------|------------|----------|
| 5670 | 27.734696482 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5671 | 27.734803534 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5672 | 27.835245666 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5673 | 27.835306397 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5674 | 27.835378778 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5675 | 27.835509181 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5676 | 27.835662744 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5677 | 27.835786386 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5678 | 27.836111332 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5679 | 27.936502252 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5680 | 27.936560563 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5681 | 27.936597974 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5682 | 27.936634765 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5683 | 27.936875809 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5684 | 27.936894340 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5685 | 27.937013522 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5686 | 27.937126974 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5687 | 27.937451760 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5688 | 27.937498581 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5689 | 27.937723335 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5690 | 27.937790456 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5691 | 27.937938019 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5692 | 27.938432308 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5693 | 27.938543231 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5694 | 27.938676413 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5695 | 27.938781515 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5696 | 28.039198057 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5697 | 28.039246858 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5698 | 28.039274439 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5699 | 28.039300349 | 127.0.0.1 | 127.0.0.1   | UDP      | 1047   | 40165 → 56488 Len=1005 | 40165      | 56488    |
| 5700 | 28.039549864 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5701 | 28.039565754 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |
| 5702 | 28.039711237 | 127.0.0.1 | 127.0.0.1   | UDP      | 47     | 56488 → 40165 Len=5    | 56488      | 40165    |

Figura 18: Intercambio de Sends y Senacks con go Back N.

Como detalle adicional, el tamaño del paquete entero está compuesto por el tamaño del mensaje de aplicación enviado (1005 para Send o 5 para Senack), el header de UDP (8 bytes), el header de IPV4 (20 bytes) y el header de Ethernet (14 bytes).

Una verificación utilizada que era particularmente clara y visualmente agradable era la de enviar una imagen, abrirla con un programa que actualice la imagen guardada constantemente (como VS Code) y ver cómo de a poco la imagen recibida va “cargando”.

## 5. Preguntas a responder

### 5.1. Describa la arquitectura Cliente-Servidor

La arquitectura cliente servidor es una arquitectura de software basada en la presencia de un proveedor de servicios (servidor) y quienes los solicitan (clientes). Estos servicios suelen ser el envío o recepción de información los cuales ocurren a través de una API que tanto servidor como cliente conocen.

Idealmente se busca que un servidor pueda procesar lo recibido de varios clientes al mismo tiempo. Para lograr esto, una estrategia es la presencia de un socket que escuche nuevas solicitudes de conexión de nuevos clientes. Al recibir una solicitud de conexión, en caso que el servidor pueda aceptarla, se abre un nuevo thread y un nuevo socket para el procesamiento de esta conexión. Otra estrategia es tener un *pool* de conexiones y si hay lugar, se acepta esa conexión. Una vez finalizadas las peticiones por parte del cliente (o que ocurre algún evento como un *timeout*), se finaliza la conexión y se liberan los recursos de esta.

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

Los protocolos de capa de aplicación se utilizan para intercambiar los datos entre los procesos que se ejecutan entre hosts. Por lo tanto es crucial que estos compartan una serie de reglas para garantizar una correcta comunicación.

A la función principal de proveer comunicación entre procesos de distintos hosts, se le pueden agregar funciones secundarias como proveer privacidad, envíos confiables, detección y corrección de errores. Ejemplos de protocolos de capa de aplicación utilizados son HTTP, HTTPS, FTP, SMTP, POP3, DNS, los desarrollados en este TP, entre otros.

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

Como ya se vio en la sección “Implementación”, para este trabajo práctico se desarrollaron dos protocolos de transferencias de archivos, uno del tipo Stop & Wait y otro Go Back N, utilizando como protocolo de capa de transporte UDP.

Para la comunicación entre el cliente y el servidor se utilizan cinco tipos de mensajes: Initiate, Inack, Established, Send y Senack.

Primero se realiza un *handshake*, iniciado por el cliente quien envía un Initiate al socket que escucha nuevas conexiones del servidor. Este Initiate contiene el tipo de transferencia a realizar (*upload* o *download*) y el protocolo a utilizar (Stop & Wait o Go Back N). Una vez recibido, en caso que el servidor acepte la conexión creará una nueva conexión con un nuevo socket exclusivo para este cliente, por el cual enviará al cliente un Inack, con el mismo tipo de transferencia y protocolo, mostrando aceptar la conexión. El cliente al recibir este Inack desde un nuevo puerto sabe a qué puerto enviar sus mensajes, así que envía un Established con el nombre del archivo a descargar/subir dando por establecida la conexión. El servidor considera establecida la conexión una vez que llega ese Established.

Una vez que la conexión queda establecida, el Uploader empieza a enviar mensajes tipo Send, con el payload del archivo y el Downloader responde a estos con Senacks. La utilización de números de secuencia permite saber si se perdieron archivos o si llegaron desordenados. En caso de haber sido corruptos, la capa de transporte, este caso que utiliza el protocolo UDP, detecta si fueron corrompidos y los descarta, para la capa de aplicación esto sería equivalente a haber perdido el paquete.

El algoritmo de envío y recepción de Sends y Senacks no depende de cuál es el cliente y cuál el servidor, solo importa cuál es el Uploader y cuál el Downloader. Y dicho algoritmo quedará especificado dependiendo de si se utiliza Stop & Wait o Go Back N. Para ver en mayor detalle la implementación de los protocolos, ver la sección **3. Implementación**.

#### Stop & Wait

En Uploader envía un Send, que contiene el primer trozo del archivo a enviar, con un número de secuencia  $n$ , el Downloader lo recibe y responde con un Senack, que tiene un *acknowledge number* (también llamado *ack*) que es igual al número de secuencia recibido. Cuando el Uploader recibe este Senack envía el próximo trozo de archivo en otro Send cuyo número de secuencia será  $n + 1$ . Este proceso se repite hasta haber enviado el archivo entero y haber recibido todos los Senacks. Una vez recibido el Senack del Send con el último trozo del archivo, el Uploader cierra su parte de la conexión, y el Downloader eventualmente cerrará la suya cuando ocurra el *Idle Timeout*.

En caso de perderse paquetes Send o Senack, luego de cierto tiempo ocurrirá un *Timeout* en el Sender, el cual reenviará el paquete Send cuyo Senack no ha llegado. En el Downloader, en caso de recibir Sends repetidos o desordenados (lo cual es improbable pero no imposible en Stop & Wait) estos son descartados.

## Go Back N

A diferencia del protocolo anterior, en este el Uploader envía secuencialmente varios Send hasta tener una cierta cantidad (ventana) de paquetes cuyo Senack todavía no llegó. Cuando llega el Senack del Send enviado más antiguo cuyo Senack todavía no había llegado, se procede a enviar el siguiente Send. Este proceso se repite hasta que el Uploader haya recibido el Senack del Send con el último trozo del archivo.

En caso de perderse un Send, el Downloader empezará a recibir Sends desordenados, los cuales descartará y responderá con Senacks cuyo *ack* será igual al último Send recibido de manera correcta y ordenada.

Al no recibir uno de los Senacks, el servidor eventualmente tendrá un *timeout* luego del cual reenviará toda la ventana de Sends cuyos Senack todavía no recibió.

En caso de perderse un Senack no pasa nada, siendo que el servidor eventualmente recibirá los Senacks de Sends posteriores, y estos Senacks implicarán la correcta recepción de todos los Send cuyos números de secuencia son menores al *ack* de los Senacks.

Si se pierde el último Senack eventualmente ocurrirá un *timeout* y se reenviarán los Sends.

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

TCP proporciona un servicio orientado a la conexión (full duplex) y confiable para la transmisión de datos. Garantiza la entrega ordenada (utilizando números de secuencia) y sin errores (utilizando un *checksum*) de los datos, maneja la retransmisión de paquetes perdidos (utilizando *timeouts*, controla el flujo de datos para evitar la congestión de la red (utilizando una ventana de envío variable) y proporciona una confirmación de entrega (*ack*) para cada segmento recibido. Se dice que es orientado a la conexión porque establece una conexión antes de la transferencia de datos a partir de un *3 way handshake* y la termina después de la transmisión. Se recomienda usar este protocolo cuando se requiere una transmisión confiable de los datos. La mayoría de las aplicaciones envían sus datos sobre este protocolo hoy en día. Debido a la gran cantidad de servicios que provee, el header de TCP es de al menos 20 bytes, y su tamaño puede ser de hasta 60 bytes debido a la presencia de campos opcionales.

UDP por otro lado, no garantiza una confiable transmisión de datos (*best effort*), simplemente envía los paquetes sin garantizar su entrega o secuencia, tampoco realiza ajustes automáticos de velocidad de transmisión ni maneja la congestión de la red. Tiene un *checksum* para la detección de errores. Sin embargo, esto hace que sea más rápido que TCP y que su header sea 8 de bytes, por lo que se recomienda usarlo cuando la latencia es crítica y la pérdida ocasional de datos no es un problema, como en transmisiones de audio y video en tiempo real, VoIP, videojuegos, etc. Además de las aplicaciones anteriores, DNS, DHCP y SNMP utilizan UDP. En caso de querer agregar servicios a la transmisión de datos utilizando UDP, es posible agregarlos en la capa de aplicación, como se hizo en este TP.

## 6. Dificultades encontradas

Las mayores dificultades encontradas durante la realización de este trabajo fueron:

- Principalmente, que el TP fue realizado por 2 personas (siendo originalmente 3 integrantes) cuando fue pensado para grupos de 5 integrantes. Toda esa carga de trabajo resultó desproporcionada, siendo que un integrante cursa 4 materias y el otro cursa 3 y trabaja.
- La implementación de concurrencia para el manejo de múltiples clientes con su sincronización a través del uso de threads y sockets.
- El desarrollo del sistema de mensajes entre los clientes y el servidor.
- La unificación de la lógica Upload y Download, para que sea independiente de si se realiza en el servidor o el cliente.
- La correcta utilización de *timers* y *timeouts* para el reenvío de paquetes, reenvío de ventana y finalización de la conexión.

## 7. Conclusiones

- Gracias a este trabajo se pudieron adquirir conocimientos más profundos acerca de los diferentes protocolos y sus capas de aplicación así como también poner en práctica aquellos adquiridos de forma teórica. Todo esto desembocó en haber logrado desarrollar protocolos RDT basados en UDP para la transferencia de datos entre un servidor y uno o varios clientes.
- Resultó interesante el análisis con Wireshark de los paquetes enviados y ver las partes del archivo siendo enviadas a medida que se iban recibiendo los *acks*. También la correcta implementación de las medidas al perder/repetir paquetes.
- Debido a la poca cantidad de integrantes de este equipo, hubiese sido imposible entregar un programa funcionando si no se hubiesen seguido buenas prácticas de diseño como muchos refactors y testeo continuo.
- Es posible agregar servicios no provistos por la capa de transporte en la capa de aplicación en caso de ser necesario.
- Cuanto mayor sea la tasa de pérdida de paquetes, mayor será el beneficio de utilizar Go Back N por sobre Stop & Wait, aunque en un contexto sin pérdidas y de muy poco RTT conviene más Stop & Wait. La decisión final de cuál usar debería hacerse considerando la tasa de pérdidas y si vale la pena el esfuerzo extra de implementar el protocolo más complejo Go Back N.