.UBAfiuba
FACULTAD DE INGENIERÍA

TP1:
File Transfer UDP

[75.43/75.33/95.60]
Introducción a los Sistemas Distribuidos
2C 2022
Grupo 11

Alumno	Padrón	Email
Gomez, Joaquin	103735	joagomez@fi.uba.ar
Grassano, Bruno	103855	bgrassano@fi.uba.ar
Opizzi, Juan Cruz	99807	jopizzi@fi.uba.ar
Stancanelli, Guillermo	104244	gstancanelli@fi.uba.ar
Valdez, Santiago	103785	svaldez@fi.uba.ar

Índice

1. Introducción

El presente trabajo práctico tiene como objetivo la creación de una aplicación de red que transfiera archivos entre cliente-servidor. 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.

2. Hipótesis y suposiciones realizadas

Para la realización del trabajo practico se tomaron las siguientes hipótesis y supuestos.

- No se puede descargar un archivo que se esta subiendo o empezar a subir por otro cliente al mismo archivo.
- Si un archivo ya existe en el servidor se remplaza.
- Puede haber clientes usando *Stop and Wait* y otros con *Go Back N* en simultaneo.
- El tamaño máximo que se puede procesar de los archivos es de $4GB$.
- No se enviaran archivos lo suficientemente grandes como para que se acaben los números de secuencia y acknowledge resultando en que se reinicie su rango.
- El largo del nombre del archivo máximo es de 58 bytes.
- Se soportan como máximo 64512 conexiones simultaneas. (64k menos los primeros 1024 bien conocidos)
- El cliente tiene espacio en su disco para guardar los archivos que descarga.

3. Implementación

3.1. Protocolo

Para el desarrollo del protocolo empezamos definiendo que meta-datos son necesarios enviar en cada segmento. Para ayudarnos en este problema tomamos como base el *header* utilizado por TCP y empezamos a debatir sobre cada campo de forma tal de simplificarlo para lo que necesitamos. El encabezado al que llegamos, denominado `RDTPHeader`, tiene los siguientes campos.

- `seq_num` Es un entero sin signo (4 bytes) que indica el numero de secuencia del segmento que se esta enviando. Al comenzar el intercambio empieza en cero.
- `ack_num` Es un entero sin signo (4 bytes) que indica hasta cual segmento se tiene conocimiento del otro lado de la comunicación. Al comenzar el intercambio empieza en cero.

  Para simplificar el entendimiento del *acknowledge* de los paquetes decidimos que estos sean por paquetes, su numero de secuencia. Por ejemplo, se envía el paquete 1, se hace acknowledge de 1.
- `fin` Es un booleano (un byte) que indica que se quiere terminar la comunicación.

Inicialmente planteamos incluir en nuestro *header* los campos posición de los datos, ventana, y *ack_ only*. Con el avance del desarrollo estos fueron descartados debido a que no fue necesario su uso.

Para poder asegurarnos de que se respetan los bytes que establecimos en la implementación se utiliza la biblioteca `struct` para convertir a tipos de datos de C representados como bytes.

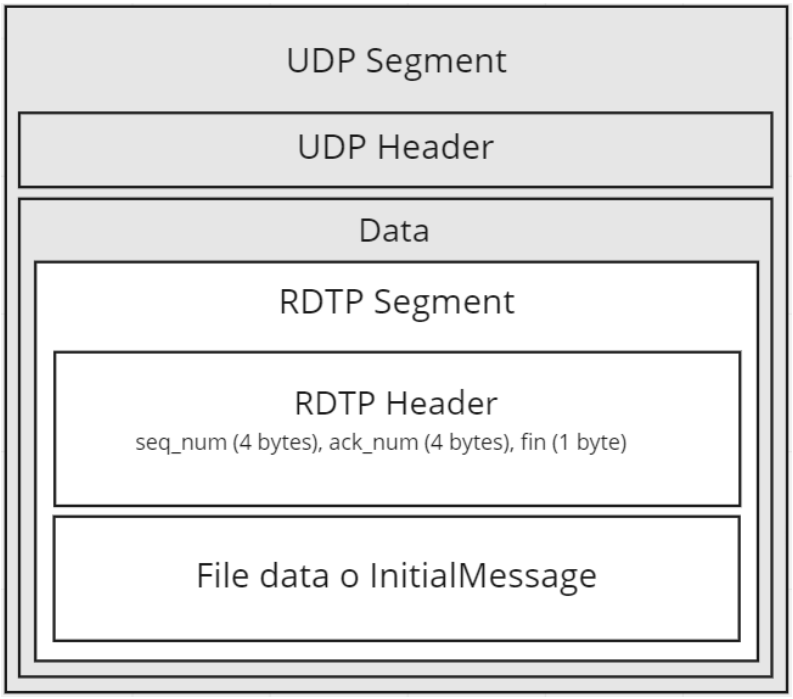


Figura 1: Diagrama mostrando la ubicación del segmento y nuestro header. Se muestran adentro de un paquete de UDP.

Como control de la validez de los paquetes esta el *checksum* realizado por UDP y la biblioteca `struct`. Esta biblioteca provee otra capa de validez en caso de que no pueda mappear correctamente al tipo esperado. Si este caso se da se descarta el paquete y considerara como un intento fallido.

3.1.1. Handshake

Con el problema del *header* ya resuelto continuamos viendo como entablar una comunicación inicial. Para este apartado tuvimos que tener en cuenta que se pueden realizar dos operaciones distintas y que se requiere en el servidor información que tiene el cliente. Para esto creamos un mensaje de 64 bytes llamado `InitialMessage` que contiene los siguientes campos.

- `upload` Es un booleano (un byte) que indica si la comunicación va a ser de subida (`True`) o de descarga (`False`).
- `is_saw` Es un booleano (un byte) que indica si el cliente quiere que la comunicación sea a través del protocolo con *Stop and Wait* (`True`). Caso contrario (`False`) se usa *Go Back N*
- `file_size` Es un entero sin signo (4 bytes) que indica el tamaño del archivo a subir o descargar en bytes. Esto implica que el tamaño máximo de archivo sea de $4 GB$
- `filename` Es un string (58 bytes) que contiene el nombre del archivo a subir o descargar. Se eligió este tamaño para que el mensaje quede en 64 bytes.

El servidor al recibir este mensaje responderá si puede atender la solicitud enviando un segmento con `InitialMessage`. Esta información va adentro de un segmento llamado `RDTPSegment`, que agrupa al `RDTPHeader` y datos.

La respuesta va a ser devolviendo el `InitialMessage` con el campo de `file_size` completo si se quiere descargar un archivo. Adicionalmente si se puede atender la solicitud se pone en *False* el campo `fin`, si se pone en *True* no se puede atender. Esto puede pasar si no se tiene espacio en disco o no se tiene el archivo.

Al enviar esta respuesta desde un nuevo *socket UDP*[1] se espera a recibir respuesta por parte del cliente (algún ACK) para tener confirmación de que sabe a que *socket* responder. A continuación se muestran diferentes diagramas mostrando el comportamiento del *handshake*.

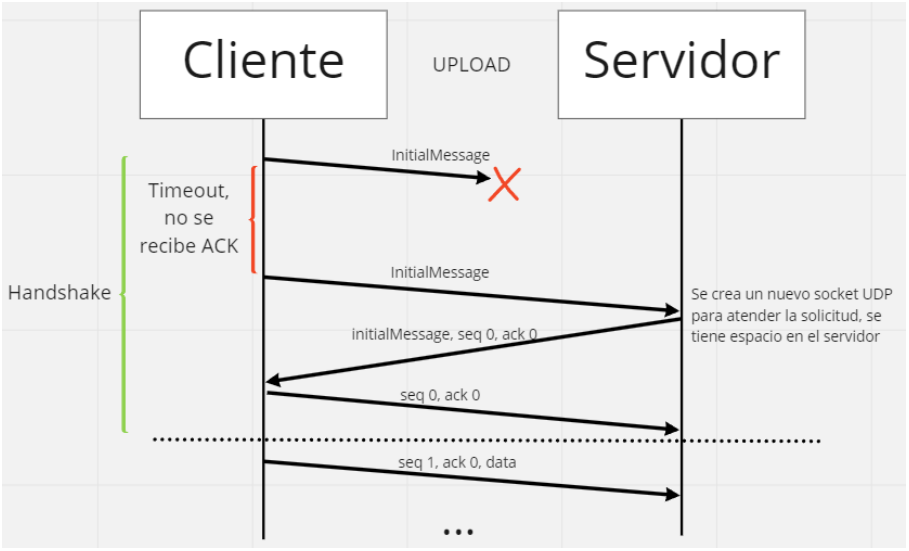


Figura 2: Perdida del primer paquete hacia el servidor, se sale por timeout y continua con handshake normal

[1] Ver sección concurrencia 3.2.3

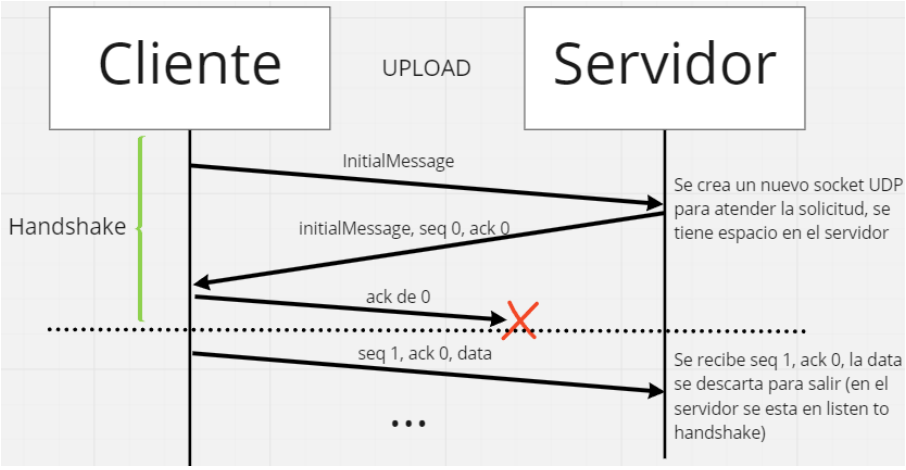


Figura 3: Perdida del ultimo ACK del handshake, en UPLOAD se recupera al recibir el mensaje posterior de datos con ACK

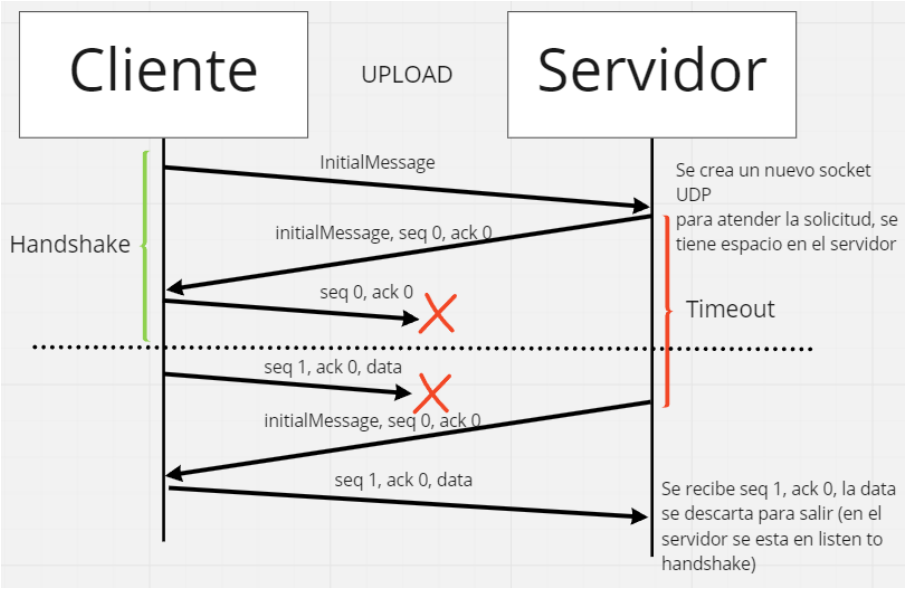


Figura 4: Perdida del ultimo ACK y primer paquete de datos, suponiendo Stop And Wait, saltaría un timeout en el handshake enviando el InitialMessage devuelta, recibiendo respuesta del cliente.

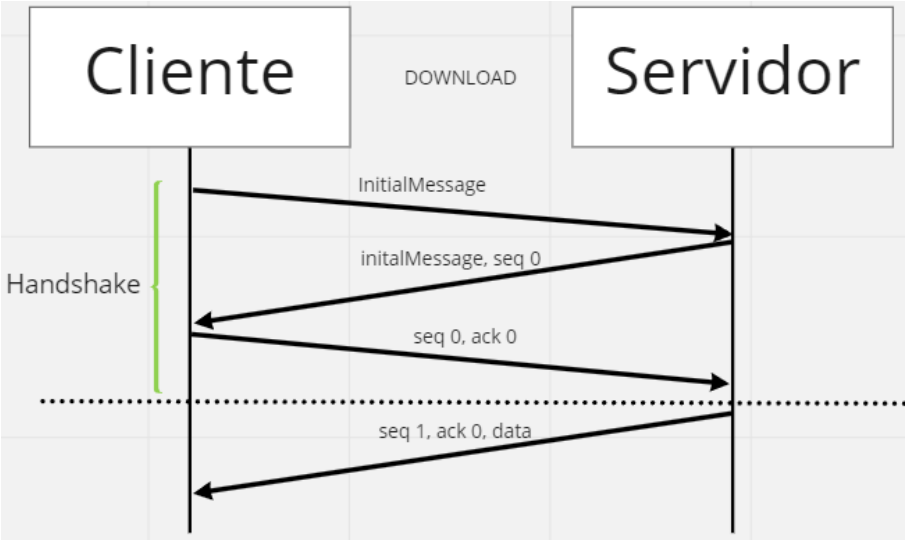


Figura 5: Handshake normal en descarga de archivo.

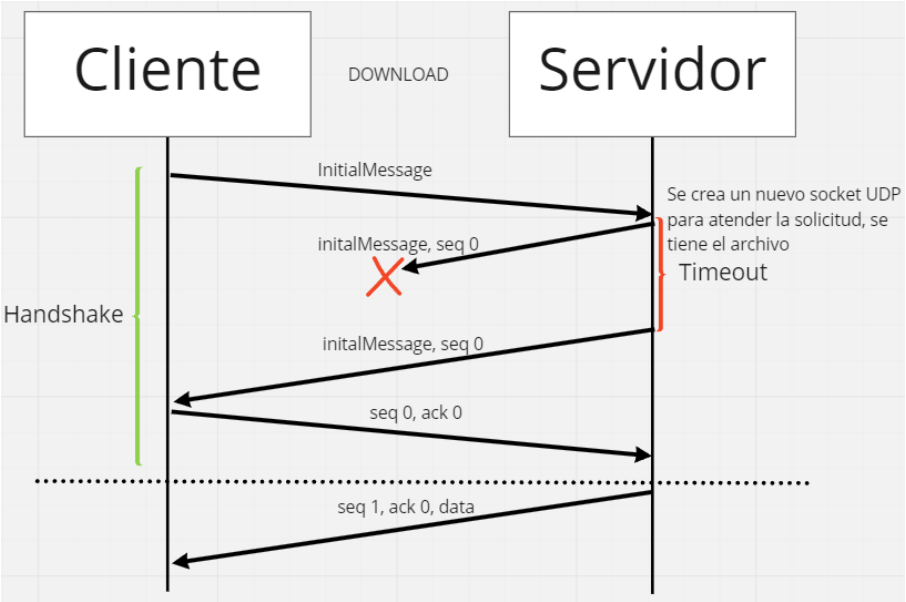


Figura 6: Se pierde paquete de Initial Message conteniendo el tamaño del archivo, salta un timeout debido a no recibir ACK y vuelve a intentar.

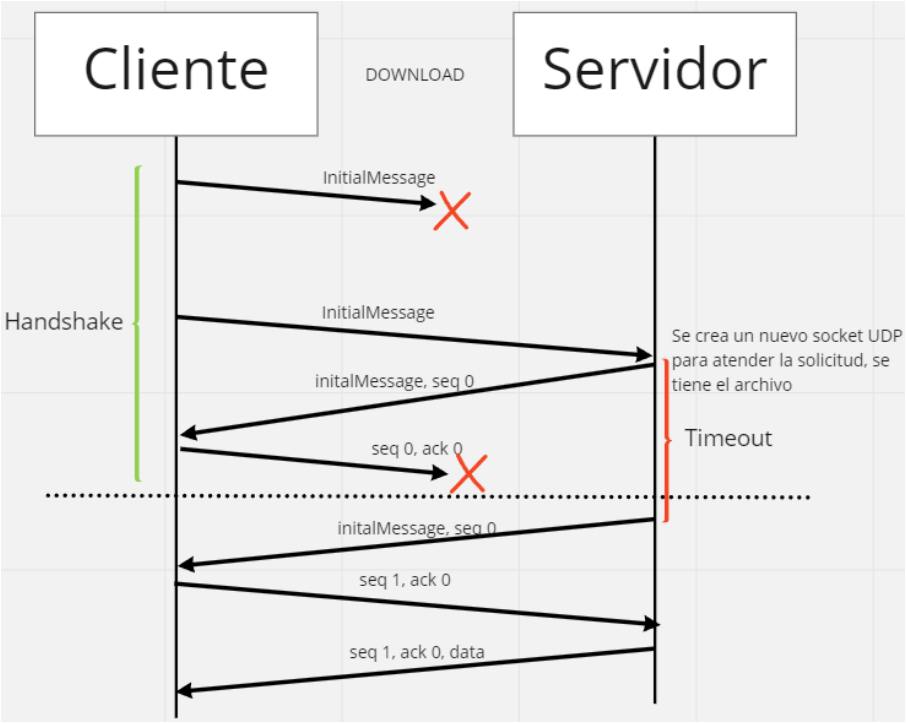


Figura 7: Se pierde primera comunicación, salta timeout y vuelve a intentar. Después se pierde el ultimo ACK, el servidor envía devuelta el Initial Message y el cliente responde con ACK

El código correspondiente al *handshake* se puede observar en `lib/protocols/base_protocol.py`

Para más detalles sobre como continúan los flujos de subida y descarga ver las secciones 3.2.2, 3.3.1, y 3.3.2.

Información adicional: En las primeras versiones del trabajo se decidió utilizar un *handshake* más sencillo que consistía de solamente la respuesta del *InitialMessage* sin un ACK. Esta versión se quedo corta cuando se empezó a encarar la descarga del archivo, ya que quien entabla la comunicación es el servidor. En la subida funcionaba debido a que se recibían los datos del archivo. Finalmente esta versión se descarto y se llego a la actual.

3.1.2. Stop and Wait

El protocolo *Stop and Wait* consiste en ir enviando y esperando al ACK de cada segmento antes de poder pasar al siguiente. En este protocolo tenemos los siguientes casos a considerar.

1. Si se envía y recibe su ACK se procede con la lectura del archivo.
2. Desde el lado del que envía, en caso de que se pierda algún paquete que envía data del archivo, eventualmente saltara un *timeout* definido en `CLIENT_STOP_AND_WAIT_TIMEOUT` ya que no se va a recibir su correspondiente ACK. Esto hará que se vuelva a enviar y repita el ciclo hasta que se reciba el ACK o repita una cantidad de veces determinada por `TIMEOUT_RETRY_ATTEMPTS`. Si este limite se alcanza se asume que se perdió la conexión con la otra parte y se procederá a cerrarla.
3. Desde el lado del que lee, si no se recibe un paquete en `CLIENT_STOP_AND_WAIT_TIMEOUT` saltara un *timeout* y repetirá `TIMEOUT_RETRY_ATTEMPTS`. Si no se obtiene nueva información se dará por perdida la conexión.
4. Desde el lado del que lee, si se pierde un ACK enviado como respuesta a un segmento o recibe un paquete anterior se enviara un paquete con el numero de ACK alcanzado y se volverá a intentar leer sin aumentar los reintentos.

El código de este protocolo se puede observar en el archivo `lib/protocols/stop_and_wait.py`

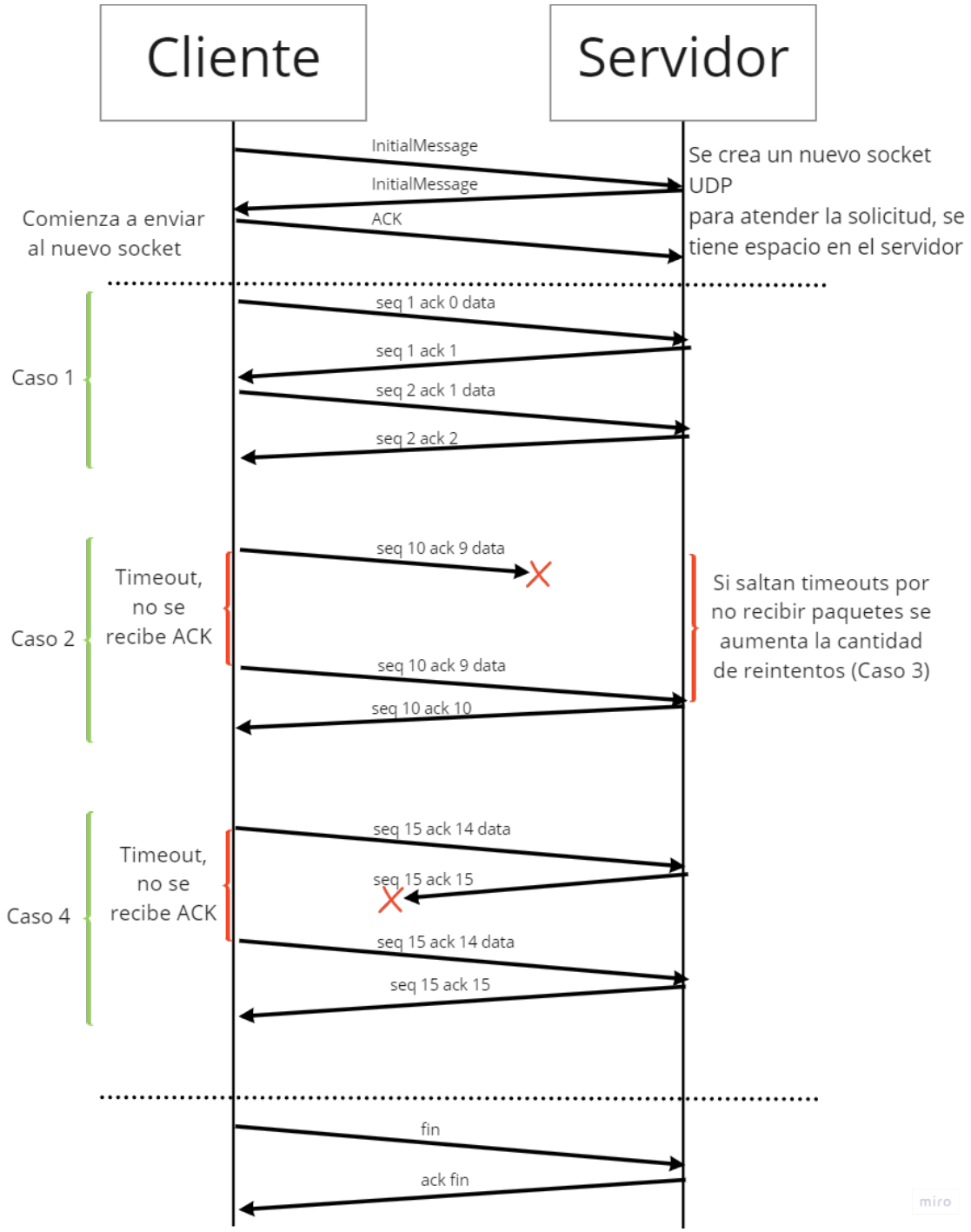


Figura 8: Diagrama mostrando los diferentes casos mencionados previamente para una subida de archivo.

3.1.3. Go Back N

El protocolo *Go Back N* consiste en ir enviando segmentos mientras que se tenga lugar en la ventana.

Tenemos los siguientes casos a considerar.

1. Si se llena la ventana se quedara esperando a recibir ACK de alguno de los paquetes enviados para liberar y hacer el deslizamiento.
2. Si se pierden ACKs pero recibe alguno mayor se deslizara la ventana.
3. Si se pierde algún paquete de datos eventualmente va a saltar un *timeout* y se reenviará la ventana. Si el receptor había recibido alguno superior se va a descartar.
4. Si no se recibe nada saltara un *timeout* y se hará el reenvió.

El código de este protocolo se puede observar en `lib/protocols/go_back_n.py`

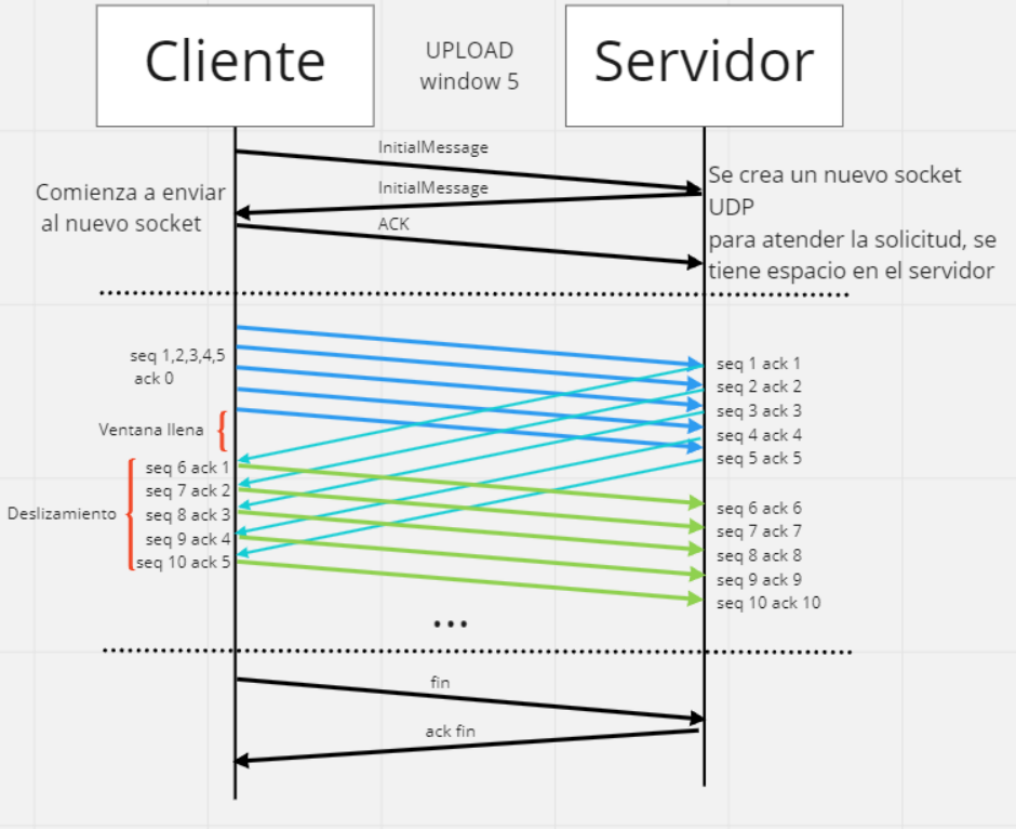


Figura 9: Diagrama mostrando la comunicación sin problemas en GBN. Se ve como se llena la ventana y se va deslizando.

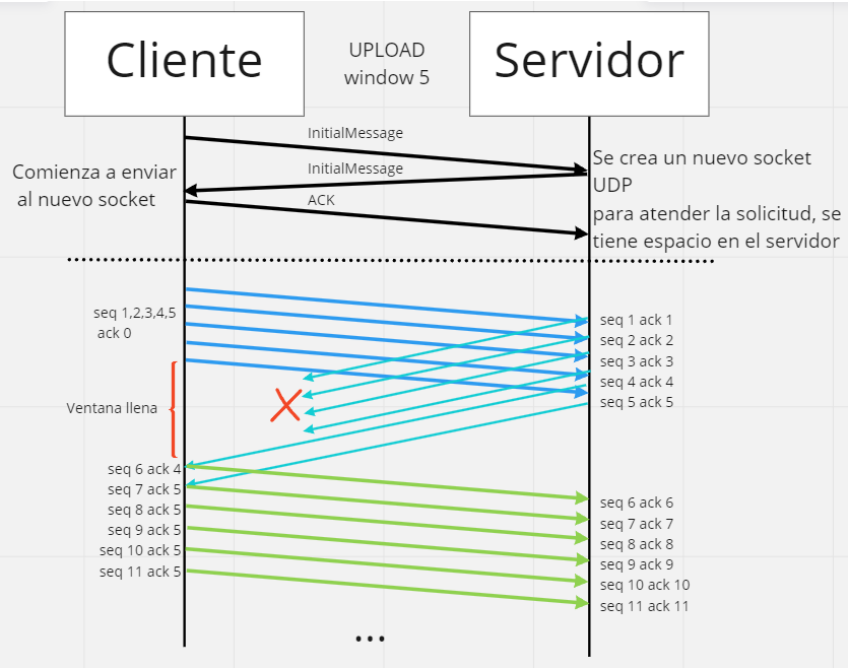


Figura 10: Diagrama mostrando perdida de paquetes de ACK, se recibe un ACK mayor y se continua con la subida.

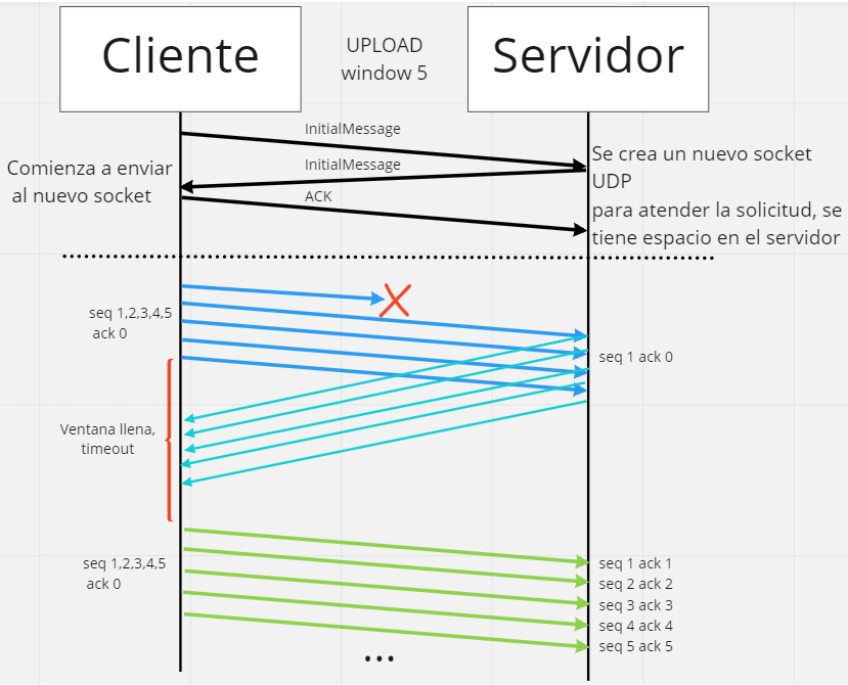


Figura 11: Diagrama mostrando la perdida del primer paquete de datos, se reciben los demás en el servidor y son descartados. Salta un timeout y se reenvían. Si sucede en cualquier otro paquete de la ráfaga es similar.

3.1.4. Close

Finalmente solo resta ver, como se finaliza una comunicación con nuestro protocolo. Para ello vamos a analizar, sin perdida de generalidad, los casos en los que el cliente esta subiendo un archivo al servidor. Esto se debe a que en el caso de descarga es idéntico, solo se intercambian los autores (cliente - servidor).

1. El primer caso a analizar es que sucede si no hay perdidas, para ello podemos observar el siguiente diagrama:

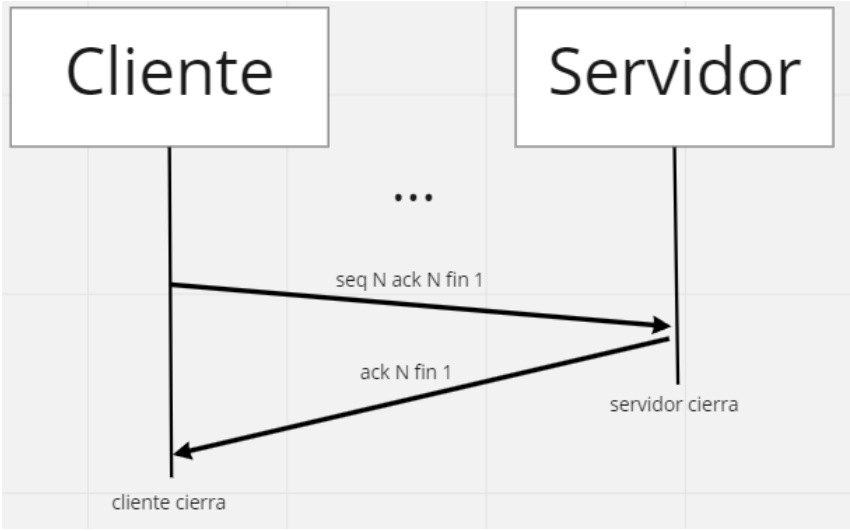


Figura 12: Diagrama mostrando el normal funcionamiento del close

Como podemos ver el cliente notifica con el *flag* de `fin` en el *header* que es el ultimo paquete a enviar y se queda esperando a recibir la confirmación del servidor. El servidor por su parte envía la confirmación y cierra automáticamente la conexión con el cliente.

2. Para el segundo caso podemos ya observar que sucede cuando se pierde el primer paquete de `fin` del cliente:

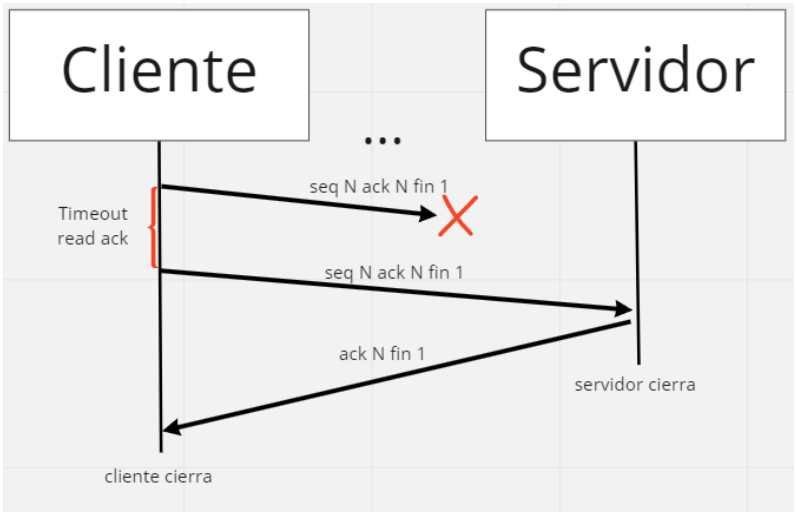


Figura 13: Diagrama mostrando la perdida del primer envío de fin, salta un timeout y se reenvía.

En este caso podemos ver que como el servidor no recibió ningún `fin`, no sabe que ya se termino la comunicación, es por ello que se decidió que el cliente espere a recibir la confirmación. Debido a que el cliente no recibe un ACK del `fin` decide, gracias al *timeout* del read, volver a enviar el paquete de `fin`, que en este caso recibe el servidor y retoma el flujo esperado del anterior diagrama.

3. Por ultimo, el caso más interesante es que sucede si se pierde el ACK del `fin` que envía el servidor:

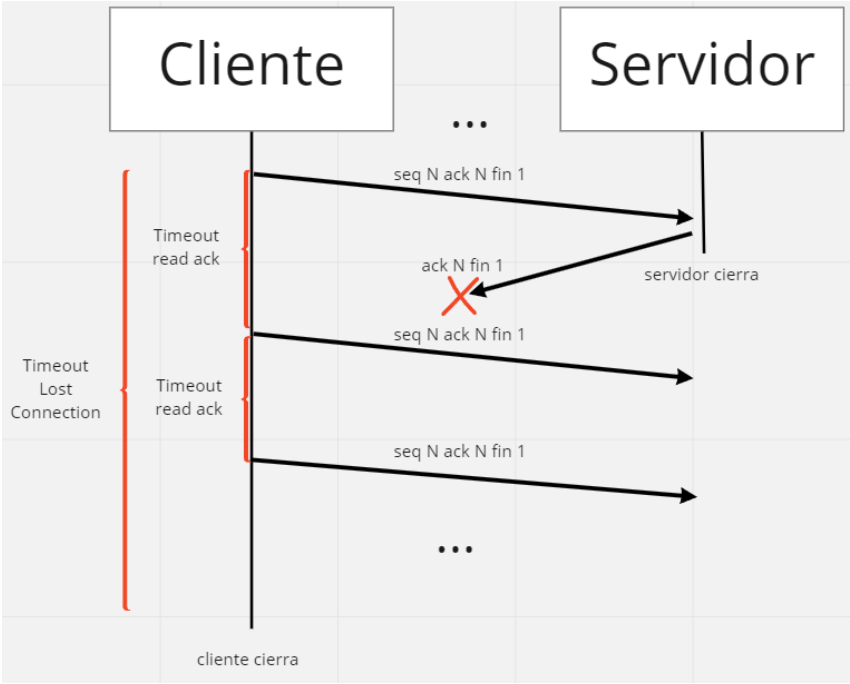


Figura 14: Diagrama mostrando la perdida del ACK enviado por el servidor, el servidor termina ejecución. El cliente permanece enviando fin hasta que termina debido a reintentos de timeout.

Como podemos ver, debido a que el servidor sale automáticamente ya no hay un servidor que responda a los sucesivos paquetes de `fin` que envía el cliente. Luego de varios intentos, salta el *timeout* pero esta vez por perdida de conexión que le hace saber al cliente que se perdió la conexión con el servidor, pero en el proceso del *close*. En este caso, al cliente no le queda otra opción que directamente asumir que se cerro bien y salir.

3.2. Servidor

Para poder iniciar el servidor se usa la siguiente linea.

```
python start-server.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-s STORAGE]
```

El servidor acepta los siguientes argumentos.

- `-h, --help` Muestra un mensaje de ayuda y termina el programa.
- `-v, --verbose` El programa se ejecuta de forma *verbose*, muestra niveles de ERROR, INFO, DEBUG
- `-q, --quiet` El programa se ejecuta de forma *quiet*, muestra solo ERROR
- `-H ADDR, --host ADDR` Establece un host para el servidor. Por defecto *localhost*
- `-p PORT, --port PORT` Para usar un puerto en especifico en el servidor. Por defecto *12000*
- `-s STORAGE, --storage STORAGE` Indica el directorio donde se van a guardar los archivos subidos. Por defecto se crea una carpeta llamada *storage* en el mismo directorio.

3.2.1. Estructura

El servidor se puede describir a través de los siguientes diagramas. Cada objeto va separando la lógica y desacoplando el modelo. De esta forma se evito mezclar responsabilidades.

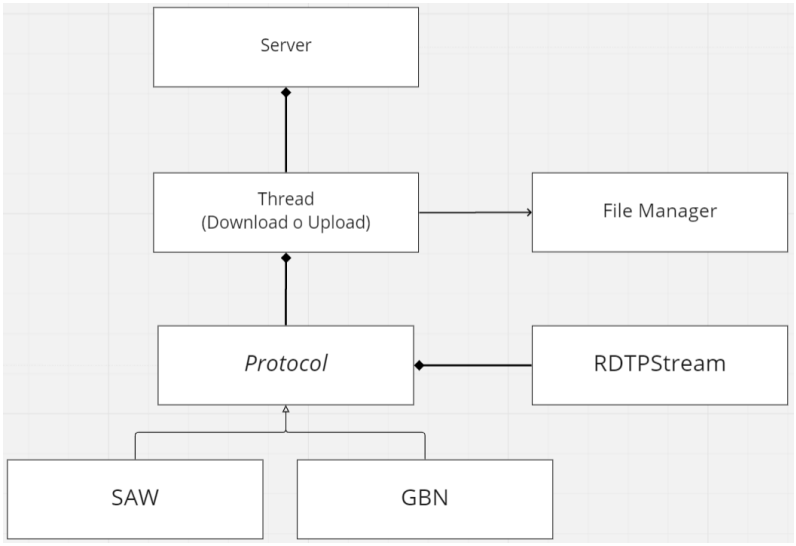


Figura 15: Diagrama mostrando la conformación del servidor.

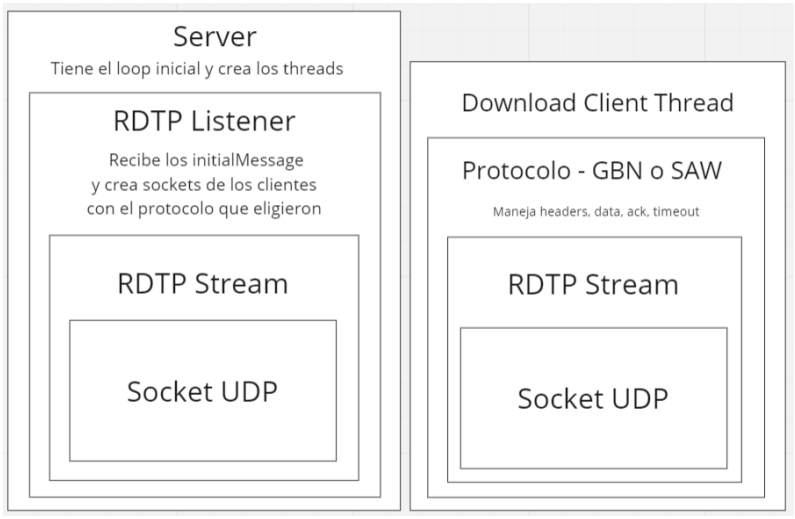


Figura 16: Diagrama mostrando como se agrupan las diferentes partes del modelo. Para el caso de subida es igual a la estructura de la derecha.

3.2.2. Algoritmo

El servidor sigue el siguiente algoritmo para su funcionamiento. Para el código ver `lib/server.py`

1. Configuraciones iniciales, parseo de argumentos, inicio del servidor
2. Mientras que se ejecute el programa:
   a) Escuchar mensajes provenientes al puerto. Estos mensajes son los *InitialMessage*.
   b) Crear un nuevo *thread* con su *socket* para manejar la nueva conexión. Este paso aplica para la descarga y subida.
   c) Iniciar el nuevo *thread*

Los nuevos *threads* creados seguirán su propia lógica.

Para la subida se siguen los siguientes pasos. Para el código ver `lib/UploadClientThread.py`

1. Realizar configuraciones iniciales, determinar si se va a utilizar *Stop and Wait* o *Go Back N*. Esta información se obtiene del mensaje inicial enviado por el cliente.
2. Determinar si existe el directorio para guardar el archivo. Si no existe crearlo.
3. Determinar si se tiene el espacio suficiente para guardar el archivo y comunicárselo al cliente. Si no se tiene el espacio cerrar la comunicación.
4. Crear el archivo
5. Repetir mientras que no se termine el archivo o la comunicación.
   a) Leer los datos recibidos del cliente
   b) Escribir al archivo.
6. Finalmente, se cierra la comunicación.

Para la descarga se siguen los siguientes pasos. Para el código ver `lib/DownloadClientThread.py`

1. Realizar configuraciones iniciales, determinar si se va a utilizar *Stop and Wait* o *Go Back N*. Esta información se obtiene del mensaje inicial enviado por el cliente.
2. Determinar si existe el archivo que se quiere descargar. Si no existe, manda un mensaje de indicándolo, el cliente devuelve ACK y se prosigue con el procedimiento de cierre de la conexión. Si existe responde agregando el tamaño.
3. Se abre el archivo en modo de lectura.
4. Repetir mientras que haya paquetes del archivo que mandar o no se termine la comunicación.
   a) Leer los datos del archivo.
   b) Mandar el paquete al cliente.
   c) Restar el avance realizado de la lectura.
5. Finalmente, se cierra la comunicación.

3.2.3. Concurrencia

Para que el servidor sea capaz de procesar de manera concurrente la transferencia de archivos con múltiples clientes se optó por crear un nuevo *thread* y *socket UDP* por cada cliente.

- El *thread* es el responsable de manejar el flujo de la subida o descarga de un archivo del servidor. En las figuras 15 y 16 se pueden ver las relaciones que tienen estos objetos.
- El *socket UDP* se crea para poder distinguir de forma más sencilla el trafico que entra y sale del servidor. Este nuevo *socket* se comunica con el *socket* del cliente para que este sepa que las futuras comunicaciones deben de dirigirse a el. Este procedimiento de creación de un nuevo *socket* es similar al realizado en TCP.

  Una alternativa que se consideró fue la de utilizar un único *socket*, el del servidor inicial, que maneje todo el trafico. Esta opción la terminamos descartando debido a que implicaría estar distribuyendo paquetes provenientes de distintos clientes.

3.3. Clientes

- El cliente en el trabajo se divide en dos aplicaciones distintas. Una para la subida y otra para la bajada.
- Para el desarrollo de las funcionalidades de *Stop and Wait* y *Go Back N* se decidió darle la posibilidad a los clientes de elegir el método. El método elegido es comunicado al servidor durante el primer mensaje.

3.3.1. Upload

Para poder subir un archivo se usa la siguiente linea.

```
python upload.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-saw | -gbn]
        [-s FILEPATH] [-n FILENAME]
```

Los argumentos tienen los siguientes significados.

- `-h, --help` Muestra un mensaje de ayuda y termina el programa.
- `-v, --verbose` El programa se ejecuta de forma *verbose*, muestra niveles de ERROR, INFO, DEBUG
- `-q, --quiet` El programa se ejecuta de forma *quiet*, muestra solo ERROR
- `-H ADDR, --host ADDR` Para agregar un host de servidor en especifico. Por defecto *localhost*
- `-p PORT, --port PORT` Para agregar un puerto de servidor en especifico. Por defecto *12000*
- `-saw, --stop_and_wait` Se usa *Stop And Wait* como algoritmo de transferencia
- `-gbn GO_BACK_N, --go_back_n` Se usa *Go Back N* como algoritmo de transferencia
- `-s FILEPATH, --src FILEPATH` Es el path del archivo a subir al servidor
- `-n FILENAME, --name FILENAME` Es el nombre del archivo a subir al servidor

El cliente de subida sigue la siguiente lógica al momento de su ejecución. Para el código ver `upload.py`

1. Realizar configuraciones iniciales, parseo de argumentos.
2. Determinar si existe el archivo que se quiere subir. Si no existe se termina el programa.
3. Obtener el tamaño del archivo que se quiere subir.
4. Crear un nuevo *socket* y protocolo en base a la configuración indicada en los argumentos.
5. Iniciar la comunicación con el servidor. Se le envía el mensaje inicial y se espera a la confirmación de que se puede empezar a subir. Si no se puede subir o no se obtiene respuesta se termina la ejecución.
6. Se abre el archivo.
7. Mientras que haya datos para enviar y se mantenga la conexión.
   a) Leer del archivo.
   b) Enviar la información al servidor.
   c) Restar el avance realizado en la lectura.
8. Finalizar la conexión y cerrar el archivo.

3.3.2. Download

Para poder descargar un archivo se usa la siguiente linea.

```
python download.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-saw | -gbn]
        [-d FILEPATH] [-n FILENAME]
```

Los argumentos tienen los siguientes significados.

- `-h, --help` Muestra un mensaje de ayuda y termina el programa.
- `-v, --verbose` El programa se ejecuta de forma *verbose*, muestra niveles de ERROR, INFO, DEBUG
- `-q, --quiet` El programa se ejecuta de forma *quiet*, muestra solo ERROR
- `-H ADDR, --host ADDR` Para agregar un host de servidor en especifico. Por defecto *localhost*
- `-p PORT, --port PORT` Para agregar un puerto de servidor en especifico. Por defecto *12000*
- `-saw, --stop_and_wait` Se usa *Stop And Wait* como algoritmo de transferencia
- `-gbn GO_BACK_N, --go_back_n` Se usa *Go Back N* como algoritmo de transferencia
- `-d FILEPATH, --dst FILEPATH` Es el path del destino del archivo a descargar
- `-n FILENAME, --name FILENAME` Es el nombre del archivo a descargar del servidor

El cliente de descarga sigue la siguiente lógica al momento de su ejecución. Para el código ver `download.py`

1. Realizar configuraciones iniciales, parseo de argumentos.
2. Si no existe la carpeta de destino, la crea.
3. Crear un nuevo *socket* y protocolo en base a la configuración indicada en los argumentos.
4. Iniciar la comunicación con el servidor. Se le envía el mensaje inicial y se espera a la confirmación de que se puede empezar a bajar. Si no se puede bajar o no se obtiene respuesta se termina la ejecución.
5. Se abre/crea un archivo en el que se van a descargar los datos.
6. Mientras que haya datos para recibir y se mantenga la conexión.
   a) Recibir la información del servidor.
   b) Escribir los datos recibidos a disco.
   c) Restar el avance realizado en la escritura.
7. Finalizar la conexión y cerrar el archivo.

4. Pruebas

4.1. Pruebas de consola

Se muestran resultados obtenidos por consola de diferentes escenarios probados.

4.1.1. Subida de archivo con SAW

Ambos con configuración por defecto, `localhost`, puerto `12000`. Se guarda en `./storage2`. Se muestra con nivel de información hasta DEBUG. Se sube un archivo de 2KB.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/upload.py -n 2K.o -saw -s . -v
2022-10-02 18:35:40,115 -  [ INFO ]  - Starting client (upload.py:69)
2022-10-02 18:35:40,116 -  [ DEBUG ]  - Found file in ./2K.o with size 2048 bytes (upload.py:32)
2022-10-02 18:35:40,117 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 sending first message of handshake (base_protocol.py:35)
2022-10-02 18:35:40,122 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 received second message of handshake (base_protocol.py:38)
2022-10-02 18:35:40,122 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending third message of handshake (base_protocol.py:45)
2022-10-02 18:35:40,124 -  [ DEBUG ]  - Opened file with name: ./2K.o (file_manager.py:21)
2022-10-02 18:35:40,125 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:35:40,125 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message with seq_num: 1 (stop_and_wait.py:23)
2022-10-02 18:35:40,128 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:35:40,128 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message with seq_num: 2 (stop_and_wait.py:23)
2022-10-02 18:35:40,130 -  [ DEBUG ]  - Closed file with name: ./2K.o (file_manager.py:29)
2022-10-02 18:35:40,130 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:35:40,131 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message to end connection (base_protocol.py:91)
2022-10-02 18:35:40,131 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:35:40,131 -  [ INFO ]  - End of execution (upload.py:74)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/upload.py -n 2K.o -saw -s . -v
2022-10-02 18:35:40,115 -  [ INFO ]  - Starting client (upload.py:69)
2022-10-02 18:35:40,116 -  [ DEBUG ]  - Found file in ./2K.o with size 2048 bytes (upload.py:32)
2022-10-02 18:35:40,117 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 sending first message of handshake (base_protocol.py:35)
2022-10-02 18:35:40,122 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 received second message of handshake (base_protocol.py:38)
2022-10-02 18:35:40,122 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending third message of handshake (base_protocol.py:45)
2022-10-02 18:35:40,124 -  [ DEBUG ]  - Opened file with name: ./2K.o (file_manager.py:21)
2022-10-02 18:35:40,125 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:35:40,125 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message with seq_num: 1 (stop_and_wait.py:23)
2022-10-02 18:35:40,128 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:35:40,128 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message with seq_num: 2 (stop_and_wait.py:23)
2022-10-02 18:35:40,130 -  [ DEBUG ]  - Closed file with name: ./2K.o (file_manager.py:29)
2022-10-02 18:35:40,130 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:35:40,131 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 49368 sending message to end connection (base_protocol.py:91)
2022-10-02 18:35:40,131 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:35:40,131 -  [ INFO ]  - End of execution (upload.py:74)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

4.1.2. Subida de archivo con GBN

Ambos con configuración por defecto, `localhost`, puerto `12000`. Se guarda en `./storage2`. Se muestra con nivel de información hasta DEBUG. Se sube un archivo de 2KB. Se remplaza el archivo subido previamente con SAW.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/start-server.py -s storage2 -v
2022-10-02 18:40:48,177 -  [ INFO ]  - Starting server (start-server.py:24)
2022-10-02 18:40:48,178 -  [ INFO ]  - Ready to receive connections (server.py:21)
2022-10-02 18:40:58,141 -  [ DEBUG ]  - Client request coming from: ('127.0.0.1', 34212) (rdtplistener.py:29)
2022-10-02 18:40:58,141 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 18:40:58,143 -  [ DEBUG ]  - Started thread for a new client (server.py:38)
2022-10-02 18:40:58,145 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 34212 sending second message of handshake (base_protocol.py:73)
2022-10-02 18:40:58,147 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 34212 sending third message of handshake (base_protocol.py:76)
2022-10-02 18:40:58,147 -  [ DEBUG ]  - Receiving file 2K.o of 2048 from client (UploadClientThread.py:54)
2022-10-02 18:40:58,152 -  [ DEBUG ]  - Opened file with name: storage2/2K.o (file_manager.py:21)
2022-10-02 18:40:58,153 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 34212 sending message with ack: 1 (go_back_n.py:112)
2022-10-02 18:40:58,154 -  [ DEBUG ]  - Wrote 1024 bytes to file with name: storage2/2K.o (file_manager.py:55)
2022-10-02 18:40:58,154 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 34212 sending message with ack: 2 (go_back_n.py:112)
2022-10-02 18:40:58,155 -  [ DEBUG ]  - Wrote 1024 bytes to file with name: storage2/2K.o (file_manager.py:55)
2022-10-02 18:40:58,155 -  [ INFO ]  - End of client upload of file (UploadClientThread.py:63)
2022-10-02 18:40:58,156 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:40:58,156 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 34212 sending message to end connection (base_protocol.py:91)
2022-10-02 18:40:58,157 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
^C2022-10-02 18:41:17,212 -  [ INFO ]  - Signal error, closing server (start-server.py:28)
2022-10-02 18:41:17,212 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:41:17,213 -  [ INFO ]  - Closed server (start-server.py:32)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/upload.py -n 2K.o -s . -v
2022-10-02 18:40:58,138 -  [ INFO ]  - Starting client (upload.py:69)
2022-10-02 18:40:58,140 -  [ DEBUG ]  - Found file in ./2K.o with size 2048 bytes (upload.py:32)
2022-10-02 18:40:58,140 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 sending first message of handshake (base_protocol.py:35)
2022-10-02 18:40:58,146 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 received second message of handshake (base_protocol.py:38)
2022-10-02 18:40:58,146 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 36413 sending third message of handshake (base_protocol.py:45)
2022-10-02 18:40:58,148 -  [ DEBUG ]  - Opened file with name: ./2K.o (file_manager.py:21)
2022-10-02 18:40:58,149 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:40:58,151 -  [ DEBUG ]  - Window isnt full, sending packet to host: 127.0.0.1 and port: 36413 sending message with seq_num: 1 (go
_back_n.py:35)
2022-10-02 18:40:58,152 -  [ DEBUG ]  - Reading 1024 bytes from file with name: ./2K.o (file_manager.py:41)
2022-10-02 18:40:58,152 -  [ DEBUG ]  - Window isnt full, sending packet to host: 127.0.0.1 and port: 36413 sending message with seq_num: 2 (go
_back_n.py:35)
2022-10-02 18:40:58,153 -  [ DEBUG ]  - Closed file with name: ./2K.o (file_manager.py:29)
2022-10-02 18:40:58,154 -  [ DEBUG ]  - acked packet up to num: 0 (go_back_n.py:66)
2022-10-02 18:40:58,155 -  [ DEBUG ]  - acked packet up to num: 0 (go_back_n.py:66)
2022-10-02 18:40:58,155 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:40:58,156 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 36413 sending message to end connection (base_protocol.py:91)
2022-10-02 18:40:58,156 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:40:58,157 -  [ INFO ]  - End of execution (upload.py:74)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

4.1.3. Descarga de archivo

Ambos con configuración por defecto, `localhost`, puerto `12000`, se busca en `./storage`, con GBN. Se muestra con nivel de información hasta INFO. Se descarga un archivo de 100KB.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/start-server.py
2022-10-02 19:35:40,782 -  [ INFO ]  - Starting server (start-server.py:24)
2022-10-02 19:35:40,783 -  [ INFO ]  - Ready to receive connections (server.py:21)
2022-10-02 19:35:44,878 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 19:35:44,896 -  [ INFO ]  - End of reading of file (DownloadClientThread.py:61)
^C2022-10-02 19:35:49,465 -  [ INFO ]  - Signal error, closing server (start-server.py:28)
2022-10-02 19:35:49,465 -  [ INFO ]  - Closed server (start-server.py:32)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/download.py -n 100.o -d download -gbn
2022-10-02 19:35:44,877 -  [ INFO ]  - Starting client (download.py:69)
2022-10-02 19:35:44,898 -  [ INFO ]  - Downloaded file (download.py:51)
2022-10-02 19:35:44,899 -  [ INFO ]  - End of execution (download.py:74)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

4.1.4. Muerte de servidor con cliente funcionando

Ambos con configuración por defecto, `localhost`, puerto `12000`, se busca en `./storage`. Se muestra con nivel de información hasta INFO. Se sube un archivo y se termina el servidor.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/start-server.py
2022-10-02 19:40:40,920 -  [ INFO ]  - Starting server (start-server.py:24)
2022-10-02 19:40:40,920 -  [ INFO ]  - Ready to receive connections (server.py:21)
2022-10-02 19:40:47,351 -  [ INFO ]  - Received a new client request (server.py:29)
^C2022-10-02 19:40:49,654 -  [ INFO ]  - Signal error, closing server (start-server.py:28)
2022-10-02 19:40:49,655 -  [ ERROR ]  - Closing error.  (UploadClientThread.py:72)
2022-10-02 19:40:49,657 -  [ INFO ]  - Closed server (start-server.py:32)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/upload.py -n 100M.o -s . -saw
2022-10-02 19:40:47,348 -  [ INFO ]  - Starting client (upload.py:70)
2022-10-02 19:40:52,660 -  [ ERROR ]  - Lost connection to server.  (upload.py:59)
2022-10-02 19:40:52,661 -  [ INFO ]  - End of execution (upload.py:75)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

4.1.5. Concurrencia

Configuración por defecto, `localhost`, puerto `12000`. Se guarda en `./storage`. Se muestra con nivel de información hasta INFO. Se sube un archivo de 5MB con *Stop and Wait*, se descarga uno de 1M con *Go Back N*, y se sube otro de 100KB con *Go Back N*.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/start-server.py
2022-10-02 18:51:27,135 -  [ INFO ]  - Starting server (start-server.py:24)
2022-10-02 18:51:27,135 -  [ INFO ]  - Ready to receive connections (server.py:21)
2022-10-02 18:51:28,741 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 18:51:29,604 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 18:51:29,789 -  [ INFO ]  - End of reading of file (DownloadClientThread.py:61)
2022-10-02 18:51:30,113 -  [ INFO ]  - End of client upload of file (UploadClientThread.py:63)
2022-10-02 18:51:30,559 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 18:51:30,574 -  [ INFO ]  - End of client upload of file (UploadClientThread.py:63)
^C2022-10-02 18:51:52,495 -  [ INFO ]  - Signal error, closing server (start-server.py:28)
2022-10-02 18:51:52,496 -  [ INFO ]  - Closed server (start-server.py:32)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

Figura 17: Se muestran con colores diferentes entre que puntos están corriendo los clientes.

4.1.6. Archivo inexistente

Ambos con configuración por defecto, `localhost`, puerto `12000`, se busca en `./storage`. Se muestra con nivel de información hasta DEBUG.

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/start-server.py -v
2022-10-02 18:18:05,583 -  [ INFO ]  - Starting server (start-server.py:24)
2022-10-02 18:18:05,584 -  [ INFO ]  - Ready to receive connections (server.py:21)
2022-10-02 18:18:09,628 -  [ DEBUG ]  - Client request coming from: ('127.0.0.1', 60804) (rdtplistener.py:29)
2022-10-02 18:18:09,629 -  [ INFO ]  - Received a new client request (server.py:29)
2022-10-02 18:18:09,630 -  [ DEBUG ]  - Started thread for a new client (server.py:38)
2022-10-02 18:18:09,631 -  [ ERROR ]  - File in ./storage/inexistente.o doesn't exists (DownloadClientThread.py:41)
2022-10-02 18:18:09,631 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 60804 sending second message of handshake (base_protocol.py:73)
2022-10-02 18:18:09,633 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 60804 sending third message of handshake (base_protocol.py:76)
^C2022-10-02 18:18:14,033 -  [ INFO ]  - Signal error, closing server (start-server.py:28)
2022-10-02 18:18:14,034 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:18:14,034 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:18:14,035 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 60804 sending message to end connection (base_protocol.py:91)
2022-10-02 18:18:14,035 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:18:14,036 -  [ INFO ]  - Closed server (start-server.py:32)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

```
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$ python3 src/download.py -n inexistente.o -saw -d . -v
2022-10-02 18:18:09,626 -  [ INFO ]  - Starting client (download.py:68)
2022-10-02 18:18:09,627 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 sending first message of handshake (base_protocol.py:35)
2022-10-02 18:18:09,632 -  [ DEBUG ]  - Socket in host: localhost and port: 12000 received second message of handshake (base_protocol.py:38)
2022-10-02 18:18:09,632 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 32788 sending third message of handshake (base_protocol.py:45)
2022-10-02 18:18:09,633 -  [ ERROR ]  - File not found in server (download.py:40)
2022-10-02 18:18:09,633 -  [ DEBUG ]  - Ending connection (base_protocol.py:88)
2022-10-02 18:18:09,633 -  [ DEBUG ]  - Socket in host: 127.0.0.1 and port: 32788 sending message to end connection (base_protocol.py:91)
2022-10-02 18:18:09,634 -  [ DEBUG ]  - Closing socket (rdtpstream.py:64)
2022-10-02 18:18:09,634 -  [ INFO ]  - End of execution (download.py:73)
bruno@Desktop3:/mnt/d/Users/Bruno/Documents/GitHub/TP1-File-Transfer-UDP$
```

4.2. Mediciones

Para comparar Go Back N y Stop and Wait se utilizarán tres archivos de prueba de 100KB, 1 MB y 5 MB respectivamente. En cuanto a la perdida de paquetes se evaluará sin perdida, con 5 % y con 10 % utilizando la herramienta solicitada *Comcast*. La operaciones a realizar serán upload y download.

Para poder crear los archivos se pueden usar los siguientes comandos.

```
truncate -s 100K 100.o
truncate -s 1M 1.o
truncate -s 5M 5.o
```

Para correr el programa.

```
# 'lo' es el nombre de la red
comcast --device=lo --packet-loss=10% --target-addr=127.0.0.0/8
        --target-proto=udp --target-port=1024:65535

comcast -device=lo --stop
```

El tiempo se midió cronometrando el tiempo de ejecución del cliente con el fin de mostrar el orden de magnitud de las corridas, más allá de mostrar valores exactos.

En las siguientes secciones se muestran los resultados obtenidos para los diferentes casos y por último un breve análisis.

4.2.1. Mediciones de upload para Stop and Wait

	100KB	1MB	5MB
Sin perdida	108ms	357ms	1.482 s
5 % de perdida	564ms	3.517s	18.358s
10 % de perdida	948ms	8.974s	39.070s

4.2.2. Mediciones de upload para Go Back N

	100KB	1MB	5MB
Sin perdida	103ms	316ms	1,211s
5 % de perdida	289ms	1.897s	9.683s
10 % de perdida	418ms	4.043s	17.735s

4.2.3. Mediciones de download para Stop and Wait

	100KB	1MB	5MB
Sin perdida	120ms	337ms	1,426s
5 % de perdida	416ms	3.433s	18.420s
10 % de perdida	878ms	7.573s	38.531s

4.2.4. Mediciones de download para Go Back N

	100KB	1MB	5MB
Sin perdida	72ms	309ms	1.250s
5 % de perdida	193ms	1,971s	9.065
10 % de perdida	415ms	4.022s	19.548s

4.2.5. Pruebas adicionales hechas

Se probaron archivos superiores a los 5MB y perdidas mayores a 10 % antes de optar por las pruebas realizadas por motivos del tiempo que empezaron a tomar las corridas.

Un ejemplo de los mencionados es la discografía completa de *Slayer* en el orden de 1.2GB **sin perdida**.

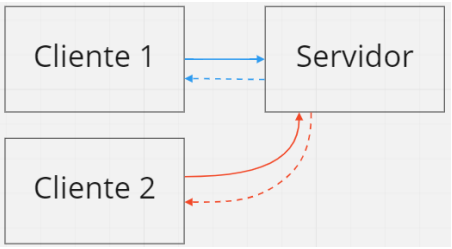
4.2.6. Análisis de resultados

Como era de esperarse, el protocolo utilizando *Go Back N* resulta en ejecuciones más rápidas en comparación a *Stop And Wait*. Esta diferencia se puede observar más a medida que aumenta la perdida de paquetes.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor consiste de un *host* llamado Servidor a la que le llegan *requests* de otros *hosts* llamados Clientes. Se dice que el servidor responde a lo que pide el cliente, siendo que el cliente es el que inicia la comunicación.



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

La capa de aplicación tiene como función principal lograr la creación de aplicaciones que correr en distintos *hosts*, comunicándose a través de la red. Esta capa se encarga de brindar soporte a las aplicaciones que desarrollamos. Dicho protocolo se encarga de la comunicación de aplicaciones entre *hosts*. Estas aplicaciones puede ser, por ejemplo, WhatsApp o Telegram.

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

El protocolo de aplicacion desarrollado se encuentra detallado en la sección 3.1 correspondiente a la implementación.

5.4. ¿Qué servicios proveen TCP y UDP? ¿Características? ¿Usos?

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

¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características?

Empezamos viendo UDP, este protocolo no ofrece demasiados servicios.

- Chequeo de errores a través del *checksum*, se puede pasar el paquete con algún *warning* a la capa de aplicación o descartarlo.
- Comunicación *Process to Process*, se comunican los procesos entre si a través de un canal en ambas direcciones.

Respecto de TCP, ofrece los de UDP y los siguientes.

- *Reliable Data Transfer*, esto implica que a la capa de aplicación le van a llegar los paquetes en orden, sin perdidas, sin errores y sin duplicados.
- Control de flujo, evita que el que envía desborde a la aplicación que hace la lectura del *buffer*. Iguala la tasa de envió a la de lectura.
- Control de congestión, limita la tasa a la que el transmisor envía trafico a la conexión en función de la congestión que percibe en la red.

Una característica común a ambos es que no proveen una capa de seguridad ni aseguran un tiempo de transmisión.

¿Cuando es apropiado utilizar cada uno?

Las razones por las que se puede elegir UDP son las siguientes.

- Es un protocolo que provee un mayor control de que datos y cuando se le pasan a la capa de aplicación. Esto se debe a que UDP agrega sus *headers* a los datos y pasa el segmento a la capa de red. En el caso de TCP esto no es así debido a su mecanismo de control de congestión y envío de datos confiable.
- No es necesario un establecimiento de la conexión previo como en TCP. De esta manera se esta evitando algo de *delay*.
- No mantiene un estado de la conexión como en TCP. Esto permite que se puedan mantener muchos más clientes activos en UDP que TCP.
- Tiene un menor *header overhead*, son 8 bytes en vez de los 20 bytes de TCP.

Algunos ejemplos de uso de UDP son DNS o aplicaciones que puedan tolerar algo de perdida, por ejemplo vídeos o audio. La perdida resultaría en unos *frames* perdidos.

TCP se vuelve apropiado utilizar cuando se requiere de una transmisión de datos confiable para asegurarse que la otra parte tiene los datos de forma correcta.

6. Dificultades encontradas

Durante la realización del trabajo práctico nos encontramos con las siguientes dificultades:

- En un principio se había pensado utilizar bits para representar los flags de ACK y FIN y de esta forma tener un menor *overhead* al agregar nuestro *header*. Por limitaciones presentadas por el lenguaje y las bibliotecas utilizadas, se decidió utilizar bytes en estos campos.
- Se nos presentaron dificultades propias de no conocer Python en profundidad a la hora de poder importar los diferentes módulos que fuimos creando.
- Se nos dificulto la división de tareas inicialmente debido a los niveles de acoplamiento que tienen. Después de lograr armar una versión inicial pudimos organizarnos mejor.
- Se presentaron dificultades para la instalación y correcta ejecución del programa *comcast*.

7. Conclusiones

Se llegaron a las siguientes conclusiones.

- Se logro adquirir conocimiento sobre el funcionamiento de *sockets* UDP y arquitectura Cliente-Servidor expandiendo lo aprendido en materias previas.
- Se logro implementar un protocolo confiable con versiones que implementan *Stop and Wait* y *Go Back N*. Estas versiones se pudieron utilizar correctamente en la subida y descarga de archivos logrando un nivel de separación aceptable entre el protocolo en si y la aplicación.
- Se pudo observar la diferencia en *performance* entre *Stop and Wait* y *Go Back N* viendo diferencias en el orden de la mitad del tiempo a medida que aumenta la perdida de paquetes.
- El protocolo de *Go Back N* tiene lugar para optimizaciones si se guardasen los paquetes posteriores en vez de estar descartándolos.