Redes

(TA048) Curso 02 Alvalrez Hamelin

Trabajo Práctico
TP N○1: File Transfer

3 de Octubre de 2024

Mateo Alvarez
108666

Martín Juan Cwikla
107923

Máximo Gismondi
110119

Juan Manuel Pascual Osorio
105916

Maximiliano Nicolas Otero Silvera
108634

Índice

1. Introducción

Este trabajo tiene como objetivo la implementación de una aplicación de transferencia de archivos bajo la arquitectura cliente-servidor, haciendo uso de los protocolos de transporte denominados UDP y TCP.

La arquitectura planteada para la aplicación sigue el modelo Cliente-Servidor, donde el cliente puede realizar operaciones de upload y download de archivos, mientras que el servidor se encarga de gestionar y almacenar dichos archivos. Para garantizar la entrega confiable de los datos al utilizar UDP, se implementarán los mecanismos de control de errores y reenvio de paquetes llamados "Stop and Wait" y "Selective Acknowledgment" (SACK), que complementan la funcionalidad básica de este protocolos.

A través de este trabajo, se implementará un protocolo de capa de aplicación capaz de gestionar operaciones de carga y descarga de archivos. Esto nos permitirá profundizar en los principios del "Reliable Data Transfer" (RDT), aplicando estos conocimientos en la construcción de una solución que garantice la entrega confiable de datos.

A su vez, se buscará estudiar, probar y analizar las diferencias en la performance de Stop and Wait y Selective Acknowledgment en condiciones de red variables. Las condiciones que evaluaremos serán escenarios con distintos valores de pérdida de paquetes y de latencia, simulando estas condiciones con un programa llamado "Mininet".

2. Hipótesis

En el contexto de la capa de Transporte, el protocolo utilizado junto con su implementación resultan vitales para determinar la velocidad, seguridad y la fiabilidad del envío de paquetes a través de una red. Estos factores no solo afectan el rendimiento del sistema, sino que también determinan su capacidad para gestionar errores, congestión y pérdidas de datos. Antes de proceder con la implementación del sistema de descargas propuesto, es posible formular una serie de hipótesis y predicciones sobre cómo los distintos protocolos y configuraciones impactarán en el comportamiento de la red.

- En primer lugar, se asume que al trabajar en un entorno estándar de localhost y realizar descargas, no se debería presenciar una perdida de paquetes. Esta hipótesis surge del entendimiento de que trabajando en una misma computadora, al no pasar nunca a la capa de redes, las posibilidades de que se pierda un paquete enviándolo de un proceso a otro a través de un socket son cercanas a cero. Distinto será cuando se altere el sistema con la aplicación de Mininet que permite simular la pérdida de paquetes.
- Por otro lado, comparando los dos mecanismos que se implementarán de retransmisión de paquetes, se infiere que habrán diferencias notables en la velocidad de descarga de archivos de cada uno. Se espera que en un contexto de pérdida de paquetes recurrente, un servidor que reenvíe varios paquetes selectivamente en base a las señales ACK recibidas, es decir, con un mecanismo de tipo SACK, tendrá una velocidad de descarga considerablemente mayor a un servidor que implemente Stop-and-Wait. Esto se supone ya que el primero aprovecha mejor los recursos disponibles al minimizar el tiempo de espera entre transmisiones, mientras que el segundo, al retransmitir paquetes uno a uno y esperar confirmación tras cada envío, introduce retrasos significativos, lo que reduce la eficiencia total de la transferencia de datos.
- Por último, se teoriza que las velocidades de las operaciones de Download y Upload deberían ser muy similares entre sí, ya que se trata del mismo tipo de transferencia de datos entre endpoints. Lo único que las distingue es que se invierten el origen y el destino de los paquetes. Sin embargo, puede que la implementación del servidor y especialmente la concurrencia entre varios clientes para un mismo servidor, provoque una diferenciación registrable en términos de cantidad de bits enviados de un lado al otro por segundo. De todas formas se supone que en un sistema con poca cantidad de clientes, ambos valores serán muy parecidos, sino idénticos.

Estos aspectos serán analizados intensivamente a la hora de probar el sistema implementado. A su vez se extraerán las conclusiones correspondientes de lo observado una vez finalizadas las pruebas.

3. Implementación

La aplicación desarrollada implementa los protocolos UDP y TCP embebidos en una arquitectura estándar de tipo cliente-servidor.

3.1. Servidor

El servidor es un sistema que se ocupa de proveer servicios de carga y descarga de archivos en base a los pedidos de distintos clientes.

En la implementación creada, se tiene una clase "Server" que se ocupa de escuchar por conexiones entrantes y a la vez gestionar los múltiples clientes, una vez conectados, que deseen realizar las acciones de upload y download. Esto se logra mediante el uso de una "thread-pool" en la cual se tienen distintos hilos de ejecución dedicados a manejar cada pedido.

En primer lugar, el servidor se inicializa en base a un archivo de configuración pasado por parámetro. Este, determina la dirección de Host, el puerto y a su vez, el algoritmo de control de flujo y retransmisión de paquetes con la que se desea levantar el servidor. Esta última configuración tiene dos opciones: Stop-and-Wait y SACK.

Una vez iniciado el servidor, se llama a su método de escucha denominado "listener()" el cual se dedica a contínuamente escuchar a través de un socket todos los paquetes entrantes a su dominio, con el fin de iniciar una conexión con un cliente. Al recibir un primer paquete, en caso de ser una conexión nueva, el programa se ocupa de crear una instancia de la clase "ClientHandler" que será la responsable de llevar a cabo las acciones que permitan a este cliente en particular interactuar con el servidor. Con el objetivo de atender a múltiples clientes, se delega la ejecución del handler del cliente al thread-pool, en la cual varios hilos "workers" procesan las acciones de los clientes. De esta manera, en estos hilos secundarios se procesan los paquetes recibidos por el hilo principal.

Luego, en el hilo principal se continúa escuchando a través del socket y cada paquete es retransmitido al correspondiente "worker" asignado al cliente proveniente de la dirección que envió este paquete.

3.2. Control de Congestión

Por otro lado, se implementa la clase State, que gestiona el crecimiento de las ventanas de congestión (cwnd) en función de los de acuso de recibo (que se denominan "ACKs") recibidos para la opción de SACK. Utiliza un enfoque similar al de Reno, combinando los algoritmos de Slow Start y Congestion Avoidance, añadiendo el concepto de Fast Recovery.

Cuando se recibe un ACK, la clase evalúa si es un ACK duplicado. Si se detectan tres duplicados, se activa FastRecovery para retransmitir el paquete perdido sin reducir drásticamente la ventana de congestión, lo que minimiza la degradación del rendimiento. Si se recibe un ACK nuevo, la ventana "cwnd" se incrementa dependiendo del estado de congestión en el que se encuentra sistema. En Slow Start, se incrementa por un MSS (Maximum Segment Size) el tamaño de la ventana por cada ACK recibido. Por otro lado, en Congestion Avoidance, tan solo se incrementará por una fracción del MSS (aproximadamente un MSS por cada ventana completa con acuso de recibo)

El método timeoutEvent() reinicia el estado a Slow Start al detectar un tiempo de espera, permitiendo recuperar la conexión perdida.

Este sistema de control de congestión mejora la capacidad de respuesta del servidor ante variaciones de la red y optimiza el uso del ancho de banda, permitiendo a los clientes cargar y descargar archivos de manera más fluida. Además, se realizan pruebas en condiciones de red simuladas para ajustar parámetros como el tamaño de la ventana y el umbral de transición, asegurando un servicio robusto y confiable.

3.3. Client Handler

Como se mencionó anteriormente, la clase "ClientHandler" es responsable de manejar la interacción individual entre el servidor y un cliente específico durante la ejecución de una transferencia de archivos. Esta clase se ocupa de recibir y enviar datos asegurándose de gestionar adecuadamente los paquetes y sus correspondientes paquetes ACKs. De esta forma se garantiza la confiabilidad de la transmisión de archivos, garantizando que ningún fragmento se haya perdido.

En primer lugar, cuando se crea una instancia de esta clase, se le asigna una dirección (que identifica al cliente), un socket a través del cual se comunicarán, y la ruta de la carpeta donde se almacenarán los archivos transferidos. La clase también inicializa una cola de datos donde se encolan los paquetes recibidos del cliente.

El handler sigue un proceso estructurado para manejar la comunicación con el cliente. Primero, recibe el paquete inicial SYN, que indica el inicio de la conexión. Luego, dependiendo del tipo de solicitud del cliente (carga o descarga de archivos), la clase maneja la operación correspondiente, enviando o recibiendo los datos de los archivos en fragmentos pequeños que se ajustan al tamaño máximo permitido de cada paquete.

El ClientHandler gestiona la retransmisión de paquetes en caso de pérdida de datos, y cada paquete que se envía o recibe es cuidadosamente validado para asegurar que sea nuevo y no duplicado. Además, para garantizar que el cliente reciba los datos correctamente, el servidor envía un acuse de recibo (ACK) por cada fragmento recibido o retransmitido.

Para los procesos de carga y descarga, el handler divide los archivos en fragmentos del tamaño máximo permitido y los envía al cliente.

3.4. Stop-and-Wait

En el caso de inicializar el servidor con la opción de retransmisión de tipo Stop-and-Wait, la implementación es relativamente sencilla. En el caso de enviar archivos, se espera al paquete de ACK por cada uno de los envíos antes de continuar con el siguiente. Por el otro lado, a la hora de recibir archivos, se guardan los fragmentos que se van recibiendo a través de los paquetes hasta completar cada uno y se confirma la recepción enviando un ACK de vuelta al cliente luego de cada recepción. Una vez terminada la operación, ya sea de envío o recepción de archivos, se procesan los paquetes de terminación FIN y ACK para cerrar la conexión de manera adecuada y segura.

3.5. Selective Acknowledgment

En el caso de utilizar el servidor con la opción de retransmisión de tipo SACK, la lógica es un poco más compleja. A diferencia de SW, SACK permite manejar de manera más flexible la pérdida de paquetes y la recepción fuera de orden.

Durante el envío de archivos, el servidor puede enviar múltiples fragmentos de datos consecutivamente sin esperar por una confirmación inmediata de cada uno. El cliente, al recibir los paquetes, envía ACKs que pueden incluir bloques SACK, indicando qué rangos de secuencias fueron recibidos correctamente. De esta manera, el servidor puede retransmitir únicamente los fragmentos que no fueron confirmados, optimizando el proceso y evitando retransmisiones innecesarias.

Durante el envío de archivos, el servidor puede enviar múltiples fragmentos de datos de manera continua sin esperar confirmación inmediata para cada uno de ellos. Cada vez que el cliente recibe estos fragmentos, envía un paquete SACK que incluye los números de secuencia de los paquetes recibidos correctamente. Este feedback permite al servidor retransmitir únicamente aquellos paquetes que no fueron confirmados. De esta manera se logra enviar más de un paquete por intervalo, a diferencia del algoritmo SW.

Al recibir archivos, el servidor almacena los paquetes recibidos, ya sea de forma ordenada o fuera de orden. Los paquetes recibidos en orden se almacenan directamente en una cola, mientras que los fuera de orden se guardan en un diccionario para ser reordenados más tarde. Esto asegura

que los fragmentos lleguen correctamente antes de reensamblar el archivo y que los paquetes fuera de secuencia sean manejados adecuadamente.

Además, se implementa un sistema para manejar los paquetes SACK, donde el servidor indica al cliente cuáles son los bloques recibidos correctamente y qué partes deben ser retransmitidas.

Al final de la operación, ya sea de envío o recepción, el servidor procesa los paquetes de terminación (FIN y ACK) de forma similar a Stop-and-Wait, produciendo que la conexión se cierre adecuadamente tras la transmisión completa del archivo.

4. Pruebas

Se realizaron múltiples pruebas en el servidor implementado utilizando el software "Mininet", con el fin de emular la pérdida de paquetes y la latencia en la red. Se evaluó lado a lado el rendimiento de los protocolos **Stop-and-Wait** y **Selective Acknowledgment**. Se registraron los tiempos de transferencia de tres archivos de distintos tamaños, bajo estas diferentes condiciones.

Se usó un tiempo máximo para recepción de paquetes (Timeout) de 1 segundo. Esto significa que si el servidor o el cliente no recibieron un paquete en 1s, vuelven a envíar su correspondiente pedido. A su vez, se fijó la desconexión automática al llegar a 10 timeouts.

A continuación se muestran los resultados obtenidos:

4.1. Imagen en formato .png de 12KB

Transferencia sin pérdida de paquetes y sin latencia

- **SW**: 0.09s, 0.09s, 0.10s
- **SACK**: 0.08s, 0.08s, 0.09s

Transferencia con 5 % de pérdida de paquetes y sin latencia

- **SW**: 4.10s, 4.10s, 4.15s
- **SACK**: 0.10s, 1.08s, 2.07s

Transferencia con 10 % de pérdida de paquetes y sin latencia

- **SW**: 14.2s, 3.4s, 12.1s
- **SACK**: 2.08s, 1.07s, 4.07s

Transferencia con 15 % de pérdida de paquetes y sin latencia

- **SW**: 12.12s, 18.11s, 12.12s
- **SACK**: 4.08s, 3.08s, 2.09s

Transferencia con 20 % de pérdida de paquetes y sin latencia

- **SW**: 22.13s, 22.17s, 28.12s
- **SACK**: 6.15s, 5.07s, 5.09s

Transferencia con 50 % de pérdida de paquetes y sin latencia

- **SW**: 22.13s (real time)
- **SACK**: No se realizó prueba

Transferencia sin pérdida de paquetes y con 20ms de latencia

- **SW**: 2.17s, 2.17s, 2.18s
- **SACK**: 0.46s, 0.47s, 0.46s

Transferencia con 10 % de pérdida de paquetes y 20ms de latencia

- **SW**: 9.19s, 7.21s, 11.16s
- **SACK**: 5.32s, 4.39s, 3.37s

Transferencia con 10 % de pérdida de paquetes y 70ms de latencia

- **SW**: 15.3s, 19.7s, 18.6s
- **SACK**: 4.06s, 2.08s, 4.79s

4.2. Imagen en formato .webp 175KB

0 % de pérdida de paquetes:

- **SW:** 0.22s
- **SACK:** 0.19s

10 % de pérdida de paquetes:

- **SW:** 10 % de pérdida de paquetes: 1:25.33s
- **SACK:** 10 % de pérdida de paquetes: 7.09s

4.3. Prueba de estrés - Archivo .pdf de 12MB

1 % de pérdida de paquetes, timeout = 30ms, latencia = 10ms

- **SW**: 1:03.22s
- **SACK**: 0:11.07s

5. Análisis

El análisis de las pruebas realizadas revela varios aspectos importantes sobre el comportamiento de los protocolos Stop-and-Wait (SW) y Selective Acknowledgment (SACK) bajo distintas condiciones de red.

En primer lugar, se confirmó la hipótesis propuesta acerca de que cuando se trata de una red local, no se presenta pérdida alguna de paquetes sin simularla con algún software externo. Por el lado de la comparativa de algoritmos, se observó que cuando no se configura la pérdida de paquetes con Mininet, ambos protocolos funcionan de manera eficiente y sin mayores diferencias en cuanto a rendimiento. Tanto SW como SACK muestran tiempos de transferencia casi idénticos en estas situaciones, lo que refleja que, sin pérdida de paquetes, ambos protocolos son igualmente eficaces. Los tiempos de transferencia de archivos pequeños, como la imagen en formato .png de 12KB, son prácticamente iguales en ambos casos, registrando tiempos cercanos a 0.09 segundos para SW y 0.08 segundos para SACK.

Sin embargo, cuando se introduce latencia al envío de paquetes, se observa un aumento lineal en los tiempos de transferencia para ambos protocolos. Esto es particularmente notable en la transferencia sin pérdida de paquetes, pero con 20ms de latencia. En este caso, aunque el incremento es visible en ambos protocolos, SACK sigue mostrando tiempos de respuesta más rápidos que SW. Por ejemplo, mientras SW presenta tiempos de alrededor de 2.17 segundos, SACK reduce ese tiempo a aproximadamente 0.46 segundos.

El impacto más significativo en el rendimiento se produce cuando se introduce una pérdida de paquetes. En estas condiciones, el protocolo SACK muestra una ventaja notable sobre SW. A medida que aumenta la tasa de pérdida de paquetes, SW experimenta tiempos de transferencia significativamente más largos, mientras que SACK gestiona de manera más eficiente la retransmisión de los paquetes perdidos, por lo que se traduce en tiempos de transferencia más cortos y consistentes. Por ejemplo, con un 10 % de pérdida de paquetes y sin latencia, SW presenta tiempos que varían ampliamente, con picos de hasta 14.2 segundos, mientras que SACK logra completar la misma transferencia en tan solo 2.08 segundos. Esta tendencia se mantiene en todas las pruebas con pérdida de paquetes.

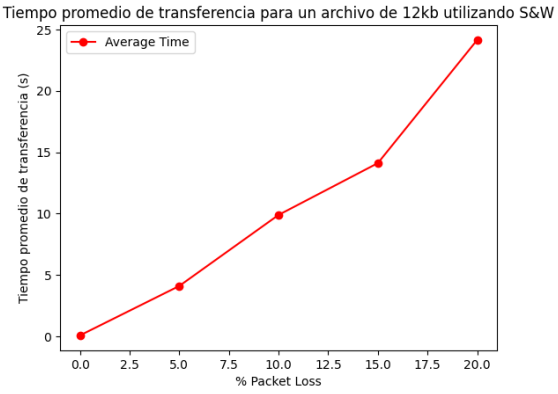

Figura 1: Tiempo de transferencia de SW de un archivo de 12KB en función de la pérdida de paquetes

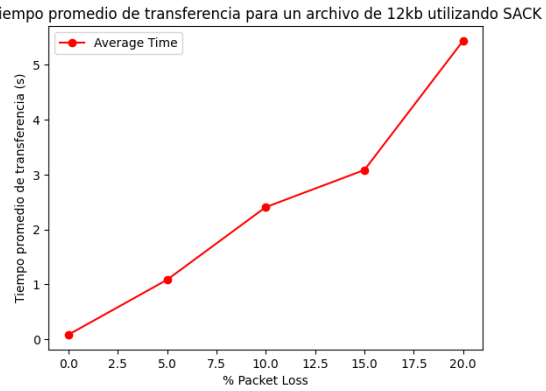

Figura 2: Tiempo de transferencia de SACK de un archivo de 12KB en función de la pérdida de paquetes

Finalmente, en la prueba de estrés que consistió en la transferencia de un archivo .pdf de 12MB, se destaca aún más la capacidad de SACK para manejar grandes volúmenes de datos con pérdidas mínimas en tiempos razonables. Este algoritmo logro completar la transferencia en tan solo 11 segundos. SW, por su parte, tuvo una demora de más de 1 minuto. Esto confirma que SACK es la opción más robusta y confiable cuando se trabaja con redes que presentan pérdidas de paquetes, permitiendo la transferencia de archivos grandes en tiempo y forma.

6. Preguntas

1. Describa la arquitectura Cliente-Servidor.

   La arquitectura Cliente-Servidor es un modelo ampliamente utilizado en sistemas distribuidos, donde las funciones y responsabilidades entre las partes involucradas en una comunicación de red, están claramente separadas. Se separa en dos tipos de entidades: El cliente y el servidor.

   En primer lugar, el servidor es un sistema que se encarga de gestionar y controlar el acceso a los recursos compartidos. Generalmente, un servidor está diseñado para manejar múltiples solicitudes simultáneamente de diferentes clientes, lo que lo convierte en el punto central de la interacción en el sistema. Lógicamente, los servidores suelen tener acceso a todo lo que es requerido por los clientes, ya sea dentro de una base de datos, archivos, etc. En términos simples, la función de un servidor es atender a las necesidades de los clientes para el correcto funcionamiento de todo el sistema.

   Por otro lado, el cliente es una entidad totalmente distinta del sistema. Generalmente se trata de una aplicación corriendo en un "end-system", que actúa como el solicitante de un servicio. En esencia, el cliente envía peticiones al servidor con el objetivo de acceder a datos, ejecutar procesos o manipular recursos que no están localmente disponibles. El cliente no necesita tener conocimiento del funcionamiento interno del servidor. Sino que solo se comunica mediante un protocolo establecido para realizar solicitudes y recibir respuestas.

   El objetivo de este modelo es la centralización de los recursos y servicios en el servidor, lo que permite una administración más eficiente, un control de acceso más seguro, y la posibilidad de realizar mantenimientos o actualizaciones sin afectar directamente a los clientes. Además, la escalabilidad es una característica destacada, ya que es posible aumentar la capacidad del servidor o agregar nuevos servidores para manejar un mayor número de clientes sin necesidad de modificar las aplicaciones cliente.

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

   La función de un protocolo de capa de aplicación es establecer las reglas y procedimientos que permiten la comunicación entre aplicaciones que se ejecutan en diferentes dispositivos dentro de una red. Este tipo de protocolo define cómo las aplicaciones intercambian datos y qué formato deben tener los mensajes para que ambas partes puedan comprender y procesar correctamente la información. Al estar en la capa más alta del modelo TCP/IP, el protocolo de capa de aplicación interactúa directamente con el software de usuario, como navegadores web, clientes de correo electrónico, o aplicaciones de transferencia de archivos, como es el caso en este trabajo.

   Estos protocolos definen cómo se deben estructurar, transmitir y procesar los datos en forma de paquetes, permitiendo el correcto funcionamiento de servicios como la web, el e-mail, la transferencia de archivos, etc. Sin embargo, el protocolo no se ocupa de llevar a cabo el transporte de estos paquetes a través de la red. A esto se dedican las capas inferiores de red.

   Existen varios protocolos de capa de aplicación, como HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol) o SMTP (Simple Mail Transfer Protocol), son específicos para cada tipo de aplicación y se encargan de gestionar aspectos cruciales como la sintaxis de los datos, el formato de las peticiones y respuestas, la autenticación, la sincronización, y el control de errores. Su función es asegurar que los datos enviados desde una aplicación sean correctamente recibidos e interpretados por otra aplicación en un sistema remoto.

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

   El protocolo de aplicación desarrollado es relativamente simple y tiene una semejanza al protocolo FTP mencionado anteriormente.

   Para la implementación del sistema, se debieron crear dos tipos de paquetes distintos: uno de SACK y uno para SW. Ver figura 1.

   Debido a que el armado y procesamiento de estos paquetes tienen la responsabilidad de, en esencia, controlar y regular el correcto envío y arribo de los paquetes, junto con sus acusados

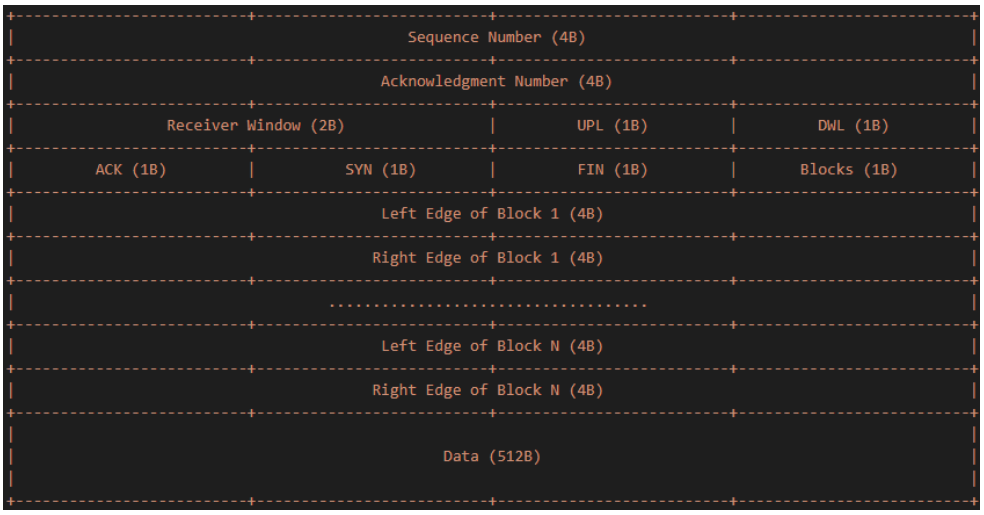

Figura 3: Mapa de bytes de Paquete SACK

de recibo, se entiende que dicha parte de la implementación vive en la capa de transporte. Sin embargo, algunos de los campos de estos paquetes sí, de hecho se terminan volcando en lo que se refiere a la capa de aplicación. En específico, los bytes DWL y UPL forman parte de ella.

Dicho esto, volviendo a la pregunta en cuestión, nuestro protocolo de aplicación hace uso de los atributos mencionados en los paquetes operando bajo el siguiente procedimiento: En primer lugar, el cliente se conecta al servidor con un mensaje que tiene un flag de download o de upload. Esto determina si el cliente va a realizar la operación de descarga o de carga. A su vez, Este mensaje es recibido por el servidor, el cual manda el correspondiente ACK. Luego, ya iniciada la comunicación, el cliente se dedica a mandar el nombre del nombre del archivo en cuestión. En el caso de estar descargando, este nombre será el correspondiente al archivo que posee el servidor. Consecuentemente, empieza la transferencia de datos, siguiendo la dinámica ya descripta en la sección de Implementación.

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

   En la capa de transporte del stack TCP/IP se ofrecen dos protocolos principales: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). Ambos protocolos son fundamentales para el envío de datos a través de redes, pero ofrecen servicios y características diferentes, lo que los hace apropiados para distintos tipos de aplicaciones.

   TCP es un protocolo orientado a un tipo de conexión que garantiza la entrega confiable y en orden de los datos. Cuando un emisor utiliza TCP, este establece una conexión con el receptor mediante un proceso de "handshake"de tres pasos, lo que asegura que ambas partes están listas para intercambiar información. Una vez establecida la conexión, TCP descompone los datos en paquetes, los numera y los envía. Además, incluye mecanismos de control de flujo y congestión, lo que asegura que el receptor no se vea sobrecargado y que la red no se sature. Si un paquete se pierde o llega con errores, TCP lo detecta y lo reenvía. Al finalizar la comunicación, TCP se encarga de cerrar la conexión de manera ordenada. Debido a estas características, TCP es ideal para aplicaciones donde la integridad de los datos es crítica, como la navegación web (HTTP/HTTPS), transferencia de archivos (FTP), o el envío de correos electrónicos (SMTP).

   Por otro lado, UDP es un protocolo sin conexión, lo que significa que no establece una comunicación previa antes de enviar los datos. Los paquetes de datos, conocidos como datagramas, se envían sin numerar ni comprobar su recepción, lo que implica que no se garantiza la entrega, el orden, ni la integridad de los datos. Esto hace que UDP sea mucho más rápido y

eficiente en términos de ancho de banda, pero menos confiable que TCP. UDP no implementa mecanismos de control de flujo ni de congestión, por lo que el emisor puede enviar datos al ritmo que desee, sin importar si el receptor o la red pueden manejarlos. Estas características hacen que UDP sea adecuado para aplicaciones en las que la velocidad es más importante que la confiabilidad, como el streaming de video o audio, los videojuegos en línea, y las videollamadas (VoIP), donde la pérdida ocasional de paquetes no afecta significativamente la experiencia del usuario.

La principal diferencia entre TCP y UDP radica en el equilibrio entre confiabilidad y velocidad. TCP prioriza la entrega exacta y ordenada de los datos, lo que lo hace más lento debido al overhead generado por sus mecanismos de control, mientras que UDP opta por una transmisión rápida y eficiente, sacrificando la confiabilidad ya que no tiene ningún mecanismo de recuperación de paquetes, entre otras cosas. La elección entre TCP y UDP depende de las necesidades de la aplicación: si se requiere la entrega precisa de los datos, como en aplicaciones de mensajería o transferencias de archivos, TCP es la mejor opción definitivamente, por su alta confiabilidad. En cambio si la prioridad es la velocidad y la latencia baja, como en aplicaciones de transmisión en tiempo real, UDP es más apropiado al ser, en términos generales, más rápido para enviar mucha información.

7. Dificultades

En esta sección se detallan algunas de las dificultades que se enfrentaron durante el desarrollo de este trabajo y el estudio de sus temáticas.

- La implementación del protocolo Selective Acknowledgment, produjo una dificultad considerablemente mayor en comparación con el desarrollo de su análogo Stop-and-Wait. Para llevar a cabo SACK fue importante leer y entender el protocolo con mucha atención mucho antes de poder comenzar a programarlo. En primer lugar, el mayor problema encontrado fue que para probar el funcionamiento correcto de este sistema, se implicó necesariamente que estuvieran terminados los lados de emisor y receptor. Por lo tanto, se tuvieron que programar ambas partes completamente antes de poder depurarlas, lo cual produjo errores simples pero dificiles de encontrar. Adentrándonos en la implementación, mantener la referencia y control de los números de secuencia en bytes y de ACKs de por sí, también demostró cierta complejidad y tomó su tiempo de refinar para llegar a un funcionamiento adecuado.
- Respecto a la investigación realizada, se tuvieron complicaciones para leer los informes de RFC's con el fin de entender los algoritmos de . Por ejemplo, fue difícil encontrar la información de qué se debe hacer en casos específicos, como por ejemplo qué sucede cuando se reciben dos paquetes de tipo ACK casi inmediatamente uno después del otro.
- Por otro lado, el software de Mininet acarreó significativos problemas de accesibilidad. De por sí, la aplicación no tiene soporte para las últimas versiones de Python, lo que generó una problemática de compatibilidad que costó resolver. Además, el programa se tuvo que ejecutaren una máquina virtual, lo cual hace, por momentos, un poco más inaccesible el uso de la herramienta.

8. Conclusión

A lo largo del desarrollo, se logró la correcta implementación de una aplicación de red con arquitectura cliente-servidor, utilizando el protocolo UDP como capa de transporte. Se implementaron exitósamente las funcionalidades de carga y descarga de archivos, cumpliendo con los requisitos de transferencia confiable de datos (Reliable Data Transfer) a través de los dos mecanismos: Stop-and-Wait y Selective Acknowledgment.

La aplicación se validó bajo distintas condiciones de red simuladas con Mininet, incluyendo pérdidas de paquetes y latencia, demostrando ser capaz de operar eficientemente en escenarios con hasta un 10 % de pérdida de paquetes, e incluso mayor. Además, se logró procesar concurrentemente múltiples transferencias de archivos, permitiendo que varios clientes se conectaran al servidor y realizaran operaciones simultáneamente sin comprometer la integridad de los datos.

En cuanto a la comparación de los protocolos, si bien ambos cumplieron con las expectativas en condiciones de red sin pérdida de paquetes, el protocolo SACK mostró un desempeño superior en redes con condiciones adversas como la pérdida de paquetes y latencia, minimizando significativamente los tiempos de transferencia en comparación con Stop-and-Wait. Esta diferencia es particularmente notable en escenarios con archivos grandes y packet loss, donde SACK demostró una eficiencia superadora, gracias a su retransmisión selectiva de paquetes.

Este trabajo cumplió con el objetivo de profundizar nuestro entendimiento de los distintos elementos que interactúan entre sí en las capas de transporte y aplicación, a la hora de transferir información a través de la red. A su vez, el análisis comparativo realizado entre los dos algoritmos nos proporcionó una perspectiva más clara sobre cómo gestionar la calidad de servicio en redes de comunicación y cómo elegir el protocolo adecuado según las condiciones de red encontradas.