FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Introducción a los Sistemas Distribuidos
(75.43)
Trabajo Práctico 1 - File Transfer
Grupo: 7
Fecha de entrega: 04/10/2022

Integrantes:

Nombre y Apellido	Padrón
Joaquin Hojman	102264
Facundo Mastricchio	100874
Santiago Tissoni	103856
Francisco Vazquez	104128
Andres Zambrano	105500

Índice

Introducción **2**

Hipótesis y supuestos **2**

Implementación **3**

Protocolo Stop & Wait 4

Protocolo Go-Back-N 8

Pruebas **11**

Análisis **15**

Preguntas a responder **16**

Dificultades encontradas **19**

Conclusiones **19**

Introducción

El presente trabajo práctico tiene como objetivo la creación de una aplicación de red. Para tal finalidad, será necesario comprender cómo se comunican los procesos a través de la red, y cuál es el modelo de servicio que la capa de transporte le ofrece a la capa de aplicación. Además, para poder lograr el objetivo planteado, se aprenderá el uso de la interfaz de sockets y los principios básicos de la transferencia de datos confiable (del inglés *Reliable Data Transfer, RDT*).

En este trabajo práctico buscaremos comprender y poner en práctica los conceptos y herramientas necesarias para la implementación de un protocolo RDT. Para lograr este objetivo, desarrollaremos una aplicación de arquitectura cliente-servidor que implemente la funcionalidad de transferencia de archivos mediante las siguientes operaciones:

- **UPLOAD**: Transferencia de un archivo del cliente hacia el servidor.
- **DOWNLOAD**: Transferencia de un archivo del servidor hacia el cliente.

Dadas las diferentes operaciones que pueden realizarse entre el cliente y el servidor, requeriremos el diseño e implementación de un protocolo de aplicación básico que especifique los mensajes intercambiados entre los distintos procesos.

La comunicación entre los procesos será implementada utilizando **UDP como protocolo de capa de transporte**. Luego para lograr una transferencia confiable al utilizar el protocolo UDP, implementaremos una versión utilizando el protocolo **Stop & Wa**it y otra versión utilizando el protocolo **Go-Back-N**.

El servidor procesara de forma concurrente la transferencia de archivos de carga y descarga con múltiples clientes.

Hipótesis y supuestos

- Asumimos que el servidor posee una estructura escalable que le permite recibir y guardar múltiples archivos.
- No hay restricciones en el tamaño de los archivos que se le pueden mandar al servidor. Sin embargo, existe un tamaño máximo contemplado en el protocolo para el tamaño de nuestro paquete.
- En caso de que el servidor reciba un archivo con el mismo nombre exacto que un archivo que ya posee, el antiguo será sobrescrito.
- Confiamos en UDP para que valide el checksum. [RFC 768]
- En caso de que el cliente intente descargar un archivo que no exista en el servidor, el cliente hace un timeout.

Implementación

La manera de levantar el servidor y el cliente están indicadas en el archivo *README.md*. Así mismo, se cuenta con los comandos necesarios para simular una pérdida de paquete con *comcast*.

Para levantar una instancia del cliente es necesario indicar si se va a realizar un upload o un download de un archivo. Además, al levantar tanto el servidor como el cliente se debe indicar que protocolo usaremos en la comunicación: *stop and wait o Go-Back-N*.

Con esta información, el usuario y el servidor crearán una instancia de un objeto **StopAndWaitUploaderManager** (si se va a hacer upload con *stop and wait*), **StopAndWaitDownloaderManager** (si va a hacer download con *stop and wait*), o **GoBackNManager**. Estos objetos servirán como interfaz para que se pueda llevar a cabo la transferencia.

El servidor y el cliente no recibirán los mensajes exactamente igual. El usuario escucha en su socket y recibe los mensajes (o envía) por allí. Esto se debe a que el usuario trabaja con un único archivo, ya sea para enviar o recibir. El servidor, por su parte, posee un único socket pero puede estar comunicándose con muchos clientes a la vez. Es por eso que el servidor escucha mensajes en su socket, y cuando recibe uno, realiza una de las siguientes acciones:

- En caso que el mensaje tenga el **bit SYN** prendido, creará un nuevo thread (llamado *ClientWorker*, le asignará una cola de mensajes bloqueante) y lo guardará en un diccionario con el address (formado por el par host, puerto) del cliente que envió el paquete.
- En caso de que no tenga el **bit SYN** prendido, es decir no es un usuario nuevo, usará el address del cliente que envió el paquete para buscar al thread que se encarga de esa conexión, y guardará en su cola de mensajes el paquete recibido para que el thread pueda procesarlo. Luego seguirá escuchando paquetes.

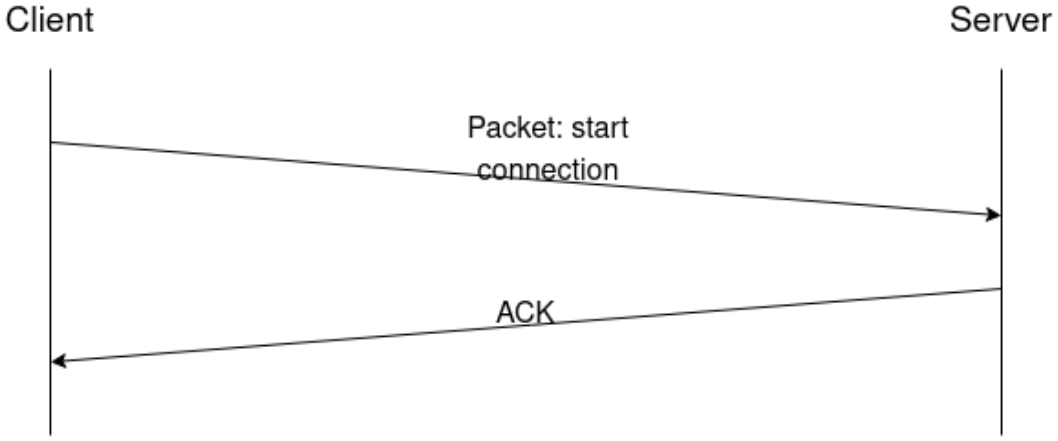
El thread de una conexión esperará a tener mensajes en su cola para procesar, y en base al mensaje que reciba contestara via su propio socket de salida al cliente que tiene asignado. Es decir, el server tiene un único *receive*, en el thread principal, pero son sus threads hijos quienes hacen *send* de los paquetes.

Es importante destacar que cada paquete que se envían el cliente y el servidor consiste en 1024 bytes del archivo (configurable) a transferir + header (más adelante detallaremos la estructura completa del paquete).

Dependiendo de si se eligió utilizar el protocolo de *Stop And Wait* o *Go-Back-N*, el cliente y su thread asociado en el servidor intercambiarán paquetes para transferir el archivo de la siguiente manera:

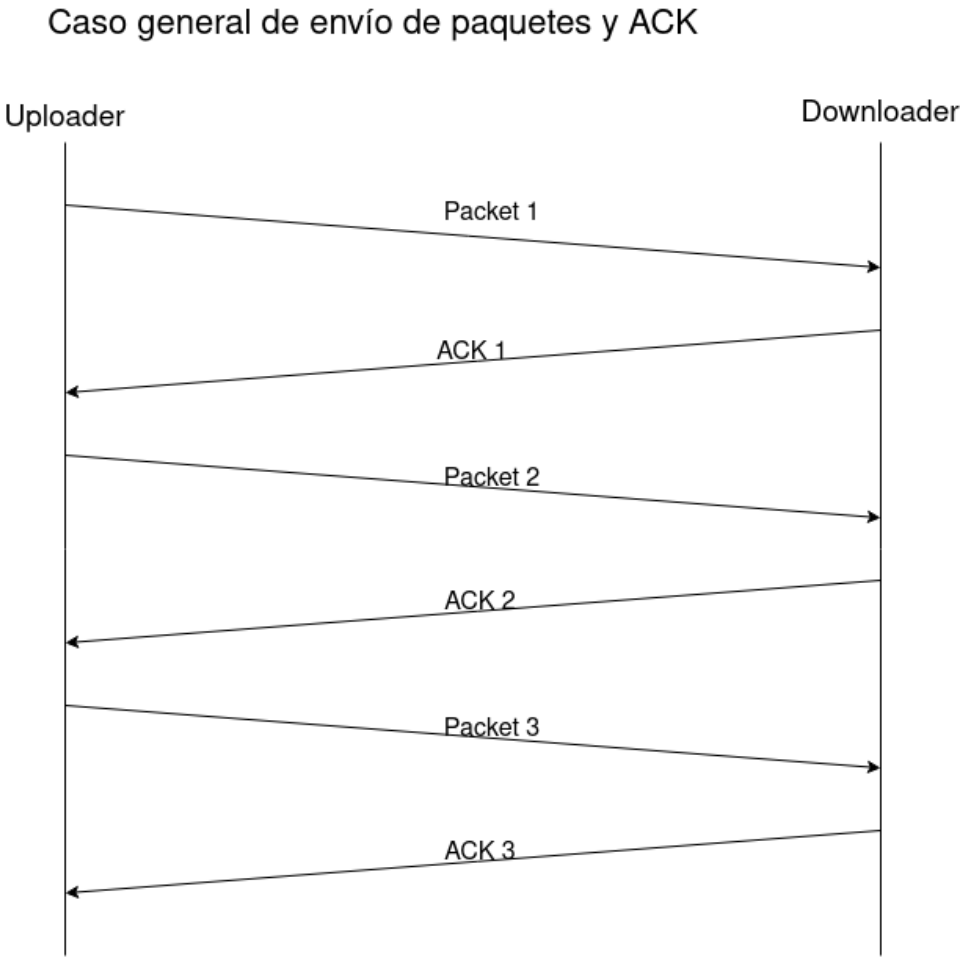
Protocolo Stop & Wait

Primero existe la etapa de inicio de la conexión, en donde el primer paquete lo manda el cliente, el servidor manda el ACK, independientemente de si el cliente pretende descargar o subir un archivo.

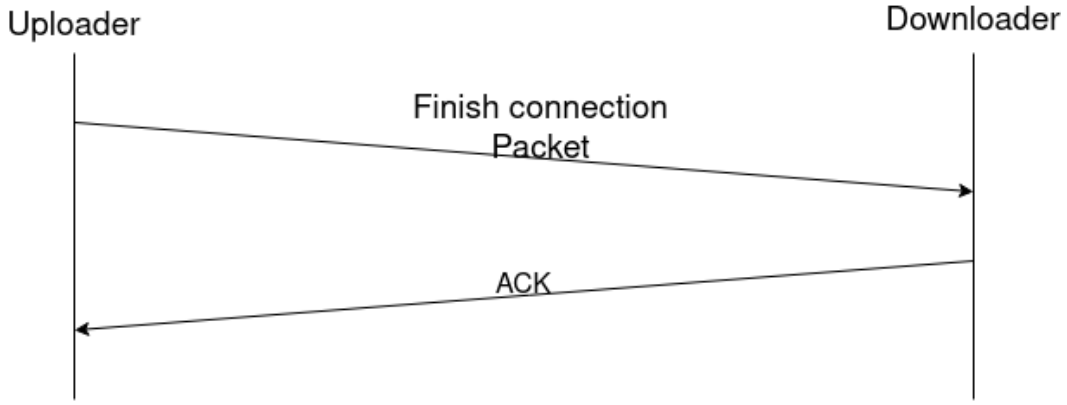


En el caso feliz (sin pérdida de paquetes), este es el primer intercambio. Es acá en donde el cliente sabe que va a descargar un archivo, y el servidor sabe que tiene que mandar un archivo, o viceversa.

A partir de este momento, no se tiene que diferenciar entre cliente y servidor, sino entre *Downloader* y *Uploader*, es decir, entre el programa que baja el archivo y el que sube el archivo respectivamente. El siguiente diagrama corresponde a un flujo normal cuando no hay pérdidas de paquete.

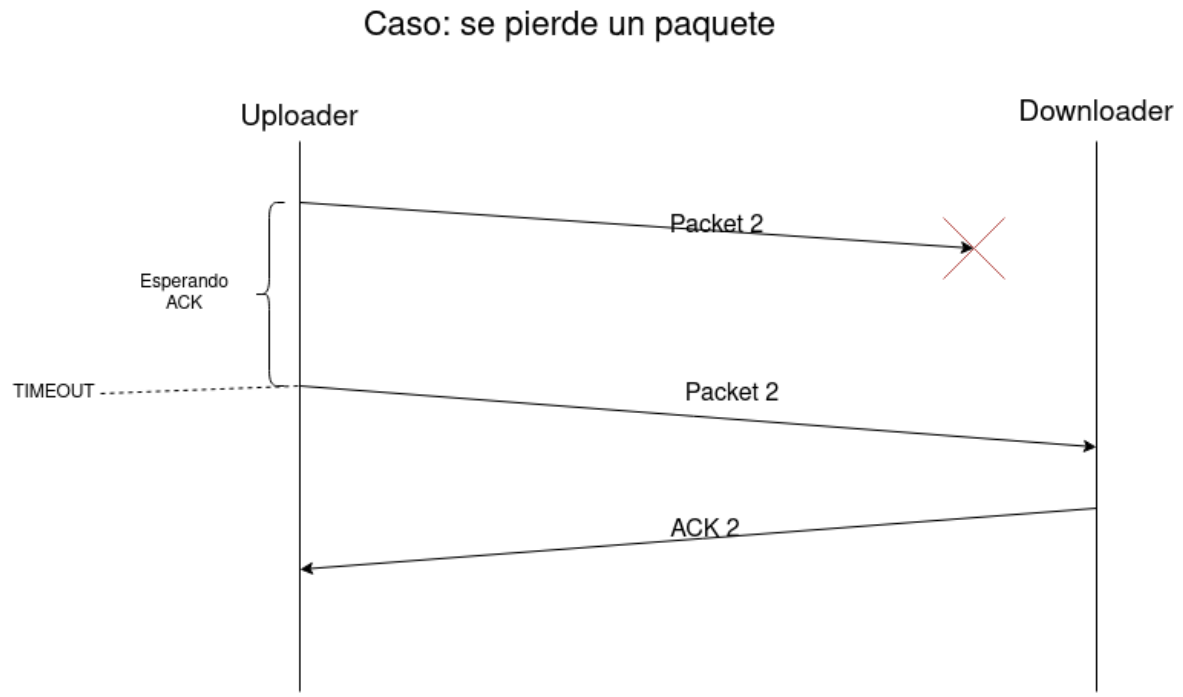


Luego de que el archivo a subir se haya procesado completamente por el lado del *Uploader* (es decir, toda su información ha sido enviada al *Downloader*), el *Uploader* envía un paquete que representa el fin de conexión.

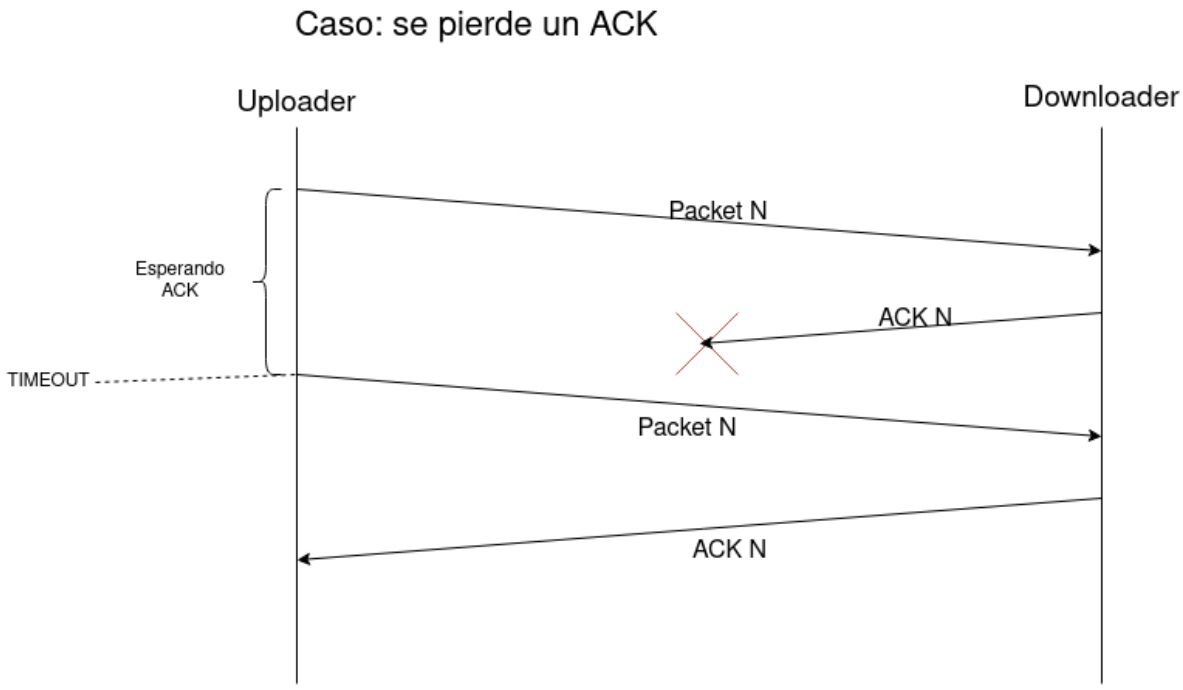


Ahora bien, es conocido que *UDP* no asegura que los paquetes lleguen, ni que lleguen en orden. En un momento de alta congestión, esto podría pasar, y el protocolo *Stop & Wait* propuesto cubre estos escenarios. Se mostrará que, en realidad, la responsabilidad cae sobre todo en el *Uploader* que está subiendo el archivo.

La pregunta base es, ¿qué pasa cuándo se pierde un paquete?



Cuando el *Uploader* manda un paquete con el número N, comienza a esperar su ACK. Si el paquete enviado se perdió, el *Downloader* que está esperando el paquete nunca va a mandar el ACK, por lo que el que sube el archivo va a volver a mandar el paquete.

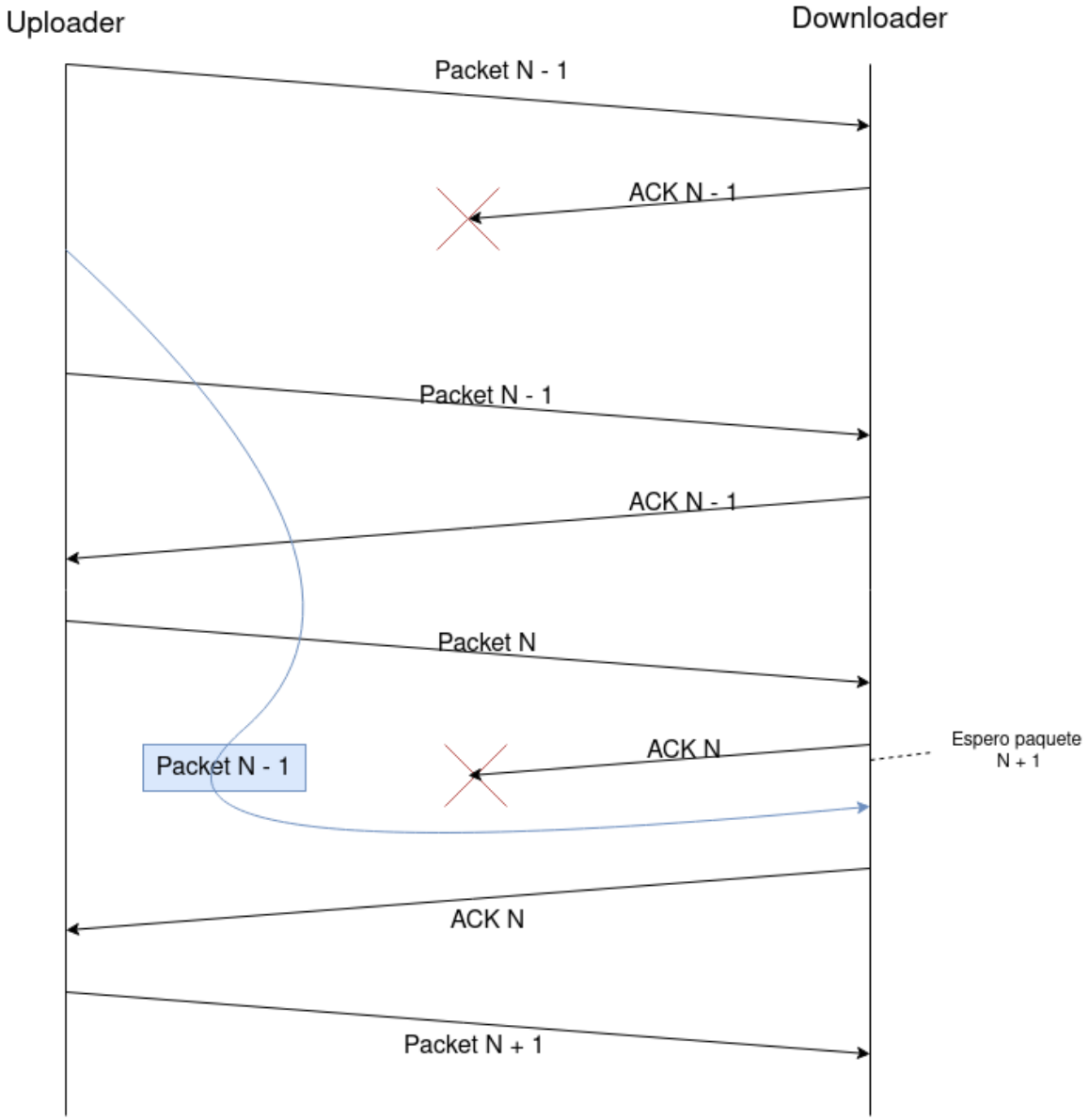


Lo mismo sucede cuando el que espera el paquete con el número N sí lo llega a recibir, pero se pierde su ACK de respuesta. El que manda los paquetes vuelve a mandar el paquete con el número N, y el otro vuelve a mandar su ACK.

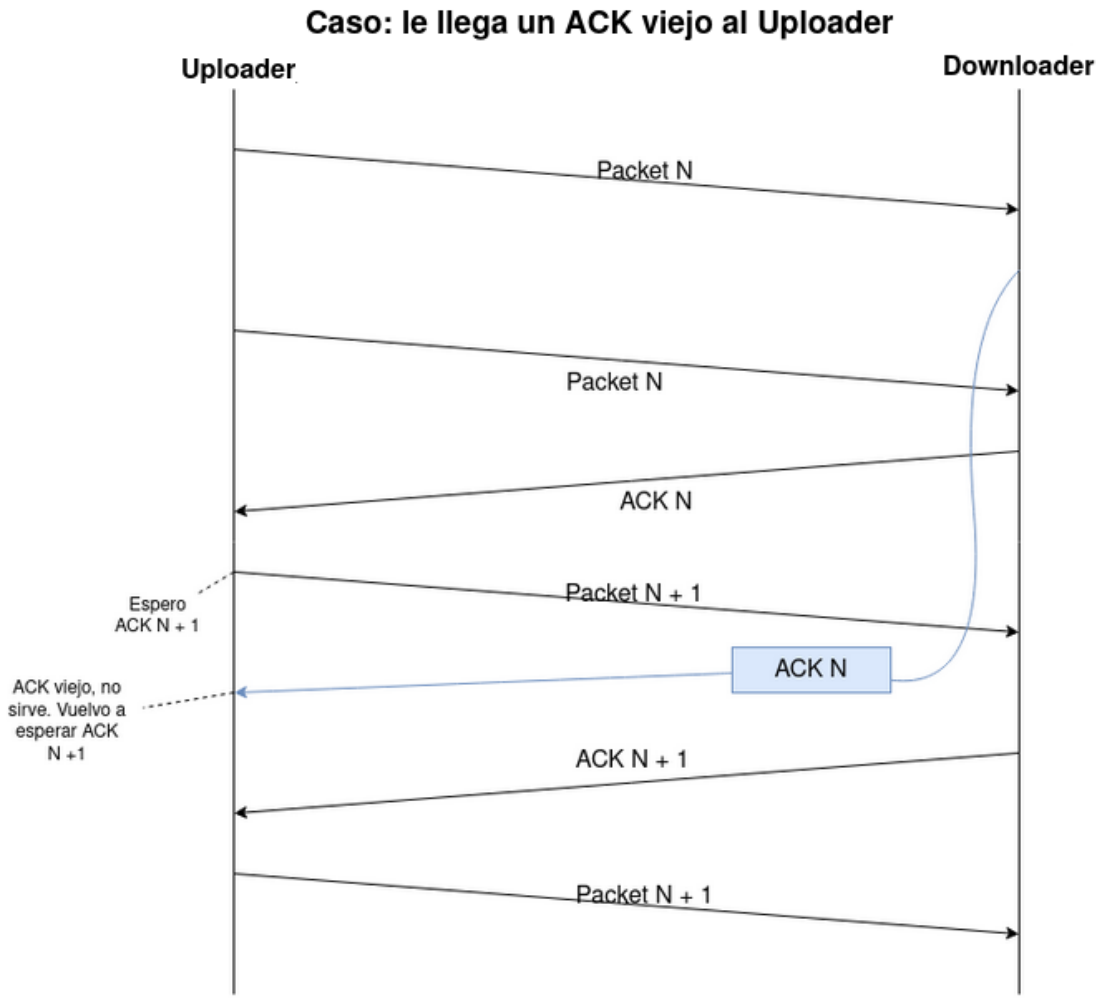
Existe un máximo configurable de reintentos. Si se llegaran a realizar dicha cantidad de reintentos, se cierra el programa si el que sube el archivo es el cliente, o se cierra el hilo que maneja el cliente en el servidor.

Nótese que esto es para un paquete arbitrario N, es decir que así también se manejan los paquetes de inicio y fin de conexión.

Caso: le llega un paquete viejo al downloader



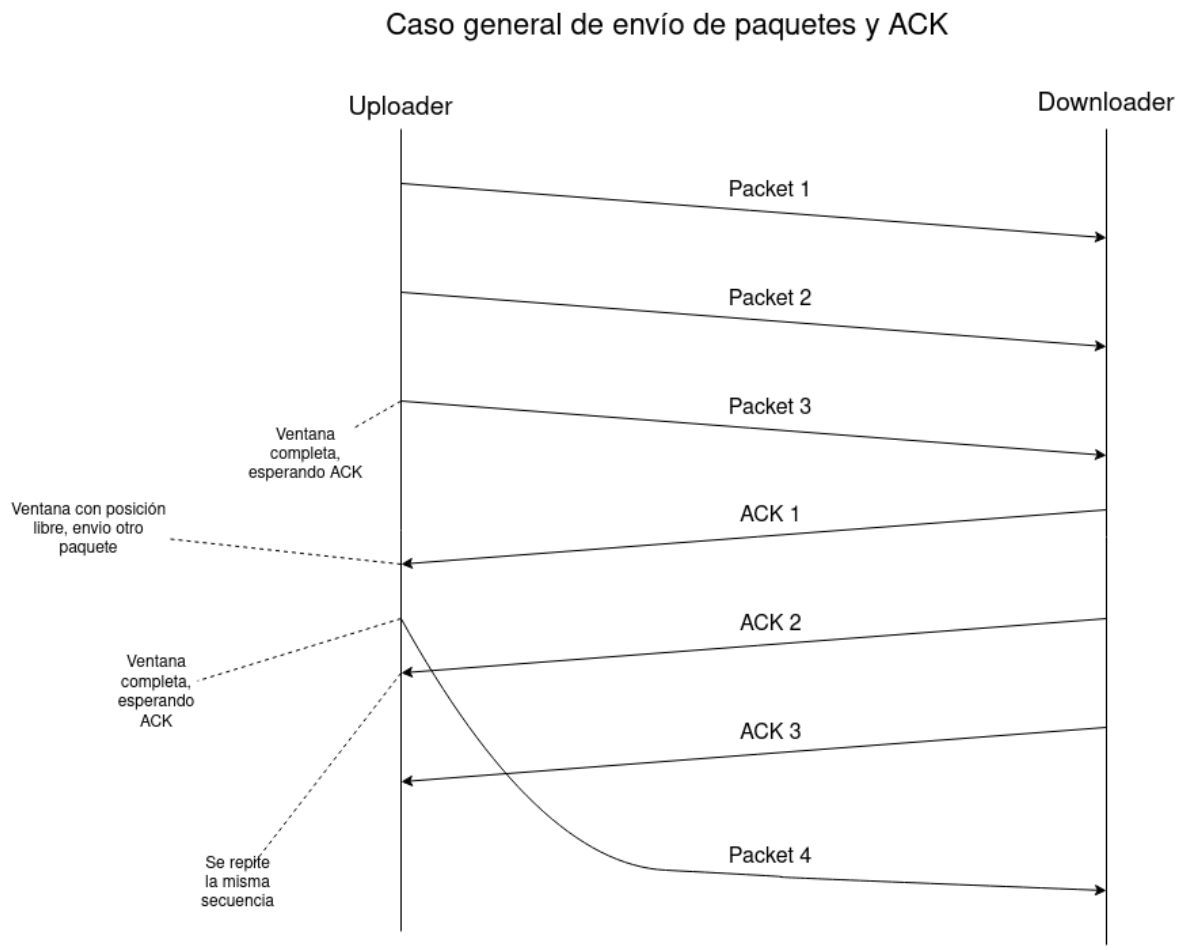
En este escenario, se muestra como el *Downloader* re-envía el ACK del último paquete en orden que recibió cuando recibe un paquete viejo.



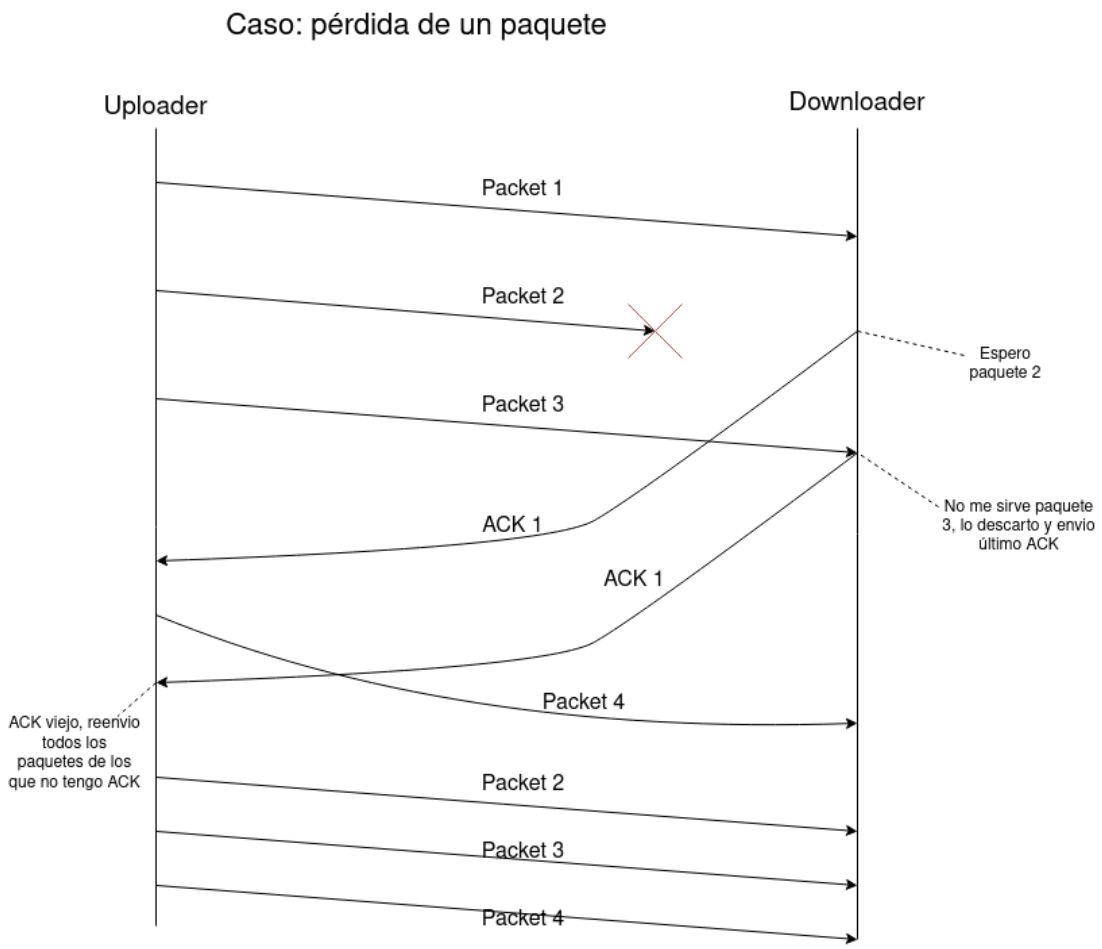
Por último, ¿qué pasa si llega un ACK viejo? El *Uploader* no vuelve a enviar ningún paquete, sino que vuelve a esperar recibir un ACK hasta recibir el ACK esperado o que suceda un timeout.

Protocolo Go-Back-N

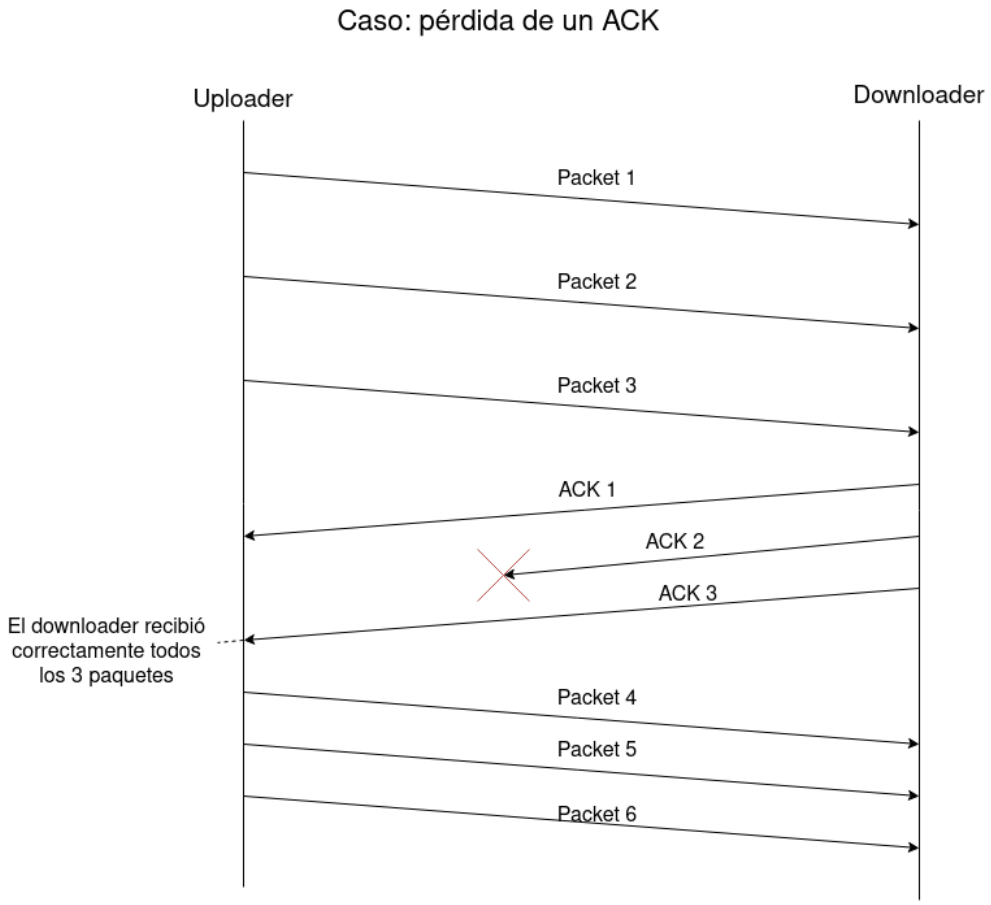
Para el protocolo *Go-Back-N*, la conexión comienza como en el *Stop And Wait*: el cliente le manda un paquete de inicio de conexión al servidor, comunicando si va a realizar una descarga o subida de un archivo. Luego, comienza el caso general que se ve de la siguiente forma (suponiendo que no hay pérdida de paquetes), con una ventana (N configurable) de 3.



Si consideramos la pérdida de paquetes, en el caso de la pérdida de un paquete con datos:



En el caso de la pérdida de un ACK:



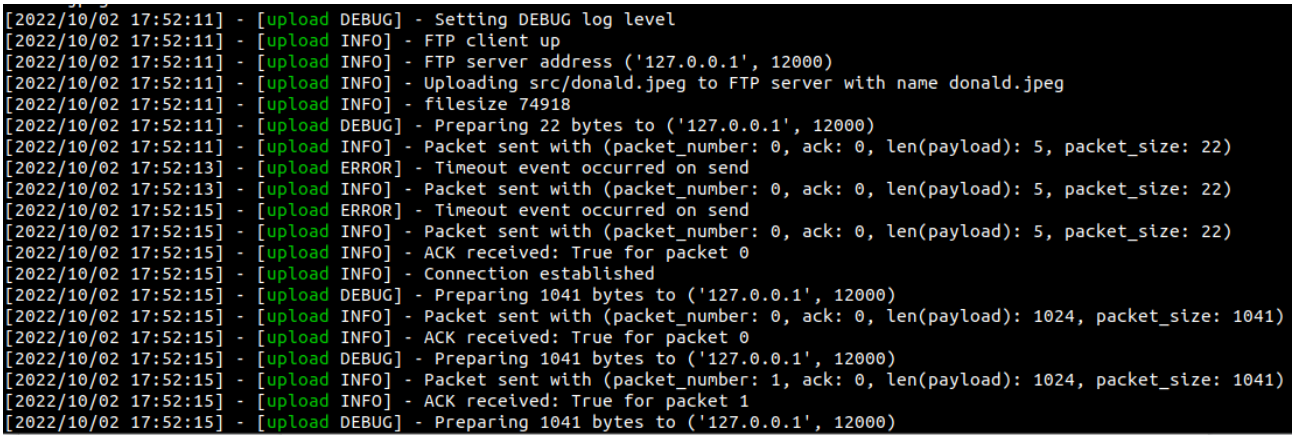
Pruebas

A continuación se utilizará el programa para **enviarle un archivo al servidor utilizando el protocolo Stop And Wait** y veremos algunos logs para verificar el correcto funcionamiento. *Comcast* está activado con pérdida de paquetes del 10%. El timeout por un paquete perdido en este caso será de 2 segundos.

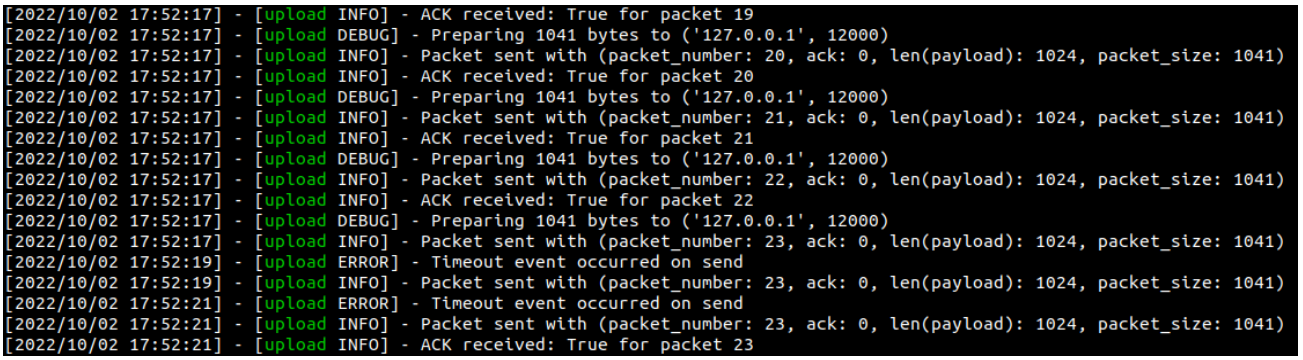
Servidor levantado:

```
joaquinhojman@joaquin-MAX:~/Escritorio/FileTransfer-grupo7$ python3 src/start_server.py -v -H 127.0.0.1 -p 12000 -P saw
[2022/10/02 17:49:53] - [server DEBUG] - Setting DEBUG log level
[2022/10/02 17:49:53] - [server INFO] - Protocol: saw
[2022/10/02 17:49:53] - [server INFO] - FTP server up in port ('127.0.0.1', 12000)
```

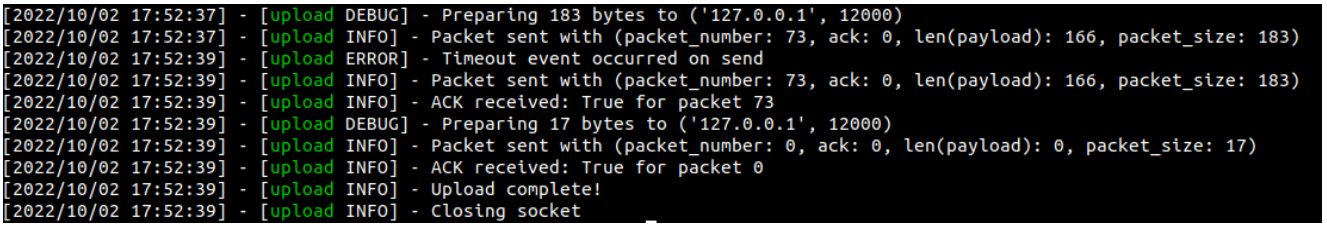
Levantamos el cliente y observamos lo siguiente:



Vemos en esa imagen cuando se establece la conexión, y se dispone a empezar a enviar el archivo. Vemos también como se envía el paquete 1 y se recibe el ACK.



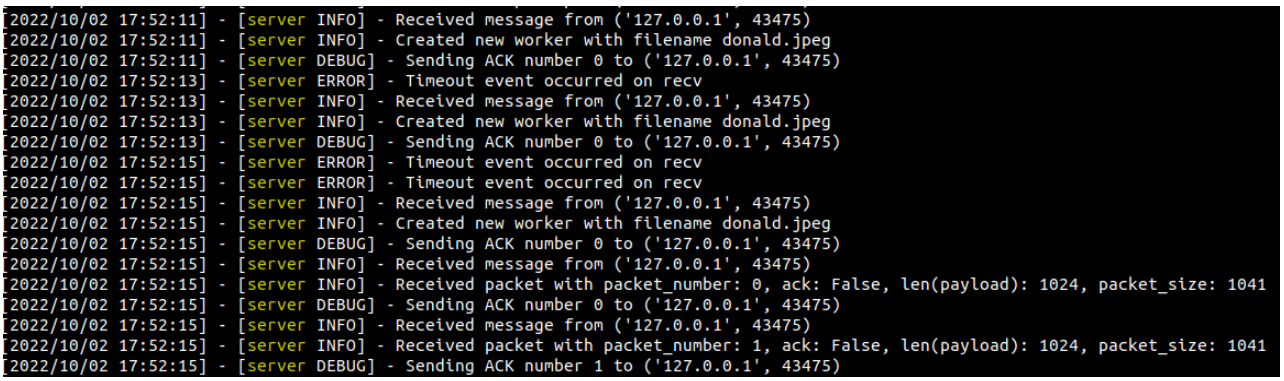
En esta imagen vemos cómo suceden timeouts con el paquete número 23, notamos mirando el tiempo que transcurren 2 segundos (los del timeout) entre cada timeout detectado y cuando se hace el retry.



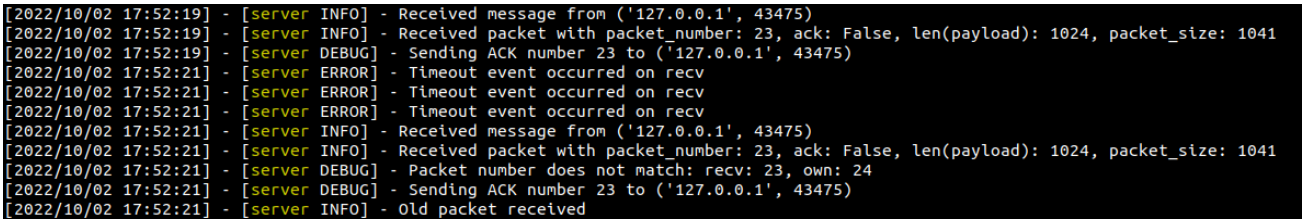
En esta imagen vemos como se envía el último paquete, el número 73, luego se recibe el ACK 73, y finalmente se envía el paquete número 0 que indica el fin de conexión, se recibe el ACK 0 de ese fin de conexión lo que significa que el server entendió que se terminó la transferencia, y se cierra el socket antes de apagar el cliente.

Vemos que el total de la transferencia fue de 28 segundos por lo que podríamos inferir que se perdieron unos 14 paquetes en total para la transferencia (y a razón de 2 segundos por timeout, eso hace 28 segundos).

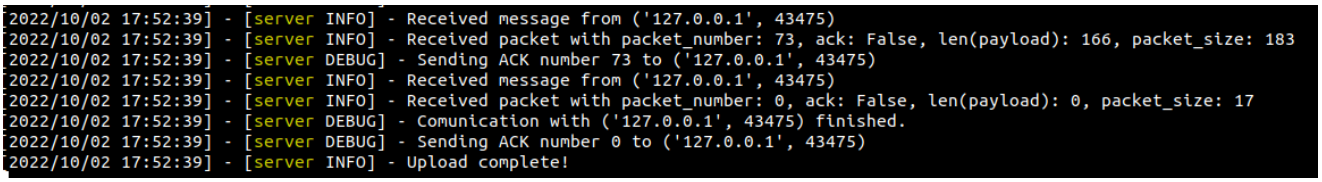
Veamos que sucedió en el server durante este tiempo.



Tenemos mucha información en esta imagen. Vemos como se recibe un nuevo mensaje de una IP y un puerto, como se crea un thread para recibir el archivo que indicaba el paquete que se recibió de esa dirección. Luego vemos como el server envía el ACK 0 (acepta la conexión) y luego comienzan a intercambiar mensajes (paquetes 0 y 1).

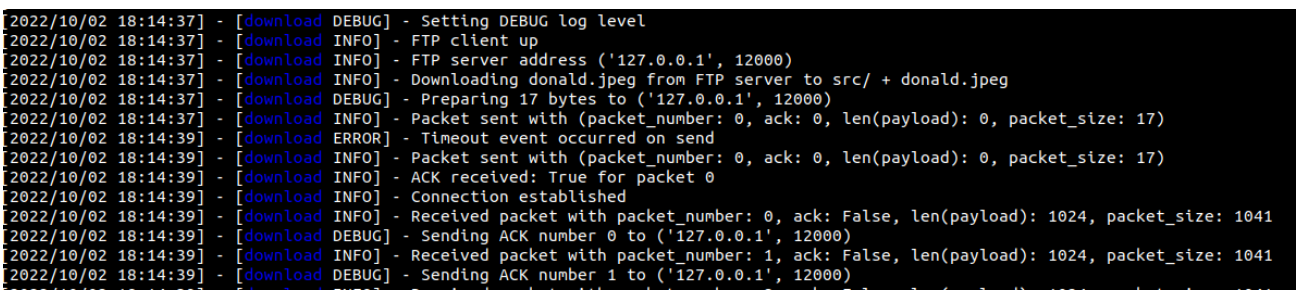


Vemos cómo actúa el servidor durante el envío del paquete 23 (recordemos que era el que había timeouteado en el cliente).



En esta imagen final vemos como el servidor recibe el último paquete y luego el que indica que finalizó la transferencia.

Veamos ahora el caso donde se **descarga un archivo del servidor utilizando el protocolo Stop And Wait.** El caso es prácticamente idéntico al anterior pero se invierten los papeles.



Vemos del lado del cliente como comienzan a intercambiarse paquetes luego de establecer la conexión.

```
2022/10/02 18:15:01] - [download INFO] - Received packet with packet_number: 50, ack: False, len(payload): 1024, packet_size: 1041
2022/10/02 18:15:01] - [download DEBUG] - Sending ACK number 50 to ('127.0.0.1', 12000)
2022/10/02 18:15:03] - [download ERROR] - Timeout event occurred on recv
2022/10/02 18:15:03] - [download INFO] - Received packet with packet_number: 51, ack: False, len(payload): 1024, packet_size: 1041
2022/10/02 18:15:03] - [download DEBUG] - Sending ACK number 51 to ('127.0.0.1', 12000)
```

Vemos ahora un caso de timeout. Vemos que es idéntico al que vimos en el servidor cuando hicimos el upload.

```
[2022/10/02 18:15:03] - [download INFO] - Received packet with packet_number: 73, ack: False, len(payload): 166, packet_size: 183
[2022/10/02 18:15:03] - [download DEBUG] - Sending ACK number 73 to ('127.0.0.1', 12000)
[2022/10/02 18:15:03] - [download INFO] - Received packet with packet_number: 0, ack: False, len(payload): 0, packet_size: 17
[2022/10/02 18:15:03] - [download DEBUG] - Comunication with ('127.0.0.1', 12000) finished.
[2022/10/02 18:15:03] - [download DEBUG] - Sending ACK number 0 to ('127.0.0.1', 12000)
[2022/10/02 18:15:03] - [download INFO] - Download complete for file: donald.jpeg!
[2022/10/02 18:15:03] - [download INFO] - Closing socket
```

Finalizando así la transferencia y el cliente obtiene el archivo pedido.

Veamos ahora el servidor:

```
[2022/10/02 18:14:34] - [server DEBUG] - Setting DEBUG log level
[2022/10/02 18:14:34] - [server INFO] - Protocol: saw
[2022/10/02 18:14:34] - [server INFO] - FTP server up in port ('127.0.0.1', 12000)
[2022/10/02 18:14:39] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server INFO] - Created new worker with filename donald.jpeg
[2022/10/02 18:14:39] - [server DEBUG] - Sending ACK number 0 to ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server DEBUG] - Preparing 1041 bytes to ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server INFO] - Packet sent with (packet_number: 0, ack: 0, len(payload): 1024, packet_size: 1041)
[2022/10/02 18:14:39] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server INFO] - ACK received: True for packet 0
[2022/10/02 18:14:39] - [server DEBUG] - Preparing 1041 bytes to ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server INFO] - Packet sent with (packet_number: 1, ack: 0, len(payload): 1024, packet_size: 1041)
[2022/10/02 18:14:39] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:14:39] - [server INFO] - ACK received: True for packet 1
```

Vemos como se levanta el server, acepta al cliente e inicia la conexión, luego comienza a mandar paquetes con el archivo.

```
[2022/10/02 18:15:01] - [server DEBUG] - Preparing 1041 bytes to ('127.0.0.1', 42910)
[2022/10/02 18:15:01] - [server INFO] - Packet sent with (packet_number: 51, ack: 0, len(payload): 1024, packet_size: 1041)
[2022/10/02 18:15:03] - [server ERROR] - Timeout event occurred on send
[2022/10/02 18:15:03] - [server INFO] - Packet sent with (packet_number: 51, ack: 0, len(payload): 1024, packet_size: 1041)
[2022/10/02 18:15:03] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:15:03] - [server INFO] - ACK received: True for packet 51
```

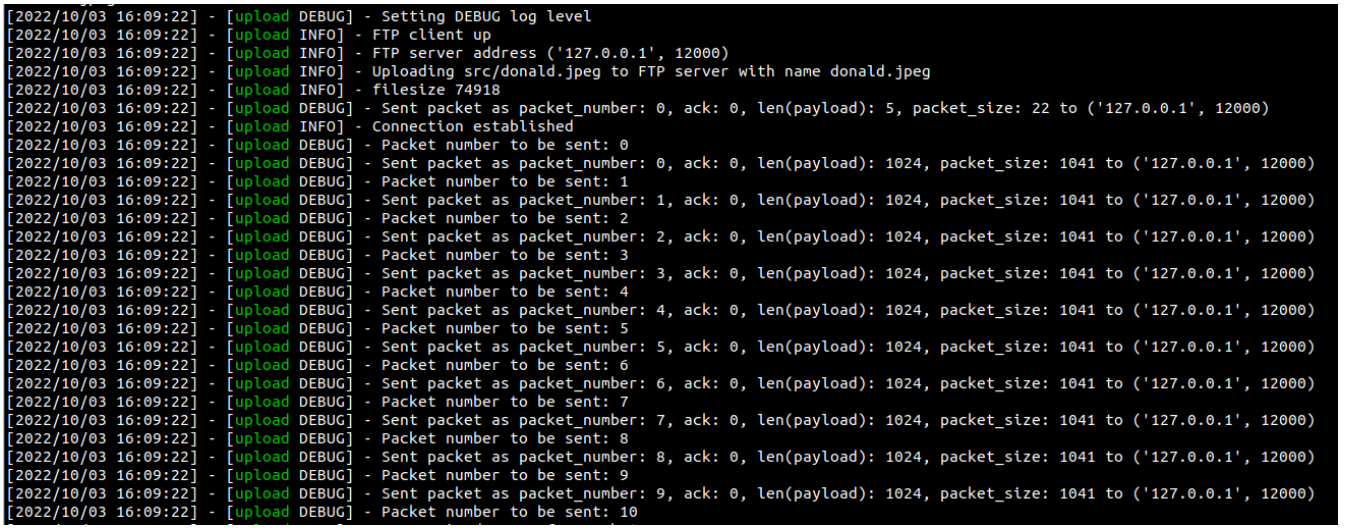
El timeout del paquete 51 que habíamos visto en el cliente.

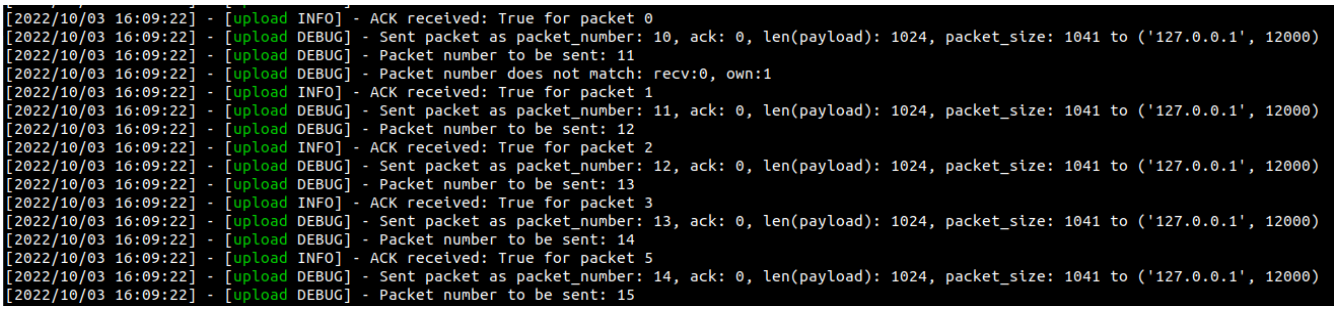
```
[2022/10/02 18:15:03] - [server DEBUG] - Preparing 183 bytes to ('127.0.0.1', 42910)
[2022/10/02 18:15:03] - [server INFO] - Packet sent with (packet_number: 73, ack: 0, len(payload): 166, packet_size: 183)
[2022/10/02 18:15:03] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:15:03] - [server INFO] - ACK received: True for packet 73
[2022/10/02 18:15:03] - [server DEBUG] - Preparing 17 bytes to ('127.0.0.1', 42910)
[2022/10/02 18:15:03] - [server INFO] - Packet sent with (packet_number: 0, ack: 0, len(payload): 0, packet_size: 17)
[2022/10/02 18:15:03] - [server INFO] - Received message from ('127.0.0.1', 42910)
[2022/10/02 18:15:03] - [server INFO] - ACK received: True for packet 0
[2022/10/02 18:15:03] - [server INFO] - Download complete!
```

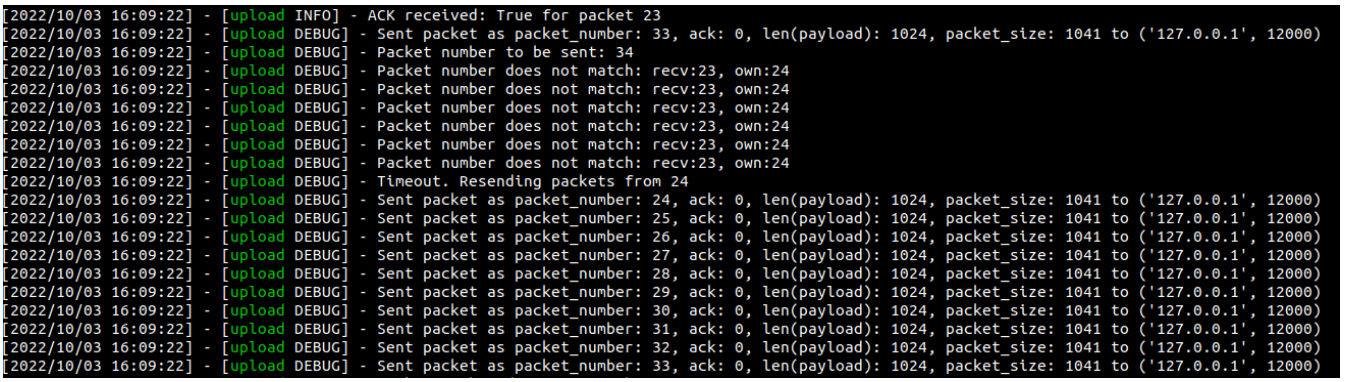
Finaliza de esta manera la transferencia. Como se comentó, es idéntico a la subida pero invertido.

En este caso el tiempo transcurrido fue de 24 segundos por lo que deben haberse perdido 12 paquetes en total.

Vemos ahora brevemente la salida del cliente al hacer **Download con el protocolo Go-Back-N.**







Análisis

Compararemos ahora la performance de la versión *Go-Back-N* del protocolo y la versión *Stop And Wait* utilizando archivos de distintos tamaños y bajo distintas configuraciones de pérdida de paquetes. Timeout = 2 segundos.

Archivo de **3 KB**:

% Pérdida de paquetes	Acción	Tiempo SAW	Tiempo GBN (N=10)
0	Upload	0,053s	0,055s
10	Upload	8,074s	2,067s
0	Download	0,056s	0,055s
10	Download	4,068s	2,062s

Archivo de **100 KB**:

% Pérdida de paquetes	Acción	Tiempo SAW	Tiempo GBN (N=10)
0	Upload	0,104s	0,093s
10	Upload	1m 6,229s	12,144s
0	Download	0,109s	0,079s
10	Download	50,172s	6,121s

Archivo de **1 MB**:

% Pérdida de paquetes	Acción	Tiempo SAW	Tiempo GBN (N=10)
0	Upload	0,507s	0,323s
10	Upload	8m 29,647s	36,684s
0	Download	0,504s	0,353s
10	Download	9m 9,460s	44,700s

Preguntas a responder

1. Describa la arquitectura Cliente-Servidor.

En una arquitectura cliente-servidor existe un host siempre activo, denominado servidor, que da servicio a las solicitudes de muchos otros hosts, que son los clientes.

Con este tipo de arquitectura los clientes no se comunican directamente entre sí. Otra característica es que el servidor tiene una dirección fija y conocida, denominada dirección IP. Puesto que el servidor tiene una dirección fija y conocida, y siempre está activo, un cliente siempre puede contactar con él enviando un paquete a su dirección IP.

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

Un protocolo de la capa de aplicación define cómo los procesos de una aplicación, que se ejecutan en distintos sistemas terminales, se pasan los mensajes entre sí. Un protocolo de la capa de aplicación define:

- Los tipos de mensajes intercambiados; por ejemplo, mensajes de solicitud y mensajes de respuesta.
- La sintaxis de los diversos tipos de mensajes, es decir, los campos de los que consta el mensaje y cómo se delimitan esos campos.
- La semántica de los campos, es decir, el significado de la información contenida en los campos.
- Las reglas para determinar cuándo y cómo un proceso envía mensajes y responde a los mismos.

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

El protocolo implementado para el intercambio consiste en, lógicamente, un servidor que está siempre escuchando peticiones. Un cliente que desee comunicarse con el servidor simplemente debe enviarle un paquete con el bit de SYN prendido (luego se detalla cómo se componen los paquetes) e informar si desea descargar o subir un determinado archivo.

En el caso del *UPLOAD*, una vez que el servidor da su aprobación para que le transfieran un archivo, el cliente comienza a mandar paquetes con 1024 bytes (configurable) del archivo, y cuando termina le envía un paquete sin payload con el bit de FINISHED prendido para que el servidor sepa que esa transferencia ya finalizó.

En el caso del *DOWNLOAD*, el cliente le envía el paquete con el SYN prendido, a lo que el servidor contesta con un ACK informando si puede descargar ese archivo, y si todo esta bien el cliente directamente se pone a esperar que el servidor comience a mandarle el archivo, hasta recibir un paquete con el bit de FINISHED que le indique que ya se transfirieron todos los paquetes.

Veamos ahora cómo se compone el paquete que intercambian el cliente y el servidor para la transferencia de archivos:

- 4 bytes para el número de paquete. El número de paquete 0 se usa tanto para el inicio o cierre de conexión.

- 1 byte de flags que está definido de la siguiente manera:
    - 1 bit indicando si es upload o download.
    - 1 bit que indica si finalizó la transferencia.
    - 1 bit que indica si el paquete es un ACK.
    - 1 bit para SYN (iniciar una conexión).
    - 2 bits de código de error.
    - 2 bits para versión de protocolo.
- 1 byte para el largo del nombre de archivo.
- N bytes para el nombre de archivo, donde N = largo del nombre del archivo. Como máximo N puede ser 255 debido al punto anterior.
- M bytes para el payload. Es importante aclarar que si bien M es configurable, no debe ser nunca superior al payload restante (considerando nuestro protocolo) de UDP. Recordemos que el payload máximo de un paquete UDP es 65.527 bytes (donde el tamaño máximo teórico del paquete es 65.535 bytes, la diferencia es de 8 bytes pertenecientes header).

Número de paquete (4 bytes)	Flags (1 byte)	Largo del nombre del archivo (1 byte)
Nombre del archivo (N bytes)		
Payload (M bytes)		

Y el byte para las flags se compone por los siguientes bits:

Es upload/download	Es fin de transferencia	Es ACK	Es SYN (inicio de conexión)	Código de error (2 bits): 00 no hubo error, el resto de combinaciones reservado para uso futuro	Versión del protocolo (2 bits): por default 0.

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?

UDP proporciona los servicios mínimos de un protocolo de la capa de transporte, es decir multiplexado y demultiplexado, y verificación de integridad. TCP, por otro lado, además ofrece los servicios de entrega confiable, control de flujo y control de congestión.

UDP tiene como características que tiene un header pequeño, no necesita conexión y pueden ocurrir pérdida de paquetes y haber paquetes duplicados. TCP en cambio es confiable, no se perderán paquetes ni llegarán duplicados; es un servicio orientado a conexión para la aplicación que lo usa y tiene muchos headers.

Se usa UDP cuando la velocidad en la entrega de los datos importa más que la confiabilidad de los mismos. Por ejemplo para streaming multimedia, telefonía por internet o juegos online, donde es probable que un paquete perdido no afecte a nadie. Por otra parte, TCP se suele usar en el resto de los casos donde la confiabilidad de entrega es imprescindible. Algunas aplicaciones que utilizan protocolos de aplicación con TCP son por ejemplo e-mail, web y transferencia de archivos.

Dificultades encontradas

- Determinar que campos debíamos incluir en el header del paquete a enviar.
- Mantener consistencia en los protocolos para mantener separada la aplicación y el envío de los paquetes.
- Determinar cuáles valores poner para los timeout y en donde poner los mismos.
- Manejar la variedad de errores que pensamos que podían suceder.
- Debuggear los errores.

Conclusiones

Tras completar la implementación del protocolo *Stop and Wait* y del protocolo *Go Back N*, y compararlos entre ellos tomando el tiempo que tardan en subir o bajar un archivo bajo diferentes condiciones de tamaño de archivo y pérdida de paquete, pudimos realizar comparaciones y obtener algunas conclusiones.

Primero que nada, como esperábamos pudimos observar que el tiempo de subir y de bajar un archivo bajo el mismo protocolo, tamaño y condiciones de pérdida son muy similares (no exactamente iguales por el estado de la red al momento de ejecutar).

Si comparamos entre protocolos, también como esperábamos pudimos observar que la implementación con *Go-Back-N* es más rápida que *Stop & Wait*. En la tabla presentada en la sección de Análisis vemos que bajo condiciones similares, en muchos casos GBN es unas 10 veces más rápido que SAW.

Consideramos el trabajo realizado muy importante para la correcta comprensión de los temas teóricos tratados. Tener que determinar cómo se conforma un paquete, cuando un paquete se perdió y tener que re-enviarlo, y de qué manera, entre otros

conceptos, fue un desafío interesante, y a la vez muy esclarecedor para todo el grupo.