75.43 - Introducción a los sistemas distribuidos

Capa de Transporte

Alumno	Número de Padrón	Email
Robles, Gabriel	95897	*grobles@fi.uba.ar*
Blázquez O., Sebastián A.	99673	*sebastian.blazquez96@gmail.com*
Alvarez Windey, Ariel	97893	*ajalvarez@fi.uba.ar*

Repositorio de código:
https://github.com/aalvarezwindey/fiuba-iasd-tp2-transport-layer

Introducción

En este trabajo se implementó una aplicación cliente-servidor que simula un servicio de “storaging en la nube”. El servidor esperará órdenes de recibir o entregar un archivo solicitado desde un cliente, con la posibilidad de que la comunicación sea sobre el protocolo de capa de transporte UDP o TCP. En el caso de que el protocolo de comunicación elegido sea UDP la aplicación garantiza entrega confiable: los paquetes llegarán a destino, llegarán en forma íntegra y en forma ordenada. El principal desafío entonces de este trabajo, será construir un protocolo propio en la capa de aplicación, que convierta una comunicación UDP en una comunicación con garantía de entrega y confiable.

Comparación TCP vs UDP vs QUIC

	TCP	**UDP**	**QUIC**
Capa en stack de internet	Transporte	Transporte	Aplicación (sobre UDP)
Garantía de entrega	Si	No	Si
Garantía de integridad	Si	No	Si
Orientado a conexión	Si	No	Si
Tamaño header	20 bytes	8 bytes	136 bytes (header largo) 104 bytes (header corto)
Checksum en header	Si	Si	No
Demultiplexación dada por	(ip_origen, ip_destino, puerto_origen, puerto_destino)	(ip_destino, puerto_destino)	Connection ID (en el header)
Sentido de información	Full-Duplex en forma nativa	Half-Duplex en forma nativa	Full-Duplex o Half-Duplex
RTT necesarios para establecer conexión	2	0 (sin conexión)	1 RTT la primera vez 0 RTT luego
Variables locales mantenidas en cada endhost	rcvw, cwnd, sequence number, input buffer, output buffer, timers, etc.	Ninguna	buffers, timers, parámetros de transporte
Control de flujo	Si	No	Si
Control de congestión	Si	No	Si
Utilizado típicamente en	Aplicaciones web, email, transferencia de archivos, etc.	Aplicaciones de tiempo real (vo-ip, videoconferencias, etc.)	Aplicaciones web.

Suposiciones y/o asunciones

Para realizar este trabajo se tuvieron las siguientes suposiciones:

1. El *storage-dir* con el que se inicia un servidor puede terminar o no con una '/'. Ambos casos son soportados.

Implementación del sistema con TCP

En esta sección se detalla a alto nivel la comunicación entre cliente y servidor en los casos de que la comunicación es sobre TCP.

Carga de un archivo al servidor

En la *Figura 1* podemos ver el intercambio de mensajes entre cliente y servidor para la carga de un archivo

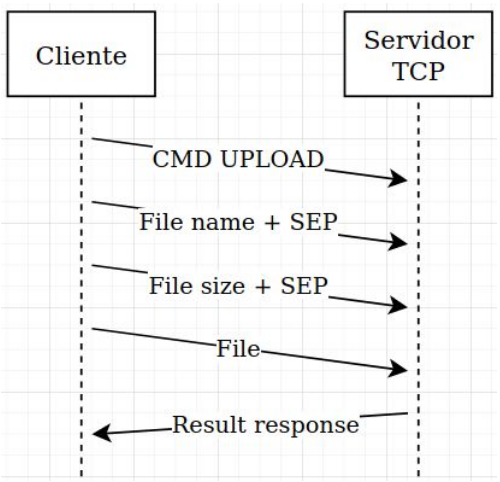

Figura 1: Carga de un archivo en TCP

Detallando los pasos del protocolo son:

1. El cliente envía el comando UPLOAD (que es el caracter '1') indicando la acción a realizar. El servidor interpreta el mismo y se dispone a esperar los próximos mensajes.
2. El cliente envía el nombre con que quiere que el servidor almacene en la nube el archivo que está cargando y un separador final (por defecto el caracter '|') para que el servidor sepa hasta cuándo dejar de leer del socket.
3. El cliente envía el tamaño en bytes del archivo a cargar, delimitado nuevamente por el separador. De este modo el servidor sabe cuántos bytes leer del socket para concluir que recibió todo el archivo.
4. El cliente transfiere (de a chunks incrementales) el archivo y el servidor a medida que va recibiendo por el socket los escribe en el archivo correspondiente previamente creado.
5. El servidor comunica el resultado de la interacción:

a. Indica que todo estuvo bien si el tamaño del archivo almacenado es igual al tamaño del archivo indicado por el cliente en el paso 3.
b. Indica que falló en otro caso

Descarga de un archivo del servidor

En la *Figura 2* vemos se detalla el análogo para la descarga de un archivo.

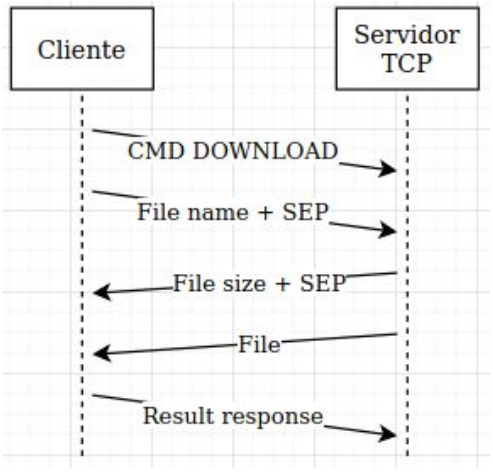

Figura 2: Descarga de un archivo en TCP

Los pasos en este caso son:

1. El cliente indica la acción a realizar en el comando DOWNLOAD (caracter '2'), el servidor lo interpreta y espera por el nombre del archivo a enviarle.
2. Similar al caso de la carga, el cliente envía el nombre del archivo y delimitador, indicando al servidor el archivo solicitado.
3. El servidor le indica el tamaño del archivo en bytes. El servidor podrá enviar '-1' en caso de que el archivo solicitado no exista, dándose por finalizada la conexión luego de esto.
4. Asumiendo que el archivo existe, el servidor envía de a chunks el archivo por el socket, mientras que el cliente recibe y va guardando en el archivo indicado en la invocación del proceso cliente.
5. El cliente comunica el resultado de la interacción:
    a. Indica que todo estuvo bien si el tamaño del archivo descargado es igual al tamaño del archivo indicado por el servidor en el paso 3.
    b. Indica que falló en otro caso.

Implementación del sistema con UDP

Carga de un archivo al servidor

1. El cliente envía el comando UPLOAD (una constante) indicando la acción a realizar y espera un ACK.
2. El servidor interpreta el comando y envía un ACK.

3. El cliente envía el nombre con que quiere que el servidor almacene en la nube el archivo que está cargando y espera un ACK.
4. El servidor recibe el nombre del archivo y envía un ACK
5. El cliente envía el tamaño en bytes del archivo a cargar y espera un ACK.
6. El servidor recibe el tamaño en bytes del archivo a cargar y envía un ACK.
7. El cliente transfiere (de a chunks incrementales) el archivo y el servidor a medida que va recibiendo por el socket los escribe en el archivo correspondiente previamente creado. Cada vez que el cliente envía un chunk espera un ACK y cada vez que el servidor recibe un chunk envía un ACK. Todo esto ocurre hasta que se envía el archivo completo.

Descarga de un archivo del servidor

1. El cliente envía la acción a realizar en el comando DOWNLOAD (una constante) y espera un ACK.
2. El servidor interpreta el comando y envía un ACK.
3. El cliente envía el nombre del archivo y espera un ACK.
4. El servidor recibe el nombre del archivo a transmitir y envía un ACK.
5. El servidor envía tamaño del archivo en bytes y espera un ACK.
6. El cliente recibe el tamaño del archivo y envía un ACK.
7. El servidor envía de a chunks el archivo por el socket, mientras que el cliente lo recibe y va guardando en el archivo indicado en la invocación del proceso cliente. Análogamente al caso anterior, cada vez que el servidor envía un chunk espera un ACK y cada vez que el cliente recibe un chunk envía un ACK. Todo esto ocurre hasta que se envía el archivo completo.

En primer lugar es necesario conocer la estructura de los datos que se envían a nivel de aplicación, lo cual nos permitirá explicar varias razones de la implementación del protocolo. Para ello se modelaron *paquetes* que contienen un número de secuencia, un *checksum* y el propio contenido del paquete (*payload*), y los ACKs que contienen un número de secuencia y un *checksum*. La longitud de los números de secuencia y *checksum* es fija, aunque fácilmente configurable.
Desde un punto de vista de alto nivel, se implementó un algoritmo de tipo *stop & wait* en el cual cuando una de las partes envía contenido, luego espera un ACK y cada vez que una de las partes recibe contenido (ya sea el contenido esperado o no) envía un ACK. De esta forma se simplifica en gran medida la implementación, pero conlleva el costo de que no es muy performante, dado que siempre hay un único paquete o ACK viajando por la red.
Con respecto a la integridad de los paquetes, se decidió implementar un *checksum* a nivel de aplicación para mostrar los principios que rigen este método. Para ello al enviar los paquetes se aplicó una función de hash sobre el número de secuencia y el *payload*, y luego en la recepción se volvió a aplicar comparando con el valor de *checksum* recibido. Análogamente se implementó lo mismo para los ACKs, pero teniendo en cuenta únicamente el número de secuencia. De esta manera implementamos un método similar al de verificación de integridad de TCP, mostrando su funcionamiento.

Otro punto a tener en cuenta es la garantía de orden, la cual se asegura por la propia simpleza del algoritmo utilizado. Dado que siempre habrá una único paquete viajando por la red nunca llegará fuera de orden.
Por otro lado lo más interesante de la implementación de este protocolo fue cómo manejar las pérdidas de paquetes (garantía de entrega). Para ello se implementó un temporizador que notifica cuando pasa más de un cierto tiempo (RTO) sin recibir una confirmación sobre el envío de un paquete. De esa forma se logra retransmitir el paquete perdido y garantizar la entrega del mismo.
Finalmente, fue interesante detectar la desconexión de alguna de las partes en la comunicación. Para ello simplemente se decidió que cuando se detectaran más de N pérdidas por RTO seguidas (en nuestro caso 5, aunque este número es fácilmente configurable), se consideraría la comunicación perdida.

Dificultades encontradas

Algunas de las dificultades encontradas fueron:

1. Tener que ajustar el uso de sockets comparado a la experiencia que se tenía con el manejo de los mismos en otras materias. En dichos casos, por ejemplo en C o C++, se enviaban bytes, entonces enviar por ejemplo enteros para indicar el tamaño de un archivo no presentaba mayor problema (siempre teniendo cuidado con el endianness). Al haber trabajado únicamente con strings en python se tuvo que recurrir al uso de delimitadores entre mensaje y mensaje.
2. Al correr pruebas automatizadas cuando se iniciaban servidores entre prueba y prueba, se obtenía un error relacionado a que el puerto ya estaba en uso. Primero se creyó que el problema era por no liberar correctamente los recursos en la aplicación del servidor, pero luego se descubrió que era un problema de cómo maneja el sistema operativo los estados de los sockets, más puntualmente uno denominado *TIME_WAIT*. Se solucionó con el *flag SO_REUSEADDR* al momento de instanciar el socket aceptador.
3. Encontramos dificultades en problemas de concurrencia con las pruebas automatizadas ya que las mismas en extraños casos podían llegar a intentar comparar los archivos en cuestión antes de que el SO haya flusheado los datos en el archivo que el servidor iba escribiendo en la operación de *UPLOAD*. Para esto se agregó un paso extra en el protocolo para TCP que figura como “*Result response*”, con este paso nos aseguramos que una vez finalizada la lectura del socket, el archivo escrito se cierre (y por lo tanto el S.O. haga el flush correspondiente) y luego validar que el tamaño del archivo generado era el esperado.

Conclusión

En este trabajo se implementó un sistema de almacenamiento y descarga de archivos cliente-servidor con tres comandos: ejecutar servidor, subir archivo y descargar archivo. Esto se implementó tanto con protocolo de capa de transporte TCP como UDP.

Gracias a que TCP es un protocolo con entrega confiable, delegamos en él la responsabilidad de asegurar que los segmentos lleguen, lleguen íntegros y lleguen en orden. Gracias a esto, desarrollar la solución mediante este protocolo no fue tan desafiante como con UDP, y simplemente se tuvo que tener especial cuidado con, a la hora de transmitir o cargar archivos, no cargarlos completamente en memoria antes de transmitirlos o escribirlos, sino de hacerlos de a partes (chunks) y asegurar que la parte que recibe el archivo termina de recibirlo antes de que el emisor cierre su conexión. Se podría decir entonces que el desafío de la implementación con TCP, más que en la transferencia de los archivos en sí, estuvo en el uso de la librería de sockets de Python (nueva para algunos integrantes del grupo) y en la algoritmia del problema.

En cuanto a la solución con UDP, se decidió optar por el algoritmo stop-and-wait para brindar entrega confiable en capa de aplicación. Este algoritmo consiste en tener como máximo un segmento en vuelo sin confirmar. Así, el origen no va a enviar una nueva pieza de datos hasta que no haya recibido el acknowledgment positivo de parte del destino. Si después de un cierto tiempo no recibe el acknowledgment correspondiente (debido a que el segmento original o el acknowledgment se perdieron), entonces lo retransmitirá. Además, cada segmento (modelado como un Paquete) tiene un checksum (calculado y agregado en capa de aplicación) para chequear la integridad en el destino. De esta forma, se garantiza la entrega confiable: los segmentos llegarán, lo harán íntegros y en orden.

Resta mencionar que el algoritmo utilizado con la transmisión UDP está lejos de ser un algoritmo performante. El throughput que alcanzará tiene una cota superior de 1 MSS por cada RTT. Esto normalmente podrá dar un throughput efectivo en el orden de las centenas de kbps. Si, por ejemplo el RTT es de 30 milisegundos y el tamaño del payload del paquete (L) es de 1000 bytes, si despreciamos el tiempo de transmisión, el origen es capaz de enviar 1000 bytes / 0.030 segundos = 266 kbps. Esta es una cota de throughput muy pobre comparada con el ancho de banda que ofrecen los canales modernos. Esto es un ejemplo de cómo un protocolo de capa de transporte puede limitar la capacidad provista por el hardware en que se soporta. Gracias a esta pobre performance, hemos visto otros algoritmos, principalmente basados en técnicas de pipelining, que permite tener más cantidad de segmentos en vuelo sin confirmar (como Go-Back-N y Selective Repeat) pero donde su implementación por software termina siendo más compleja.

Sin embargo, que UDP no disponga de los servicios que brinda TCP no implica que no fuese una alternativa como protocolo de capa de transporte en lo absoluto. Como ya se ha estudiado, TCP ha encontrado dificultades en la Internet más moderna. El ejemplo más común es la velocidad con que pueden desplegarse actualizaciones sobre el protocolo, ya que el mismo está atado al despliegue de las actualizaciones de los sistemas operativos.

Este es uno de los motivos que ha derivado en el surgimiento de QUIC, un protocolo de capa de aplicación (y por lo tanto más rápido para actualizarse) basado en UDP, donde justamente se busca mitigar las posibles falencias que ha presentado TCP a lo largo de la historia.