.UBAfiuba
FACULTAD DE INGENIERÍA

Redes
(TA048) Curso 2

Trabajo Práctico N°1: File Transfer

9 de mayo de 2024

Alan Valdevenito
107585

Mateo Julián Rico
108127

Mariana Galdo Martinez
105658

José Manuel Dieguez
106146

Manuel Rivera Villatte
106041

Índice

1. Introducción

En este informe, abordamos el problema de implementar una aplicación de red para la transferencia de archivos que utiliza UDP como protocolo de capa de transporte para la comunicación entre procesos. El objetivo principal de este proyecto es garantizar que dicha transferencia se efectúe de acuerdo con los principios de transferencia de datos confiable. A lo largo de este documento, se detallarán los procedimientos necesarios para lograr este objetivo.

La aplicación, basada en la arquitectura cliente-servidor, permite la realización de dos operaciones:

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

Para garantizar una transferencia confiable a pesar de las limitaciones inherentes al protocolo UDP, que por su naturaleza es no confiable y no garantiza la entrega de los paquetes, se implementaron dos versiones: una con el protocolo Stop & Wait y otra con el protocolo Go-Back-N.

2. Hipótesis y suposiciones realizadas

A la hora de realizar el trabajo tuvimos en cuenta los siguientes supuestos:

- No hace falta validar los paquetes con checksum ya que UDP se encarga de hacerlo.
- El servidor y los clientes deben usar el mismo protocolo (Stop & Wait o Go-Back-N) para que la aplicación funcione correctamente.
- Al recibir un archivo con un nombre que ya existe en el servidor, se reemplaza el archivo.
- Se pueden procesar archivos de no más de $2GB$.
- El servidor no se queda sin espacio en disco para recibir archivos nuevos, así como también el cliente tiene espacio para descargar el archivo pedido.

3. Implementación

En esta sección, se detallará cómo se realizaron ambos protocolos para manejar los dos tipos de comandos permitidos. Según el comando que se utiliza, el servidor y el cliente cambiarán de roles. Al usar el comando UPLOAD, el cliente enviará los datos del archivos, mientras que el servidor recibe los datos. Cuando se usa DOWNLOAD, estos roles se intercambian. Para simplificar la explicación, se definirá al emisor como aquel que envía los datos del archivo y como receptor al que recibe estos datos.

3.1. Mensaje

Antes de proceder con la implementación de los protocolos para la aplicación, resultó necesario establecer el formato del mensaje, ya que éste se mantendría constante y sería aplicable a ambos protocolos. La estandarización del formato del mensaje garantiza la interoperabilidad y la eficacia en la comunicación entre los procesos, lo cual es fundamental para el correcto funcionamiento de la aplicación.

Definimos que el tamaño del mensaje sería fijo para el header, mientras que la cantidad de bytes que representan los datos del mensaje podría variar, siempre que no se superen los 2048 bytes (2 KB).

El header está conformado por 29 bytes, que representan los siguientes valores:

- Tipo de mensaje (MessageType): es el primer byte del header. Según el tipo de mensaje, tendrá los siguiente valores
  - INSTRUCTION: Si el mensaje es una instrucción, ya sea upload o download, el valor será 1.
  - DATA: Si el mensaje contiene los datos del archivo que se envía, el valor será 2.
  - ERROR: Si el nombre del archivo enviado no es válido, se notifica al cliente que hubo un error y el valor será 3.
  - ACK: Si el mensaje es un ACK, es decir, una confirmación de que se recibió un mensaje, el valor será 4.
  - PORT: Si el mensaje envía información sobre el puerto por donde se comunican el servidor con el cliente, el valor será 5.
  - END: Si el mensaje es para indicar que se quiere cerrar la conexión, el valor será 6.
  - ACK_END: Si el mensaje indica que recibió el mensaje para cerrar la conexión y se quiere enviar la confirmación, el valor será 7.
- Número de secuencia: son 4 bytes que indican el número de secuencia del paquete. Este valor se utilizará para asegurar la confiabilidad de la red y se explicará su uso con más detalle en la explicación de la implementación de los protocolos.
- Nombre del archivo: son 20 bytes que se utilizan para enviar el nombre del archivo que se quiere descargar o enviar.
- Tamaño de la data: son 4 bytes que indican el largo en bytes de los datos que se envían.

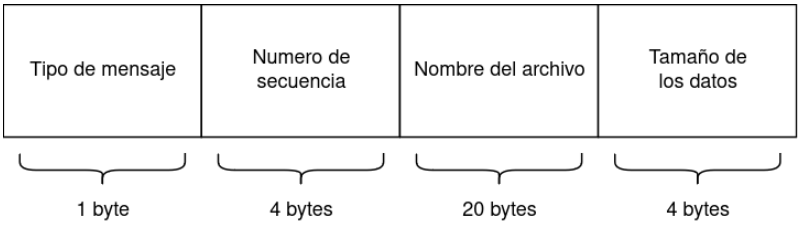

Figura 1: Formato de Mensaje

3.2. Inicio de conexión

Para ambos protocolos, el inicio de la conexión funciona de la misma manera, pero cambia según el comando a utilizar.

Para iniciar la conexión se realiza un 2-way-handshake, donde el primer mensaje es tipo INSTRUCTION, tal como fue mencionado en la Sección 3.1. Este mensaje contiene el comando que se utilizará en su campo de datos y el nombre del archivo que se quiera enviar o descargar. Este paso se mantiene igual para ambos comandos, pero la respuesta del servidor será diferente.

En el caso del comando UPLOAD, el cliente esperará un mensaje PORT, ya que cada cliente se comunicará con el servidor por puertos distintos una vez que se establece la conexión. El uso de puertos para manejar múltiples clientes se explicará en detalle en la Sección 3.6.

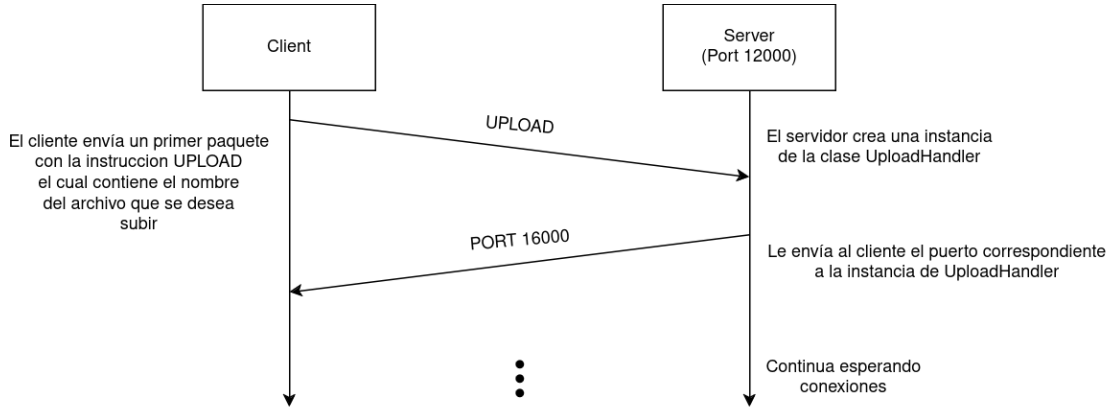

Figura 2: Inicio de conexión en UPLOAD

Para el comando DOWNLOAD, una vez enviado el mensaje con la instrucción, el cliente espera recibir un mensaje con los datos del archivo pedido. Es decir, recibirá un mensaje de tipo DATA.

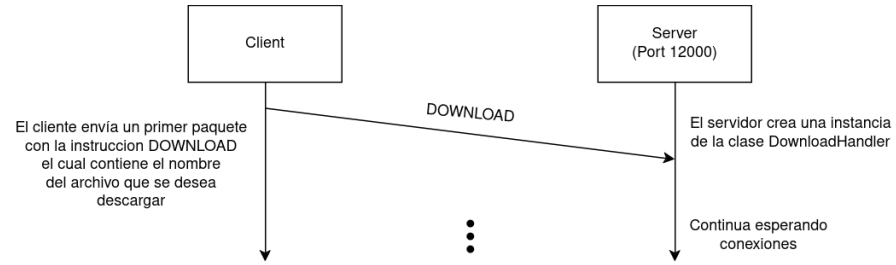

Figura 3: Inicio de conexión en DOWNLOAD

3.2.1. Manejo de pérdida de paquetes

Para manejar la pérdida de paquetes, se decidió por hacer uso de timeouts. Se definió un timeout de 100ms para esperar la respuesta del servidor, ya sea un mensaje de tipo PORT o de tipo DATA. Una vez transcurrido dicho tiempo, el cliente considerará que el paquete se perdió y lo reenviará. Este ciclo se repetirá hasta que llegue una respuesta del servidor o hasta que haya intentado enviar el mensaje 10 veces. Una vez alcanzado este número de intentos, el cliente asumirá que el servidor no está conectado para recibir mensajes.

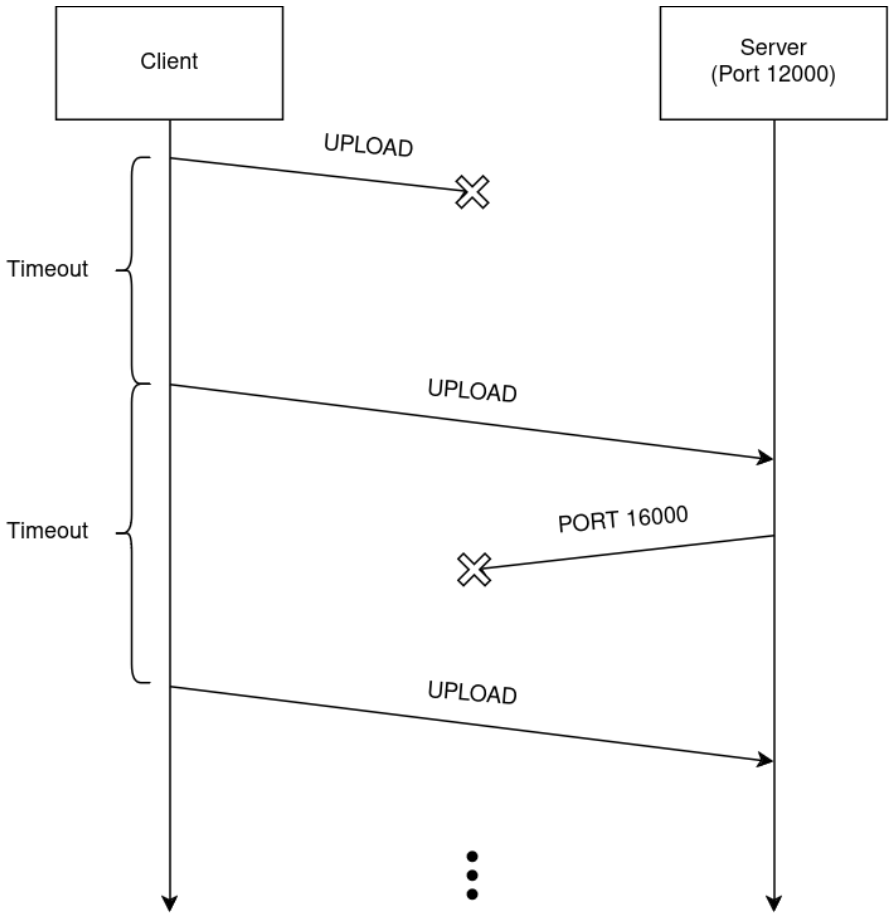

Figura 4: Inicio de conexión en UPLOAD con pérdida

3.3. Cierre de conexión

Para finalizar la comunicación, se envía un mensaje de tipo END, tal como se mencionó en la Sección 3.1. Tanto el cliente como el servidor pueden empezar el proceso de finalizar la comunicación. En el caso del download, será el servidor el que envíe primero el mensaje END, mientras que al usar el comando upload, el cliente enviará el mensaje END una vez que ya haya mandado todo el archivo.

Se explicarán los pasos que forman parte del proceso del cierre de la conexión considerando el caso en que el comando que se envía es UPLOAD. Una vez enviado el mensaje END, el cliente esperará que el servidor responda con un ACK_END, que es un tipo de mensaje para confirmar que se recibió el mensaje END. A su vez, el servidor también enviará su propio mensaje END para finalizar la conexión, por lo que el cliente esperará a que llegue este mensaje y confirmarle al servidor que lo recibió enviando un ACK_END. Los pasos al usar el comando DOWNLOAD son los mismos. La única diferencia es que el servidor envía el primer mensaje END.

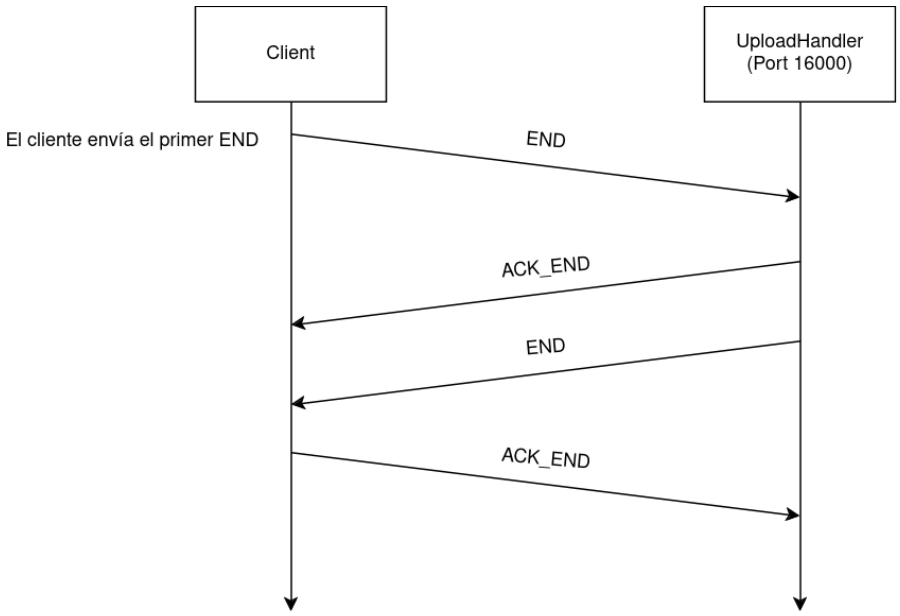

Figura 5: Cierre de conexión en UPLOAD

3.3.1. Manejo de pérdida de paquetes

Puede suceder el caso en que alguno de estos mensajes que se transmiten durante el cierre de la conexión se pierda. Al cierre de la conexión, será el emisor aquel que inicia el proceso de cierre de la comunicación y el receptor quien reciba el primer mensaje de cierre de conexión. Si se pierde el mensaje END, en ambos casos se reenviará al menos 10 veces. En el caso del receptor, no solo se reenviará el mensaje END, sino también el ACK_END del END recibido, ya que se considera el caso en que se haya perdido ese ACK y el emisor todavía lo esté esperando. Una vez que el emisor haya recibido el ACK_END de su mensaje enviado, iniciará un timeout de 3 segundos para recibir el END del receptor. Si no recibe nada, se asume que el paquete se perdió y el receptor se desconectó.

3.4. Stop & Wait

El primer protocolo que se utilizará para asegurarnos que la transmisión sea confiable es Stop and Wait. El funcionamiento principal de este protocolo es enviar un mensaje y esperar a la confirmación (ACK) del mensaje antes de enviar el próximo mensaje.

Para esta implementación, una vez establecida la conexión con el servidor, el primer paquete que se enviará tendrá número de secuencia 0. Si este paquete se recibió correctamente, el receptor, ya sea el cliente cuando se utiliza el comando DOWNLOAD, o el servidor cuando se ejecuta UPLOAD, enviará un ACK con número de secuencia 0. Es decir que, para cada paquete con número de secuencia i enviado y recibido correctamente, el emisor espera recibir un ACK con número de secuencia i.

En el caso de que haya pérdida de paquetes al enviar los datos del archivo, al usar el comando UPLOAD, por ejemplo, el cliente se quedará esperando 50ms hasta que se termine el tiempo de espera. Al terminarse el tiempo, empieza el ciclo de enviar el paquete y vuelve a esperar el ACK correspondiente. Esto mismo sucedería del lado del servidor al usar el comando DOWNLOAD.

Para la pérdida de ACKs que envía el receptor, el receptor no tiene en cuenta si el ACK se perdió o no. Es el emisor el que asume que se perdió el paquete y lo vuelve a enviar una vez que se agote el tiempo de espera.

Al analizar ambos casos de pérdida, tanto del lado del emisor y del lado del receptor, podemos

definir que es responsabilidad del emisor que lleguen los paquetes, ya sea si se perdió un paquete con los datos del archivo o un ACK.

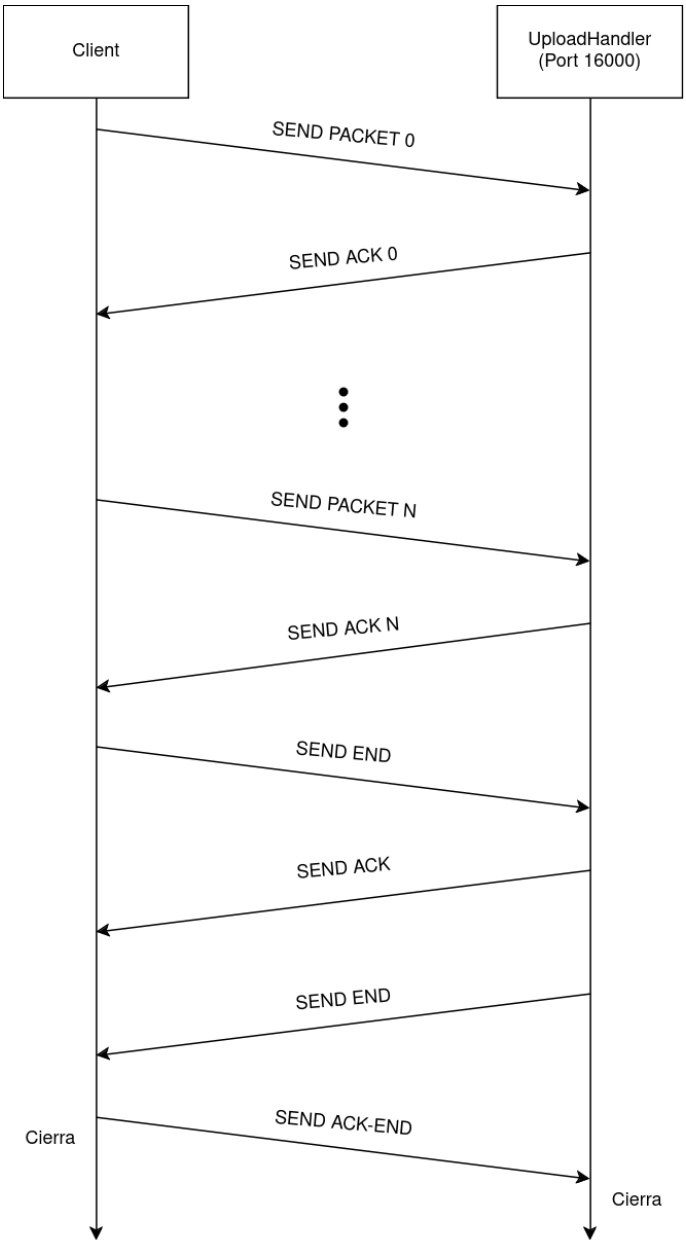

Figura 6: Stop & Wait

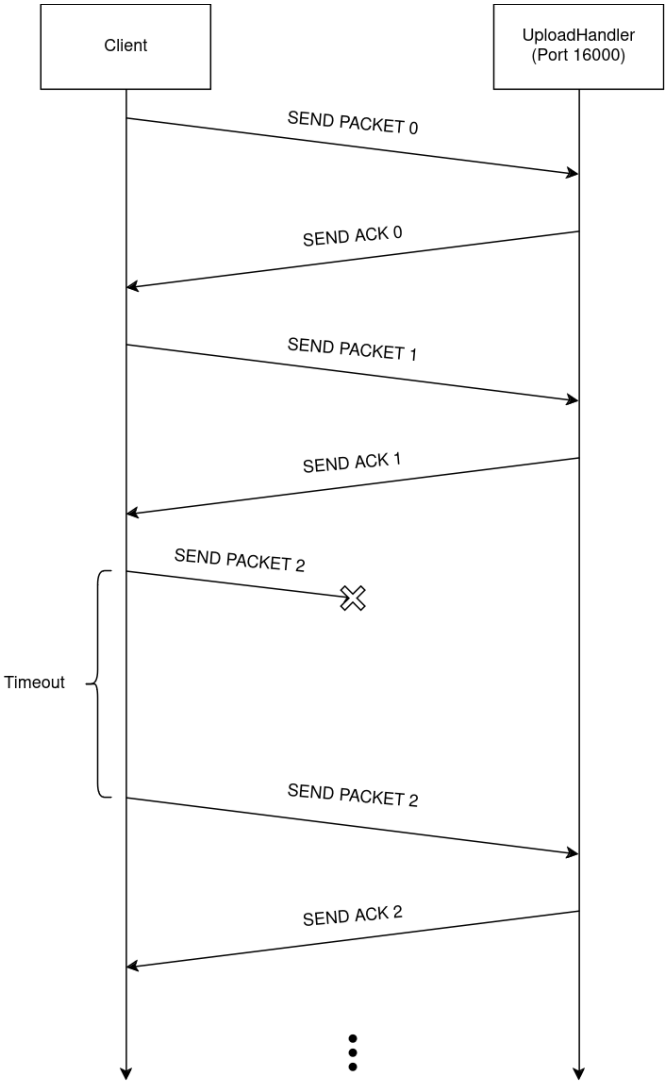

Figura 7: Stop & Wait con pérdida de paquete

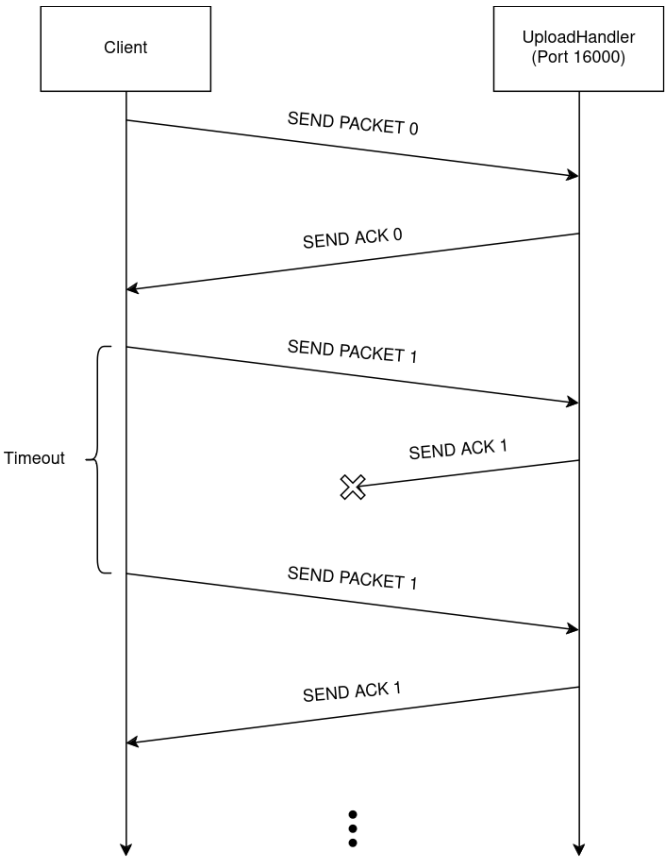

Figura 8: Stop & Wait con pérdida de ACK

3.5. Go-Back-N

Este protocolo, a comparación de Stop & Wait, nos permite enviar una cantidad de paquetes sin la necesidad de esperar a un ACK. Se utiliza una ventana deslizante que, en nuestro caso, definimos un máximo de 10 a la ventana, que define los paquetes que pueden ser enviados sin esperar los ACK. La ventana deslizante mantiene una referencia al número de secuencia del paquete más antiguo que fue enviado pero no fue confirmado (ACKed) llamado base, y al siguiente número de secuencia a enviar. Estos números de secuencias nos permiten mantener la comunicación continua entre los procesos, sin la necesidad de esperar en el envío de cada paquete. A continuación, se explicará en detalle la implementación del protocolo.

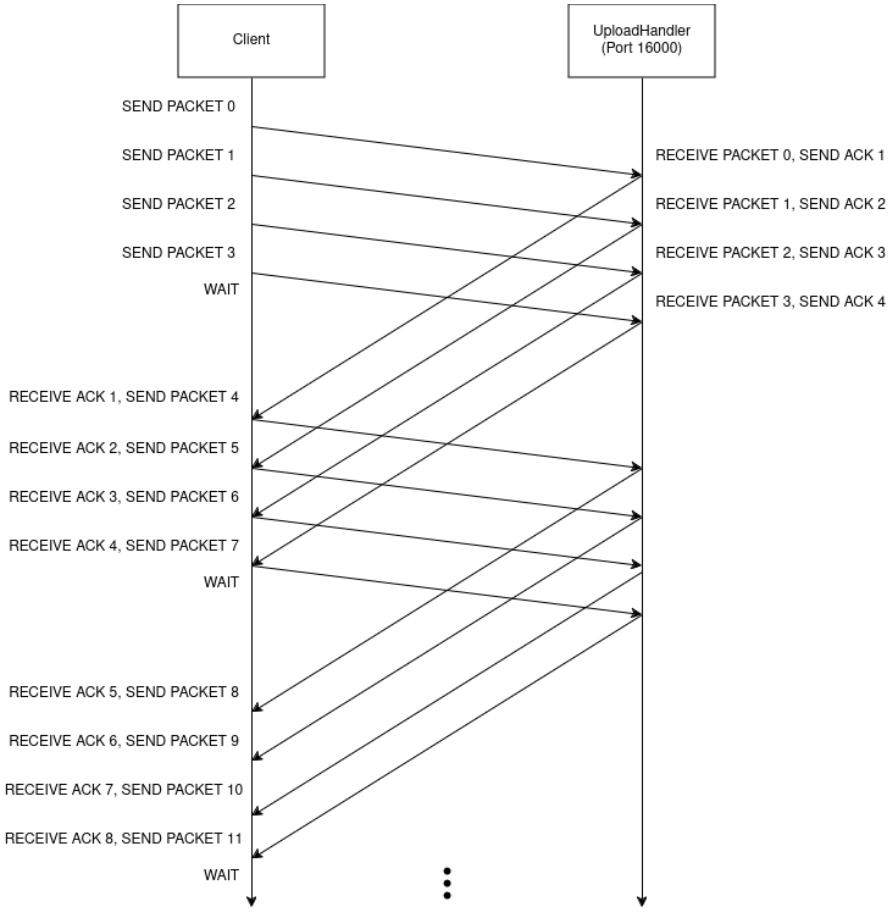

Figura 9: Go-Back-N

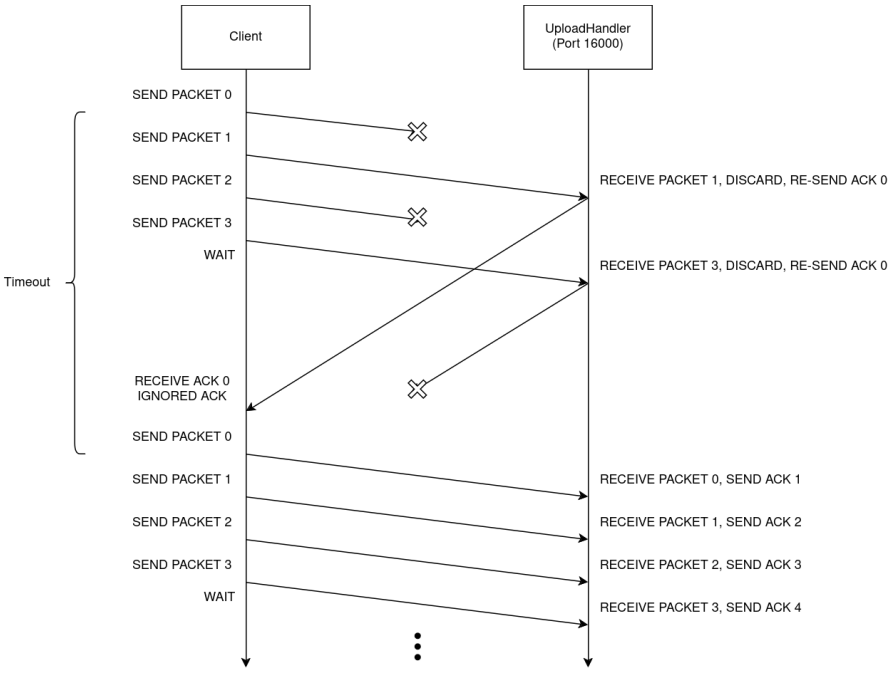

Figura 10: Go-Back-N con pérdida

3.5.1. Manejo de ACK

Como GBN es un protocolo de pipelining, tuvimos que encarar el mecanismo de ACK de forma distinta a Stop & Wait. Como queremos que el emisor envíe paquetes hasta llenar la ventana, y a la vez que esté a la espera de un ACK por parte del receiver, creamos una función no bloqueante, `recv_ack`, que es llamada dentro la función `send`, al inicio de cada operación. Además, en el caso de que se llame a `send` cuando la ventana está llena, se debe esperar a que se haga espacio; es decir, que llegue un ACK de un paquete que estaba in-flight. Para lograr esto, se utiliza esta función `recv_ack` en un ciclo para esperar la llegada de este ACK. Pero, ¿Qué sucede si ese ACK no llega? Veremos qué hacemos en ese caso en la sección "Manejo de Timeout".

El cliente, al mandar un paquete, para saber que ese paquete fue recibido correctamente, espera un ACK que sea igual al próximo número de secuencia esperado. Esto fue implementado así para considerar el caso en que el paquete con ACK 0 se pierde. Si esto sucede, y el cliente sigue enviando paquetes, el Handler enviará el ACK 0 por todos los paquetes que recibió fuera de orden. Si nosotros consideráramos que el número del ACK representa el número de secuencia del paquete que fue recibido correctamente, entonces el cliente creería que el paquete 0 llegó al Handler, cuando ese no fue el caso. Por lo tanto, el cliente espera un ACK con un valor mayor en 1 del número de secuencia del paquete.

3.5.2. Manejo de Timeout

El protocolo maneja el tiempo con una variable llamada `lastackreceived`. GBN requiere que un timer se corra para el paquete in-flight (enviado pero sin ACK) más antiguo. En el caso de que ocurra un Timeout, se deben retransmitir todos los paquetes que estén en el rango entre el paquete que provocó el timeout, y el último paquete enviado. En términos técnicos de nuestra aplicación, el índice del paquete más antiguo enviado no-ACKeado se denomina base, y el índice del paquete más recientemente enviado se llama signumsec, por lo tanto en caso de timeout se deben retransmitir todos los paquetes desde base hasta signumsec.

El timer lo implementamos utilizando la librería `time` de Python. En `lastackreceived` guardamos el valor de time actual, y para verificar un timeout comparamos ese valor con el valor de time presente, y si la diferencia entre las dos mediciones supera un umbral, provoca un timeout. La variable se llama `lastackreceived` porque el timer debe ser restablecido cada vez que recibimos un ACK, ya que el paquete base cambia. Por lo tanto, el timer puede ser interpretado como el tiempo en el que fue recibido el último ACK hasta el momento. Finalmente, el segundo caso en el que el timer es restablecido es luego de un timeout, ya que, como se explicó anteriormente, se deben retransmitir paquetes. Por lo tanto, el timer comienza nuevamente. Veamos cómo funciona el mecanismo de retransmisión de paquetes: ¿De dónde salen los paquetes ya transmitidos para que puedan volver a transmitirse?

3.5.3. Buffering de Paquetes y Retransmisión

Para poder implementar un sistema de retransmisión de paquetes, es necesaria una estructura que almacene paquetes siguiendo un criterio especial, para poder recuperar los paquetes según su número de secuencia. Para esto, el protocolo almacena como pares clave-valor (número de secuencia, mensaje) en un diccionario de Python cuando un paquete es enviado por la función `send`. Cuando se debe retransmitir una serie de paquetes, se iteran los números de secuencia desde `base` hasta `signumsec`, y se recuperan los mensajes correspondientes del diccionario y se retransmiten.

3.6. Manejo de concurrencia

Para manejar las requests de clientes decidimos modularizar el servidor haciendo uso de la concurrencia.

En nuestra implementación el servidor sólo recibe requests de tipo UPLOAD y DOWNLOAD, y por cada request lanza un nuevo hilo de ejecución que se encarga de atender esa request (enviando

o recibiendo el archivo correspondiente).

Además, para evitar que el hilo principal reciba todos los paquetes y los reenvíe a los hilos que se comunican con los clientes, utilizaremos distintos puertos para cada comunicación con un cliente, así como se mencionó en la Sección 3.2. Al lanzar el nuevo hilo de ejecución, se crea el socket con un puerto aleatorio que no esté en uso.

De esta forma, conseguimos atender múltiples clientes de forma concurrente, ya que mientras un hilo de ejecución le transfiere un archivo a un cliente, el hilo principal del servidor puede estar esperando nuevos clientes.

4. Pruebas

4.1. Stop & Wait: Upload sin pérdida de paquetes

4.1.1. Inicio de conexion y envio de los primeros paquetes

```
Receiving packet with sequence number 0 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 0

Writing data

Receiving packet with sequence number 1 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 1

Writing data

Receiving packet with sequence number 2 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 2

Writing data

Receiving packet with sequence number 3 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 3

Writing data

Receiving packet with sequence number 4 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 4

Writing data

Receiving packet with sequence number 5 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 5

Writing data
```

Figura 11: Servidor

```
Sending packet with sequence number 0 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (0) == Got (0).

Sending packet with sequence number 1 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (1) == Got (1).

Sending packet with sequence number 2 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (2) == Got (2).

Sending packet with sequence number 3 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (3) == Got (3).

Sending packet with sequence number 4 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (4) == Got (4).

Sending packet with sequence number 5 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (5) == Got (5).
```

Figura 12: Cliente

Servidor: Se queda bloqueado esperando los paquetes y envía los ACKs correspondientes.

Cliente: Envía paquetes y se queda bloqueado esperando los ACKs correspondientes.

4.1.2. Fin de conexion

```
Receiving packet with sequence number 63 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 63

Sending END to ('127.0.0.1', 48635)

Receiving ACK-END

Upload finished.
```

Figura 13: Servidor

```
Sending packet with sequence number 63 to ('127.0.0.1', 56262)
Receiving ACK from ('127.0.0.1', 56262)
Sequence number correct. Expected (63) == Got (63).

Waiting END

Receiving END

Sending ACK-END to ('127.0.0.1', 56262)

Client finished
```

Figura 14: Cliente

Servidor: Recibe el paquete con número de secuencia 63 que se corresponde con la instrucción de tipo END. Luego envía un paquete con la instrucción de tipo END y se queda bloqueado esperando la instrucción de tipo ACK-END. Finalmente cuando recibe el ACK-END termina su ejecución.

Cliente: El cliente envía el paquete con número de secuencia 63 que se corresponde con la instrucción de tipo END. Luego se queda bloqueado esperando el END de parte del servidor. Finalmente cuando recibe el END, envía el ACK-END y termina su ejecución.

4.2. Stop & Wait: Upload con pérdida de paquetes del 10 %

4.2.1. Inicio de conexion y envio de los primeros paquetes

```
Receiving packet with sequence number 0 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 0

Writing data

Receiving packet with sequence number 1 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 1

Writing data

Receiving packet with sequence number 2 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 2

Writing data

Receiving packet with sequence number 3 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 3

Writing data

Receiving packet with sequence number 4 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 4

Writing data

Receiving packet with sequence number 5 from ('127.0.0.1', 48635)
Sending ACK to ('127.0.0.1', 48635) with sequence number 5

Writing data
```

Figura 15: Servidor

```
Sending packet with sequence number 0 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (0) == Got (0).

Sending packet with sequence number 1 to ('127.0.0.1', 41988)
Sending packet with sequence number 1 to ('127.0.0.1', 41988)
Sending packet with sequence number 1 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (1) == Got (1).

Sending packet with sequence number 2 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (2) == Got (2).

Sending packet with sequence number 3 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (3) == Got (3).

Sending packet with sequence number 4 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (4) == Got (4).

Sending packet with sequence number 5 to ('127.0.0.1', 41988)
Receiving ACK from ('127.0.0.1', 41988)
Sequence number correct. Expected (5) == Got (5).
```

Figura 16: Cliente

Servidor: El servidor se queda bloqueado esperando los paquetes y envía los ACKs correspondientes.

Cliente: El cliente envía el paquete con número de secuencia 1 y se queda bloqueado esperando el ACK correspondiente. Este paquete se pierde, se da un timeout y se reenvía el paquete con número de secuencia 1. El paquete con número de secuencia 1 se pierde dos veces. Al tercer reenvío del paquete con número de secuencia 1 es que se envía sin pérdida y se recibe el ACK correspondiente.

4.3. Go-Back-N: Download sin pérdida de paquetes

Inicio de conexion y envio de los primeros paquetes

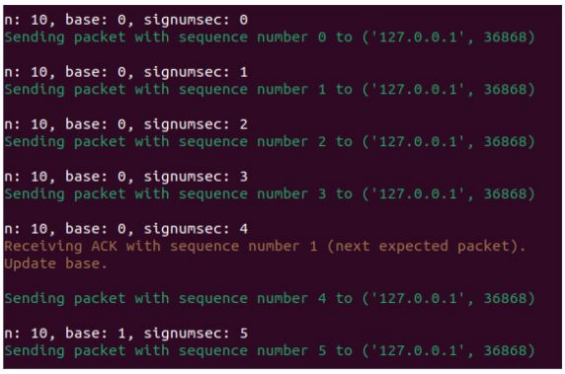

Figura 17: Servidor

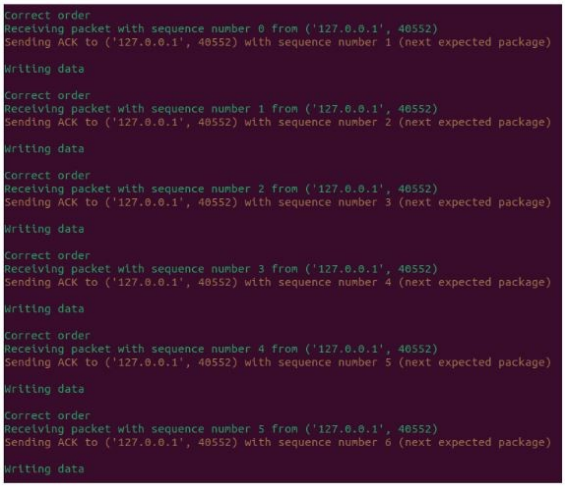

Figura 18: Cliente

Servidor: El servidor tiene una ventana de tamaño 10 por lo tanto puede enviar hasta 10 paquetes sin esperar los ACKs. Podemos ver que en el medio del envío de paquetes recibe el ACK del paquete con número de secuencia 0 y mueve su ventana.

Ventana llena

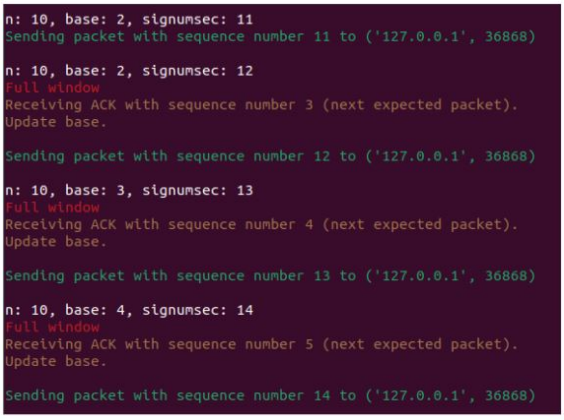

Figura 19: Servidor

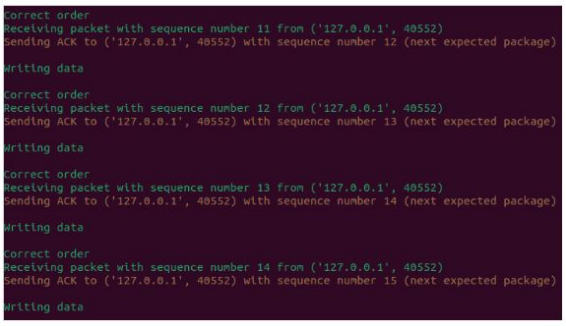

Figura 20: Cliente

Servidor: En el momento que intenta enviar el paquete con número de secuencia 12 la ventana está llena. Lo que sucede en este momento es que el servidor se queda bloqueado esperando recibir un ACK que le permita mover su ventana. Luego, recibe el ACK con número de secuencia 3, mueve su ventana y envía el paquete con número de secuencia 12.

Fin de conexión

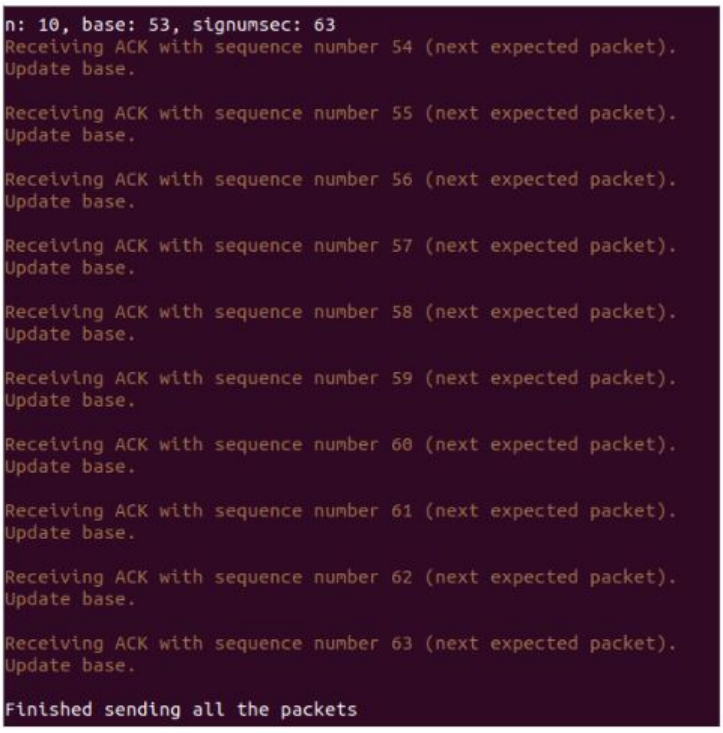

Figura 21: Servidor

El servidor antes de enviar el paquete con la instrucción de tipo END se queda bloqueado esperando recibir todos los ACKs restantes y en caso de ser necesario hace retransmisiones

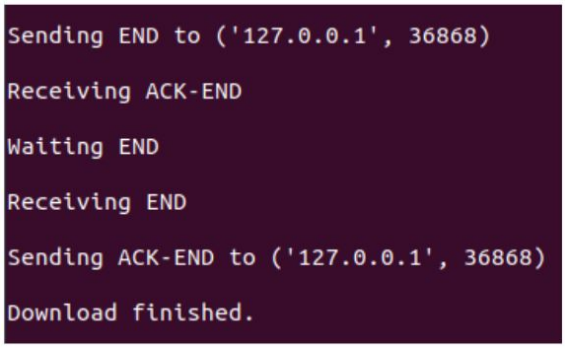

Figura 22: Servidor

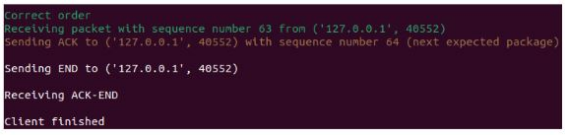

Figura 23: Cliente

Servidor: (1) Envía el paquete con la instrucción de tipo END y se queda bloqueado esperando el ACK correspondiente. (3) Recibe el paquete con la instrucción de tipo ACK-END y se queda bloqueado esperando la instrucción de tipo END. (5) Recibe el paquete con la instrucción de tipo END, envía un paquete con la instrucción de tipo ACK-END y termina su ejecución.

Cliente: (2) El cliente recibe el paquete con número de secuencia 63 que se corresponde con la instrucción de tipo END y envía el ACK correspondiente. (4) Luego envía un paquete con la instrucción de tipo END y se queda bloqueado esperando la instrucción de tipo ACK-END. (6) Finalmente cuando recibe el ACK-END termina su ejecución.

4.4. Go-Back-N: Download con pérdida de paquetes del 10 %

Inicio de conexión y envío de los primeros paquetes

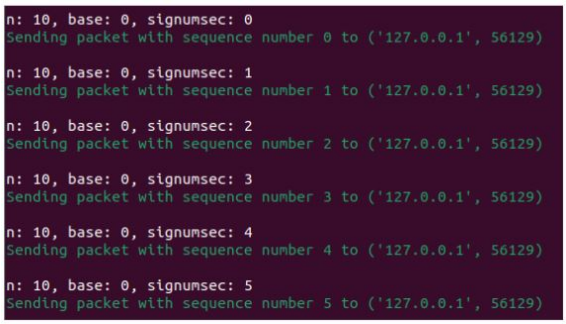

Figura 24: Servidor

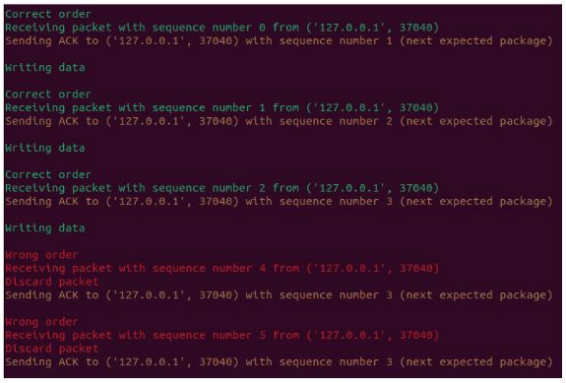

Figura 25: Cliente

Servidor: El servidor envía los paquetes con número de secuencia desde el 0 al 12.

No se muestra en la figura pero antes de enviar el paquete con número de secuencia 6 recibe un ACK con número de secuencia 2 y antes de enviar el paquete con número de secuencia 10 recibe un ACK con número de secuencia 3.

Cliente: El cliente comienza recibiendo los paquetes en orden correcto. Recibirlos en orden correcto quiere decir que no hay pérdida de paquetes. Podemos ver que se pierde el paquete con número de secuencia 3 ya que luego del paquete con número de secuencia 2 recibe el 4. Luego, se descarta el paquete recibido fuera de orden. Se descartan los paquetes con número de secuencia desde el 4 al 12.

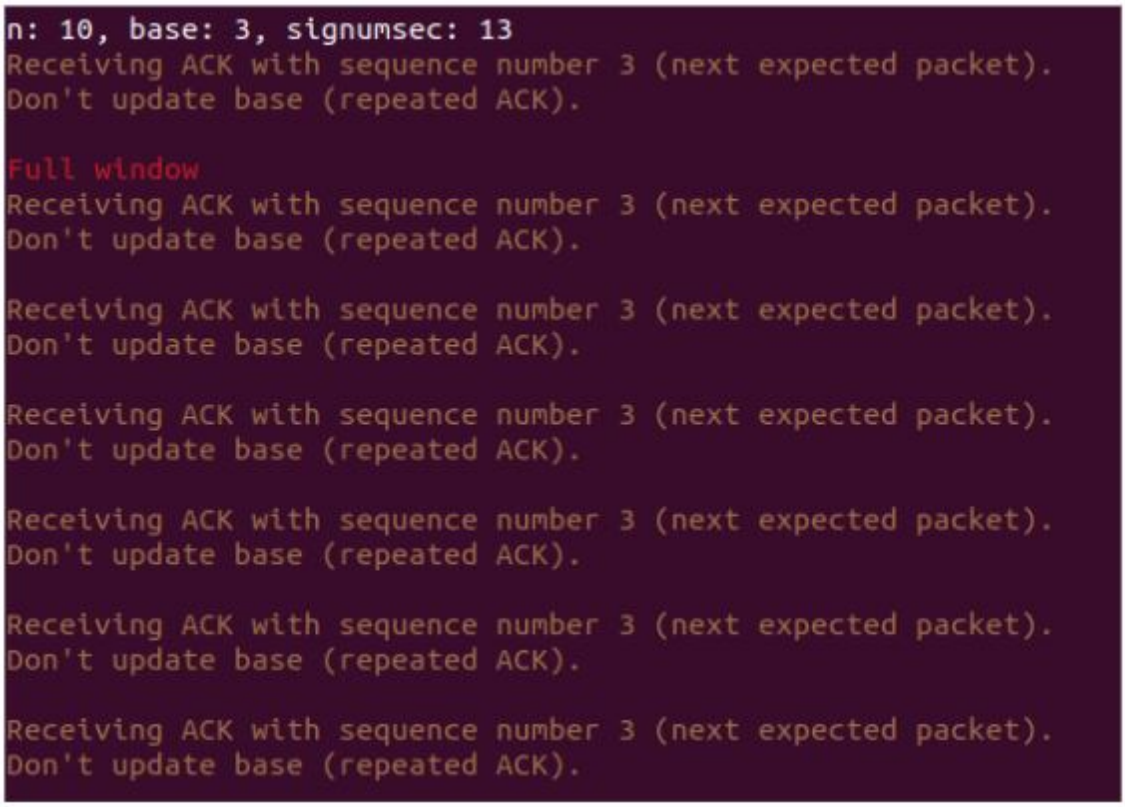

Figura 26: Servidor

Como el servidor antes recibió 3 ACKs puede enviar hasta el paquete con número de secuencia 12. Cuando intenta enviar el paquete con número de secuencia 13 se encuentra con que la ventana está llena y se queda bloqueado esperando recibir ACKs que le permitan mover su ventana. Sin embargo, como recibe ACKs con número de secuencia 3 y el próximo paquete a enviar desde el lado del servidor es el paquete con número de secuencia 13 no mueve la ventana.

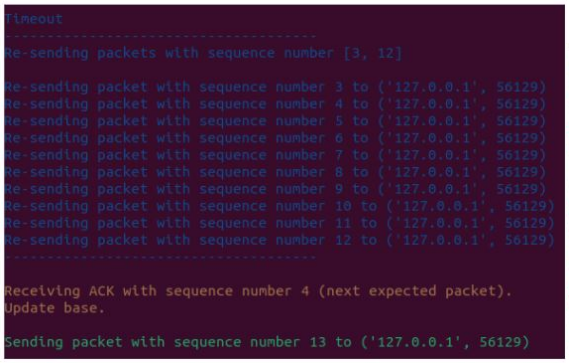

Figura 27: Servidor

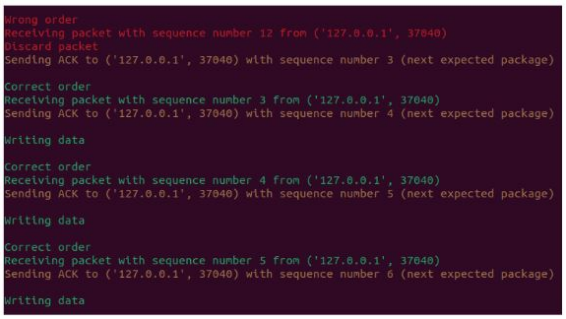

Figura 28: Cliente

Servidor: Se da un timeout y se reenvía el paquete con número de secuencia 3 hasta el paquete con número de secuencia 12.

Cliente: El cliente recibe desde el paquete con número de secuencia 3 hasta el paquete con número de secuencia 12.

4.5. Gráficos de rendimiento

Para cada protocolo, y cada operación, se realizaron pruebas transifiriendo distintos archivos, de tamaños 1MB, 2MB, 4MB, 8MB y 16MB, con y sin pérdida. Las pruebas pueden correrse ejecutando `python3 pruebas.py`. A continuación se muestran gráficos en los cuales se puede ver el rendimiento de cada protocolo.

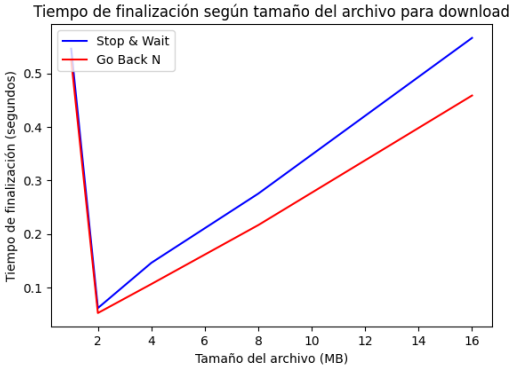

Figura 29: Comparación de los tiempos de descarga para cada protocolo sin pérdida de paquetes

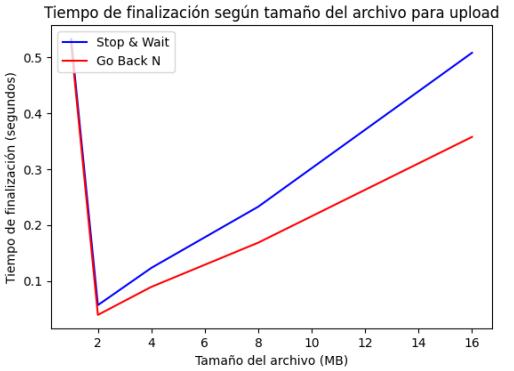

Figura 30: Comparación de los tiempos de subida para cada protocolo sin pérdida de paquetes

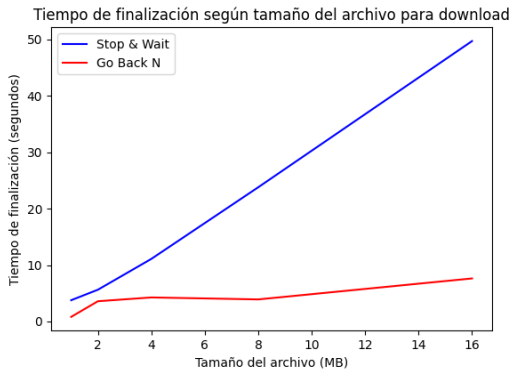

Figura 31: Comparación de los tiempos de descarga para cada protocolo con 10 % de pérdida de paquetes

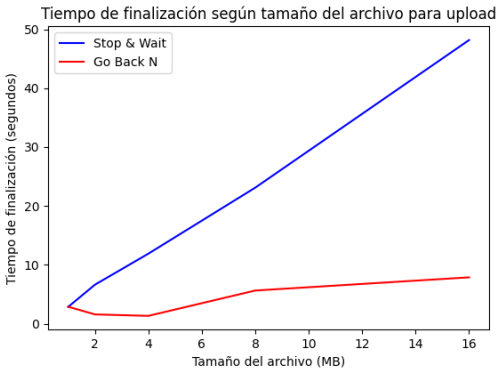

Figura 32: Comparación de los tiempos de subida para cada protocolo con 10 % de pérdida de paquetes

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es una estructura para manejar aplicaciones cuyo funcionamiento se separa en dos categorías fundamentales:

Servidores, es decir, proveedores de un servicio o recurso, y **clientes**, entidades que solicitan servicios o recursos. Tanto servidores como clientes son **hosts**. Los servidores suelen ser procesos que están constantemente a la espera de solicitudes de conexión por parte de los clientes, y por lo tanto, deben poder manejar varios clientes de forma concurrente.

Al arribo de una solicitud de conexión nueva, el servidor debe ser capaz de crear una nueva interfaz de comunicación dedicada con ese cliente, es decir, un hilo de ejecución que contenga el socket por el que se enviarán y recibirán mensajes con ese cliente particular. En el caso de utilizar sockets UDP en el servidor, en lugar de tener un hilo principal que redirrecione los mensajes a cada hilo de ejecución, simplemente se determina un nuevo **puerto** por el que pasarán los mensajes correspondientes a ese cliente. De esta forma, el socket principal del servidor se puede comprometer a esperar nuevas solicitudes por parte de los clientes, y luego delegar la responsabilidad de realizar la comunicación con el cliente a un hilo de ejecución separado.

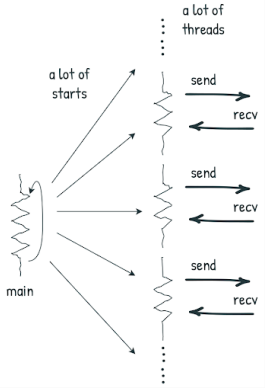

Figura 33: Esquema de la Concurrencia de un modelo Cliente-Servidor

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

Un protocolo de capa de aplicación define cómo se envían los mensajes a los procesos de las aplicaciones, aún si corren en distintos dispositivos. Según Jim Kurose, estos protocolos definen cuatro puntos concretos:

- Los tipos de mensajes intercambiados (por ejemplo, mensajes de request, o de respuesta).
- La sintaxis de los varios tipos de mensajes (cómo se componen, delimitación de sus campos).
- La semántica de los campos (es decir, el significado de la información en los mismos).
- Las reglas para determinar cuándo y cómo un proceso envía y responde a mensajes.

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

El **Servidor**, desde su hilo y socket principales, estará pendiente de recibir peticiones de nuevos clientes, los cuales especifican qué tipo de operación desean realizar: *upload* o *download*.

Esto se hace en una etapa preliminar a la transmisión del archivo en sí, donde el cliente envía al servidor un mensaje de tipo `INSTRUCTION` (donde detalla el tipo de operación que quiere realizar), que el Servidor necesita para inicializar el *handler* apropiado.

Para poder manejar esa solicitud de manera concurrente, se lanza un hilo handler particular para ese tipo de operación. Más precisamente, existen dos tipos de handler: *Upload Handler* y *Download Handler*. Estos usan el protocolo deseado por el usuario (servidor), con el cual interactúa con el cliente a través de un nuevo puerto exclusivo asociado a la nueva sesión generada. La clase *HandleFactory* inicializa el handler apropiado para cada nueva solicitud. La Figura 33 muestra esta función en detalle.

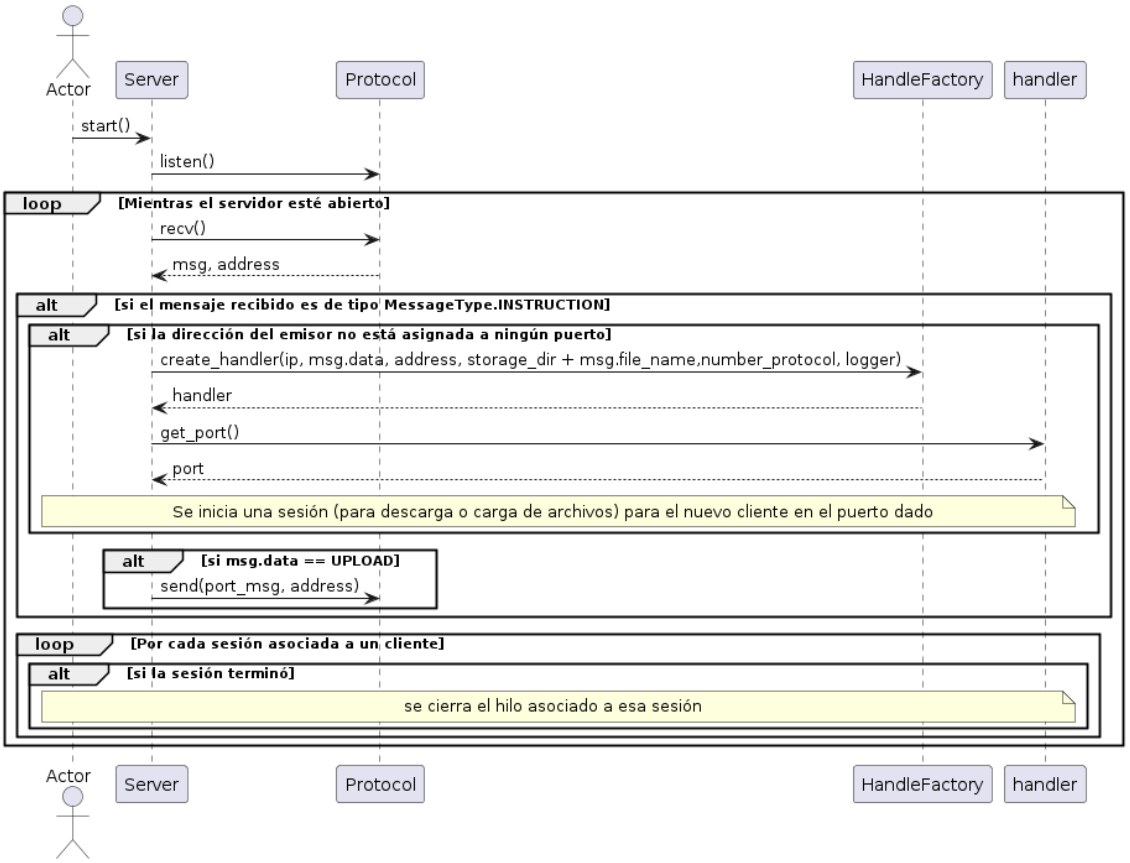

Figura 34: Ciclo de vida del Servidor

Upload

La lógica implementada para atender a un cliente dentro de UploadHandler se basa en que mientras no se reciba un mensaje END (es decir, mientras un cliente no terminó de transmitir un archivo dado), se reciben mensajes por parte del cliente, cargados de datos asociados a dicho archivo, los cuales se van escribiendo en un nuevo archivo dentro de la carpeta de **storage** del lado del servidor. Una vez que se recibe un mensaje END, podemos asegurarnos de que se terminó de recibir el archivo, por lo que enviamos un mensaje similar al cliente, para así dar por finalizada la sesión.

Esta secuencia de pasos se puede asociar a la función de **download** en el lado del cliente, ya que la lógica es la misma.

Las Figuras 34 y 35 muestran esta secuencia para el Servidor y para el Cliente respectivamente.

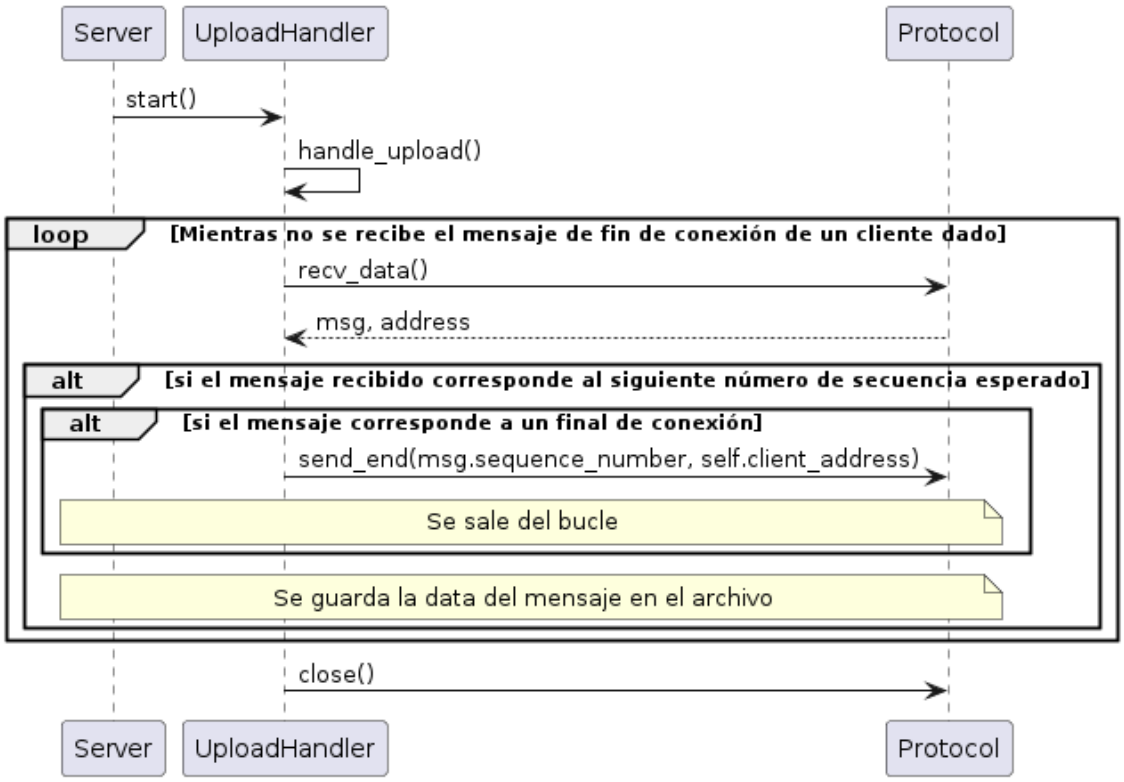

Figura 35: UploadHandler del Servidor

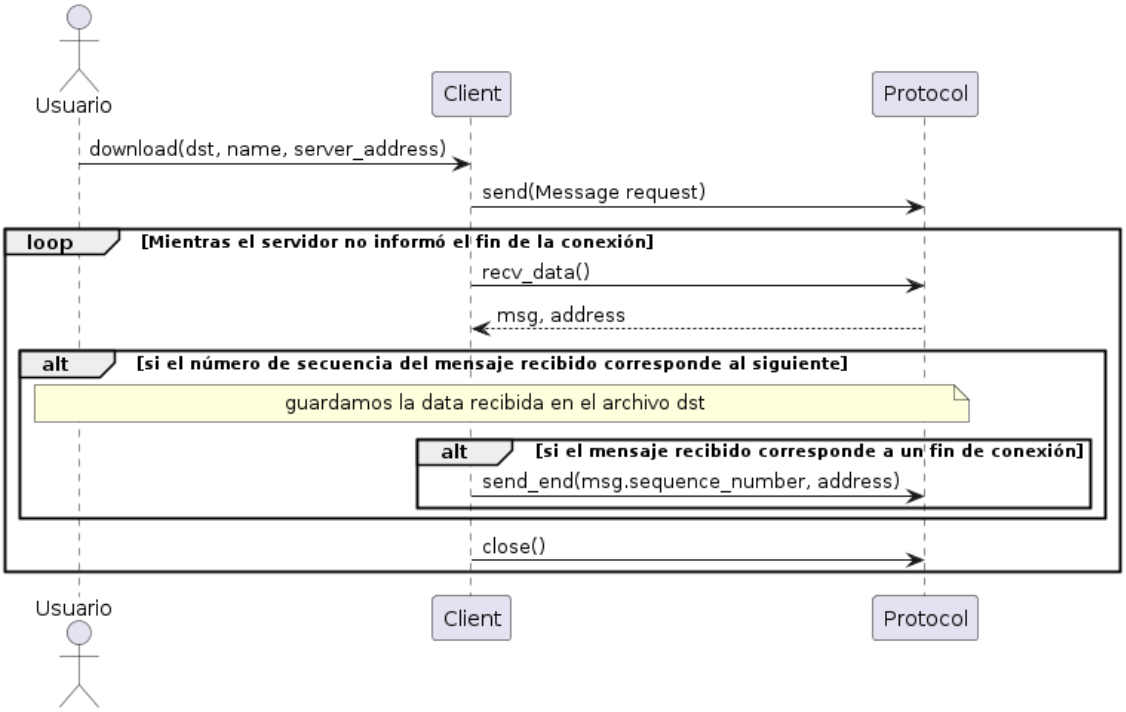

Figura 36: Download del Cliente

Download

Igualmente simétricas son las operaciones `handle_download` de *DownloadHandler* y `upload` del cliente: van leyendo bytes de un archivo y los envían hasta terminar el archivo, y luego envían un mensaje de END, y se espera a que la otra parte envíe el correspondiente ACK_END. En las Figuras 36 y 37 se pueden ver la secuencia de operaciones de forma visual para el Servidor y el Cliente respectivamente.

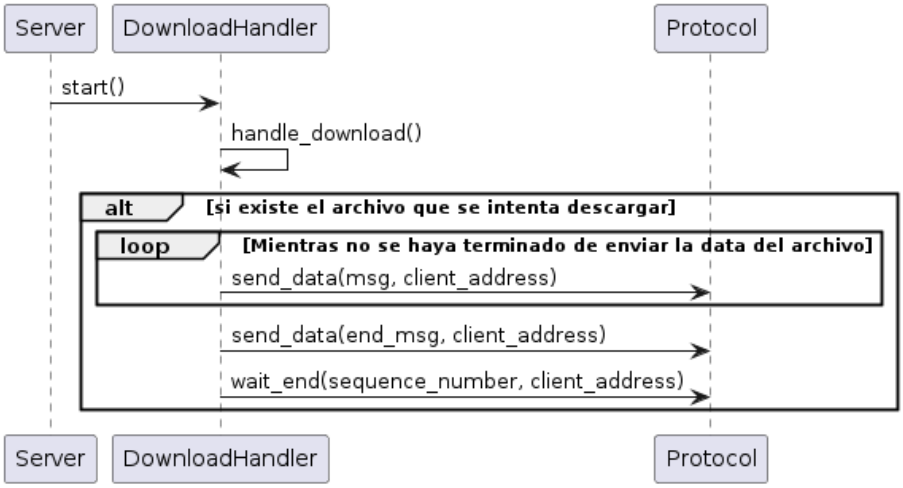

Figura 37: DownloadHandler del Servidor

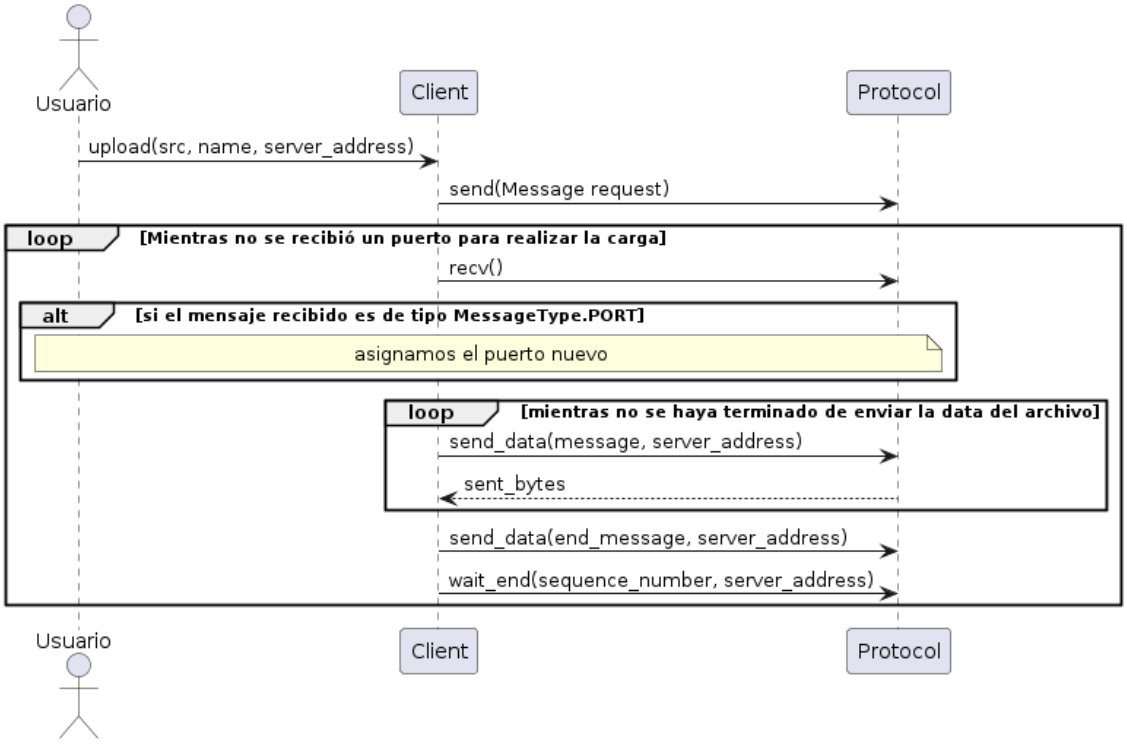

Figura 38: Upload del Cliente

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

TCP y UDP son dos protocolos de la capa de transporte, pero son fundamentalmente diferentes en varios aspectos:

- **Confiabilidad**: UDP es un protocolo que realiza el mínimo indispensable para funcionar correctamente como un protocolo de capa de transporte. Si una aplicación decide utilizar UDP, es como si se comunicara casi directamente con IP, aparte de la multiplexación y desmultiplexación. Por lo tanto, como el protocolo IP realiza su mejor esfuerzo para transmitir datagramas, UDP también. Es un protocolo no confiable.

  Al contrario, el protocolo TCP toma medidas para asegurar confiabilidad en los datos transmitidos, por medio de conexiones (como se ve más abajo), y mecanismos de control de congestión. UDP puede servir para aplicaciones que demanden una velocidad de transmisión mínima y puedan tolerar un cierto grado de pérdida de datos.

- **Conectividad**: TCP es un protocolo basado en conexiones. Antes de siquiera poder transmitir datos, utiliza un sistema de handshaking para establecer un vínculo uno a uno entre dos hosts que van a intercambiar mensajes. UDP, por el otro lado, mientras sepa a qué puerto corresponde un mensaje, lo envía o recibe sin etapas preliminares. Esto es claramente más rápido que el sistema de TCP, pero con el costo de menor confiabilidad. Es por esto que los protocolos de capa de aplicación que necesitan confiabilidad en la capa de transporte, como HTTP, optan por usar TCP, mientras que otros como DNS usan UDP para poder concentrarse en velocidad.

  Es importante destacar que las aplicaciones, a pesar de implementar UDP como protocolo de transporte, pueden implementar como parte de la aplicación funcionalidades que ayuden a lograr una mayor confiabilidad, como es el caso de Google Chrome y su protocolo QUIC.

6. Dificultades encontradas

- Interpretación de los ACKs al momento de implementar GBN.
- Cierre de conexión con un timeout para evitar el problema conocido como 'Problema de los generales bizantinos'. En este problema, los generales necesitan enviar mensajes entre sí para tomar una decisión sobre atacar o retirarse. La pérdida de mensajes (presencia de generales traidores que no envían mensajes) puede hacer que la comunicación sea inconsistente o incompleta, lo que dificulta la toma de decisiones. Teniendo en cuenta este problema, hemos tenido que implementar un timeout en el cierre de conexión para no tener este problema. Por ejemplo en UPLOAD, si el cliente envía el último ACK-END, cierra su conexión y este paquete se pierde, entonces el servidor nunca va a poder cerrar su conexión.
- Atender múltiples clientes de forma concurrente.

7. Conclusión

Al implementar dos protocolos diferentes, uno Stop & Wait y uno Go-Back-N, pudimos notar las ventajas y desventajas de cada uno:

Stop & Wait es un algoritmo relativamente sencillo de implementar, con poco uso adicional de memoria al no usar buffering de paquetes, y una lógica más simple de timers y ACKs. Éstas características lo vuelven un protocolo sorprendentemente efectivo para archivos pequeños, o cuando hay poco delay (RTTs bajos).

Go-Back-N es considerablemente más complejo de implementar, ya que al ser un protocolo que utiliza pipelining, se debe cambiar la forma en que se interpretan los ACK, se debe actualizar una ventana deslizante, y otra forma de usar el timer. El hecho de poder enviar varios paquetes sin tener que esperar un ACK a la vez lo vuelve más eficiente que Stop & Wait para archivos grandes, o cuando el RTT es más considerable.

Una reflexión que cabe destacar es que ambos protocolos, si bien toman sus respectivas medidas para lograr una transferencia de datos confiable, fueron implementados con UDP como protocolo de transporte subyacente, que no asegura esa confiabilidad. Por lo tanto, es enteramente posible utilizar UDP y sus ventajas de simpleza por sobre TCP, añadiéndole un protocolo confiable como hicimos en este Trabajo Práctico, para utilizarlo como protocolo de transporte para otras aplicaciones.

Con respecto al rendimiento, en los gráficos de las figuras 29 a 32 se puede visualizar una comparación entre ambos protocolos. Podemos ver que para archivos pequeños el rendimiento es muy similar, sin embargo, a medida que aumenta el tamaño de los archivos, Go-Back-N se destaca por sus bajos tiempos de transferencia con respecto a Stop & Wait. Cuando introducimos pérdida de paquetes, esta diferencia es aún más evidente.

8. Referencias

RFC768 Postel, J., “User Datagram Protocol,” RFC 768, DOI 10.17487/RFC768, August 1980. Disponible en línea: `https://www.rfc-editor.org/info/rfc768`.

- Python 3.9 Standard Library Documentation

RFC1122 Braden, R., Requirements for Internet Hosts - Communication Layers,RFC 1122, DOI 10.17487/RFC1122, October 1989. Disponible en línea: `https://www.rfc-editor.org/info/rfc1122`.

RFC793 Postel, J., ”Transmission Control Protocol,RFC 793, DOI 10.17487/RFC793, September 1981. Disponible en línea: `https://www.rfc-editor.org/info/rfc793`.

RFC8085 Eggert, L., Fairhurst, G., ÜDP Usage Guidelines,RFC 8085, DOI 10.17487/RFC8085, March 2017. Disponible en línea: `https://www.rfc-editor.org/info/rfc8085`.