TP N°1: File Transfer

Gabriel Bedoya (107602), Juan Manuel Díaz (108183), Julián González Calderón (107938), María Sol Moon (107976) y Francisco Javier Pereira (102609)

25 de Abril de 2023

Índice

1. Introducción

El objetivo del presente trabajo es la implementación de un protocolo RDT (Reliable Data Transfer), mediante el desarrollo de una aplicación con arquitectura cliente-servidor para la transferencia de archivos, aplicando los principios básicos de transferencia de datos confiables.

Con este propósito, es necesario comprender cómo los procesos se comunican a través de la red y cómo la capa de transporte ofrece un modelo de servicios a la capa de aplicación. Además, se utilizará la interfaz de sockets y la concurrencia para permitir la conexión entre varios clientes al servidor.

2. Hipótesis y suposiciones realizadas

- Se asume que no llegarán paquetes corruptos.
- El tamaño máximo de archivo que se puede transferir es de 130 MB.
- En caso de subir un archivo cuyo nombre coincide con uno existente en el servidor, se reemplazará por el nuevo.
- La retransmisión de paquetes se realiza cuando se alcanza un timeout.

3. Implementación

La aplicación desarrollada sigue una arquitectura Cliente-servidor, donde un cliente puede enviarle un archivo al servidor (*upload*), o pedirle un archivo al servidor (*download*).

Como toda la transferencia se realiza con sockets de UDP, fue necesario agregarle un servicio de confiabilidad a la aplicación. Para lograrlo, se desarrollaron distintas estructuras y un protocolo de capa de transporte que implementa tanto el protocolo *Stop and Wait* como el *Selective Repeat*. Cuando se inicializa el servidor se puede elegir qué protocolo utilizar.

En primer lugar, cuando un cliente quiere hacer una transferencia de archivos, ya sea una subida o descarga, debe empezar una conexión con el servidor, por lo que le envía un paquete con el nombre del archivo. En caso de que sea una subida el paquete será del tipo `TFTPWriteRequestPaquet`, y en el caso de que sea una descarga el paquete será del tipo `TFTPReadRequestPaquet`. Esto sigue el protocolo TFTP, descripto en la RFC 1350.

Por cada request que el servidor recibe, crea un hilo para poder manejarla, haciendo que la aplicación sea multithreading. Cuando el servidor recibe la request, responde con un `TFTPAckPacket` para indicar que recibió el pedido; cuando el cliente recibe el reconocimiento puede subir el archivo, si esa era la operación, o esperar a que el servidor le envíe el archivo que quería descargar. En ambos casos, como no se conoce el tamaño del archivo, se utilizan estructuras para separarlo o unirlo, dependiendo de la operación. Si se está enviando un

archivo, se lo separa en paquetes con el `Segmenter`, y si se está recibiendo un archivo se lo vuelve a unir con el `Desegmenter`.

Todos los paquetes enviados entre cliente y servidor, antes de pasar por el socket de UDP, pasan por la capa de transporte implementada, que funciona como un wrapper que permite hacer un envío confiable.

Esta abstracción es un `ReliableTransportProtocol`, que se encarga de verificar que no se pierdan paquetes y que a la capa de aplicación le llegue toda la información en orden. Para que sea más fácil utilizar el protocolo, se implementaron las interfaces `ReliableTransportClient` y `ReliableTransportServer` que permiten al proceso del cliente y del servidor interactuar con el socket de forma simple.

A su vez, el protocolo implementado tiene dos tipos de paquetes: `TransportAckPackage` y `DataAckPackage`. En sus headers, ambos contienen un código que permite identificar qué tipo de paquete es, un número de secuencia y, en el caso del paquete de datos, también el tamaño de los datos que contiene el paquete. El `TransportAckPackage` se envía cuando se recibe un paquete de datos, para notificar que se recibió bien, y se incluye el número de secuencia del paquete recibido para indicar qué paquete se recibió. El `TransportDataPackage`, por otro lado, se utiliza para enviar la información en sí.

Por lo tanto, se puede observar que cuando un cliente envía una solicitud al servidor recibirá dos reconocimientos: uno proveniente de la capa de transporte y otro de la capa de aplicación.

Cuando se envía un paquete de datos, se activa un temporizador para saber cuándo reenviarlo si no se recibe el reconocimiento correspondiente. De esta forma, el protocolo evita la pérdida de paquetes. A su vez, se tiene una ventana con un tamaño fijo que permite llevar la cuenta de los paquetes que aún no fueron reconocidos, utilizando la estructura de Python `Semaphore`. Permite reutilizarla y decidir qué protocolo se va a utilizar dependiendo del tamaño de la ventana. En el caso de *Stop and Wait*, la ventana tiene tamaño 1 ya que el protocolo espera a que un paquete sea reconocido para enviar el siguiente, y en el caso de *Selective Repeat* la ventana tiene un tamaño mayor para poder tener múltiples paquetes esperando ser reconocidos.

Asimismo, por cada conexión establecida se crea un `ReliableStream`, que es donde se manejan los timers y la ventana de cada conexión. El `ReliableTransportProtocol` contiene un diccionario con todos los streams activos y a qué dirección están asociados, que utiliza para enviar y recibir los datos.

De esta forma, se realizó una aplicación de transferencia de archivos confiables por sobre el protocolo UDP.

4. Pruebas

Para la siguiente sección, se utilizará el modo verbose para detallar las operaciones realizadas.

4.1. Inicio del Servidor

```
gaby@gaby:~/Documentos/distro/TPIntroDistribuidos/src$ python3 start_server.py -p 7000 -v
>> Waiting for requests at: ('127.0.0.1', 7000)
```

En el establecimiento de conexión, el servidor queda permanentemente esperando el establecimiento de nuevas conexiones.

4.2. Solicitud de subida

```
gaby@gaby:~/Documentos/distro/TPIntroDistribuidos/src$ python3 upload.py -s test.txt -n test.txt
 -H 127.0.0.1 -p 7000 -v
>> Sending upload request to server
>> Received AckFPacket from server
>> Uploading file: test.txt
>> Finished uploading
```

Desde el lado del cliente, se observa la petición del establecimiento de conexión enviada para subir el archivo, se recibe el ACK del lado del servidor e inicia la subida del archivo.

```
gaby@gaby:~/Documentos/distro/TPIntroDistribuidos/src$ python3 start_server.py -p 7000 -v
>> Waiting for requests at: ('127.0.0.1', 7000)
>> Recieved upload request from: ('127.0.0.1', 54282)
>> Waiting for requests at: ('127.0.0.1', 7000)
>> Recieving file storage/test.txt from ('127.0.0.1', 54282)
>> File saved at: storage/test.txt
```

Por el lado del servidor, se establece la conexión y se recibe la petición del cliente de subida, el servidor continúa escuchando en otro thread en caso de que nuevos clientes envíen alguna solicitud y continúa el proceso de subida.

4.3. Solicitud de Descarga

Para la solicitud de descarga (del archivo previamente subido) se ejecutará:

```
gaby@gaby:~/Documentos/distro/TPIntroDistribuidos/src$ python3 download.py -H 127.0.0.1 -p
7000 -d test.txt -n test.txt
>> Downloading file: test.txt
>> Finished downloading
```

Desde el lado del servidor, envía el archivo guardado previamente en storage durante la subida del mismo, en la descarga al igual que la subida, el servidor continúa esperando por solicitudes nuevas.

```
gaby@gaby:~/Documentos/distro/TPIntroDistribuidos/src$ python3 start_server.py -p 7000 -v
>> Waiting for requests at: ('127.0.0.1', 7000)
>> Recieved upload request from: ('127.0.0.1', 36862)
>> Waiting for requests at: ('127.0.0.1', 7000)
>> Recieving file storage/test.txt from ('127.0.0.1', 36862)
>> File saved at: storage/test.txt

>> Recieved download request from: ('127.0.0.1', 39666)
>> Waiting for requests at: ('127.0.0.1', 7000)
>> Sending file storage/test.txt to ('127.0.0.1', 39666)
>> File sent to ('127.0.0.1', 39666)
```

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

En una arquitectura cliente servidor se tienen por lo menos dos procesos, en general corriendo en hosts distintos, donde uno actúa de servidor y el otro de cliente. Un servidor es un proceso que está siempre activo, ya que espera y escucha por conexiones entrantes de clientes. Cuando los clientes se conectan, el servidor debe manejar sus requests.

Por ejemplo, HTTP tiene una arquitectura Cliente-Servidor, donde una de las requests que el servidor debe manejar es la siguiente: cada vez que alguien abre una página web, ese proceso cliente le pide al servidor que la çargue", lo que implica buscar toda la información de esa página (texto, imágenes, diseño) y enviársela al cliente. En el host cliente, la aplicación se encargará de mostrarle la página al usuario como espera verla.

El servidor tiene una dirección IP asignada y conocida por el resto, ya que los clientes deben saber a dónde conectarse para enviar requests. Esto implica también que los clientes no se comunicarán entre sí de forma directa, si no siempre con el servidor y, si fuera necesario, el servidor podría actuar de intermediario entre los clientes.

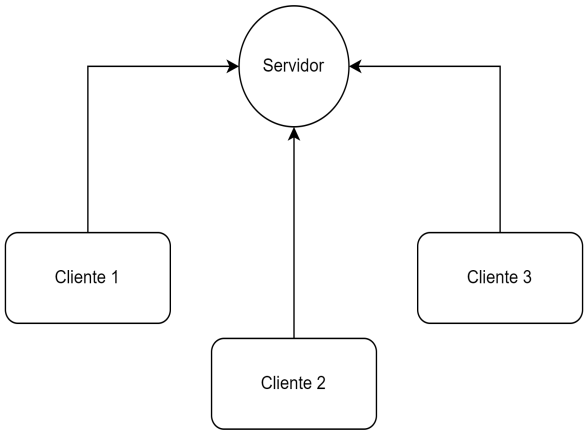

Figura 1: Múltiples clientes conectados a un único servidor.

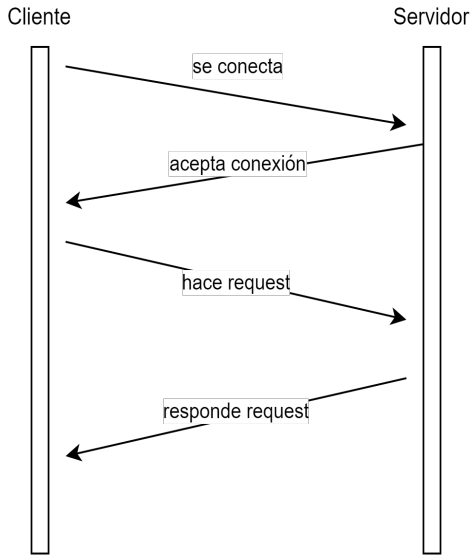

Figura 2: Descripción secuencial de una conexión entre un cliente y un servidor.

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

La función principal de un protocolo de la capa de aplicación es establecer la forma en que los programas de aplicación interactúan y transmiten datos a través de una red, para esto determina reglas y convenciones para la comunicación entre

procesos de aplicaciones que se ejecutan en diferentes sitemas terminales.

Dentro de esas reglas, se debe definir qué tipo de mensajes se van a enviar, la sintaxis que deberían tener, la semántica de los campos y qué información contienen, y reglas para determinar cuándo y cómo es el envío y la respuesta de los mensajes.

Los mensajes se envían y reciben a través de una interfaz de software llamada socket, el cual es una interfaz entre la capa de aplicación y la de transporte.

Hay ciertos protocolos de la capa de aplicación que se especifican en RFCs que son de dominio público, como por ejemplo HTTP y SMTP, pero existen muchos otros protocolos que son privados.

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

El protocolo de aplicación de este trabajo tiene dos funcionalidades: carga y descarga de archivos. Está basado en el protocolo TFTP, descripto en la RFC 1350, por lo que comparte ciertas similitudes.

Al estar por sobre un protocolo de transporte propio que permite utilizar sockets UDP de manera confiable, la aplicación no debe realizar tantos chequeos, como por ejemplo a la hora de ordenar paquetes, ya que llegan ordenados. Sin embargo, sí envía `TFTPAckPackets` para reconocer los datos que le llegan.

Siguiendo ese protocolo, se tienen distintos tipos de paquetes: `TFTPWriteRequestPacket`, `TFTPReadRequestPacket`, `TFTPDataPacket`, `TFTPAckPacket`, y `TFTPErrorPacket`. Los primeros dos se utilizan para hacerle los pedidos al servidor de subida y descarga de archivos, respectivamente, y una vez que el servidor envió el reconocimiento, el archivo se encapsula en un paquete de datos. Luego, se tienen los paquetes de errores para prevenir eventualidades.

Todos los paquetes tienen un header que contiene el código de operación, que identifica qué tipo de paquete es, y luego cada uno contiene información distinta. Por ejemplo, los paquetes de requests contienen el nombre del archivo que se quiere transferir, el paquete de datos el contenido del archivo, y el error el código de error.

El protocolo implementado por sobre UDP que implementa el transporte confiable entre hosts, utiliza unicamennte dos tipos de paquetes diferenretes: `TransportDataPacket` que se encarga de enviar los datos y `TransportAckPacket` que se encarga de confirmar que los datos fueron recibidos.

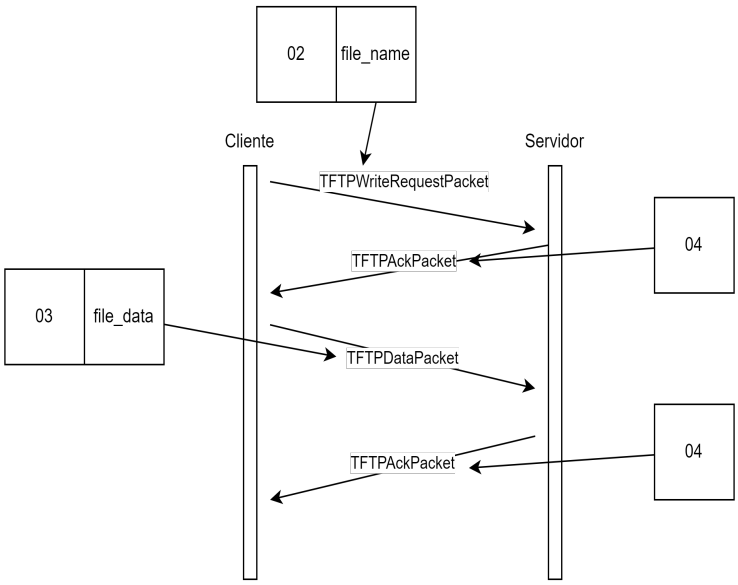

Figura 3: Subida de un archivo.

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

TCP es un protocolo orientado a la conexión, por lo que antes de enviar cualquier paquete debe establecer un handshake de tres partes (*three-way handshake*), donde se intercambia información, para asegurarse de que la conexión sea segura y estable.

Sumado a eso, TCP guarda mucha información sobre la conexión que tiene abierta, ya que eso le permite realizar un control de congestión de los enlaces, e ir variando la velocidad de transmisión de forma acorde, y mantener un control sobre los paquetes que se enviaron. Cuando TCP no recibe un reconocimiento sobre un paquete que fue enviado, lo vuelve a enviar hasta recibir la respuesta esperada. De esta forma, TCP ofrece un servicio confiable, lo que significa que se va a asegurar de que todos los paquetes enviados lleguen a destino y de forma ordenada. Cuando se requiere eviar pérdidas y que la conexión sea confiable, las aplicaciones suelen preferir utilizar TCP.

Por otro lado, UDP no es un protocolo orientado a conexión, por lo que cuando recibe datos de la capa de aplicación solo los empaqueta y pasa a la capa de red. Gracias a que no realiza validaciones extensas como TCP, funciona mucho más rápido, pero sin la certeza de que los paquetes van a llegar a destino, ya que el único chequeo es sobre el campo *checksum* del header, que verifica la

corrupción de bytes. Por lo tanto, cuando se busca priorizar la velocidad de entrega y se puede tolerar que haya pérdida de paquetes, es usual utilizar UDP, como por ejemplo lo hace DNS o las aplicaciones de streaming o de juegos online. Sin embargo, es posible utilizar UDP y tener un envío confiable si se incluyen las validaciones en el protocolo de la capa de aplicación, similar a como lo hace el protocolo QUIC.

UDP no guarda información sobre la conexión, lo que permitiría a un servidor dedicado a alguna aplicación particular mantener más conexiones de clientes activas, en comparación a TCP que, al guardar muchos datos sobre las conexiones gasta más memoria. Siguiendo esto, el header de UDP es mucho más pequeño que el de TCP (8 bytes vs 20 bytes), por lo que permite enviar más información en cada paquete.

Asimismo, como UDP tampoco realiza control de congestión sobre los enlaces, muchas aplicaciones de multimedia prefieren utilizarlo por sobre TCP, para no tener variaciones en las velocidades de transmisión.

Por lo tanto, a la hora de elegir un protocolo de capa de transporte, no existe uno mejor que otro, si no que es necesario pensar en la funcionalidad que se busca para la aplicación y qué protocolo le brindará el mejor servicio.

6. Dificultades encontradas

Al comienzo del trabajo, se empezó a plantear un protocolo de aplicación que se encargue tanto de la transferencia de archivos como de asegurarse que sea confiable. Sin embargo, al hacerlo, se generó mucho acoplamiento entre las funcionalidades, resultando en un programa confuso y con mucha interdependencia.

Por lo tanto, luego de pensarlo nuevamente, se decidió pensar al programa como un protocolo de aplicación que realizaría la transferencia de archivos, y un protocolo de transporte que se aseguraría de que los paquetes lleguen de forma confiable.

Otra dificultad fue encontrar las estructuras apropiadas que permitieran organizar el código para obtener una resolución ordenada y clara del programa. Con eso, también surgió el problema de sincronizar todos los hilos, ya que en varios casos requerían acceder a un mismo recurso y se podían generar condiciones de carrera. Poder identificar y manejar todos esos casos agregó una complejidad, propia de los programas multithreading.

7. Conclusión

En conclusión, se logró implementar un protocolo RDT aplicando los contenidos vistos en la materia, teniendo en cuenta todas las características que debería tener un protocolo confiable a partir de un protocolo no confiable como lo es UDP.

Con este propósito, fue necesario comprender el funcionamiento de la interfaz de sockets y la arquitectura cliente-servidor.

Además de esto, fueron implementados las metodologías Stop and Wait y Selective Repeat, tanto para la subida como descarga de archivos.

Cabe destacar que, para la implementación del protocolo, se tuvo en gran consideración el RFC 1350 "THE TFTP PROTOCOL".