TP1: File Transfer

[75.43] Redes
Primer Cuatrimestre de 2025

Integrantes

Alumno	Padrón
Villa Jimenez, Alexander	95428
Juares, Damaris	108566
Schejtman, Ezequiel	102110
Mariano Tomas Medela	101769
Zanardo, Nicolás Francisco	96155

Índice

Introducción

El trabajo práctico consiste en implementar un protocolo que provea Reliable Data Transfer por sobre UDP. En el mismo se implementa este servicio en un protocolo de aplicación y provee dos formas de entregar los paquetes: Stop & Wait y Selective Repeat. Esto debe permitir a un cliente comunicarse con el servidor para subir un archivo o para descargarlo.

Para ello se implementó datagramas, interacciones con estos, y los protocolos y su reacción ante estos. Además, se utilizaron timeouts y fue testeado con la herramienta Mininet para simular pérdidas.

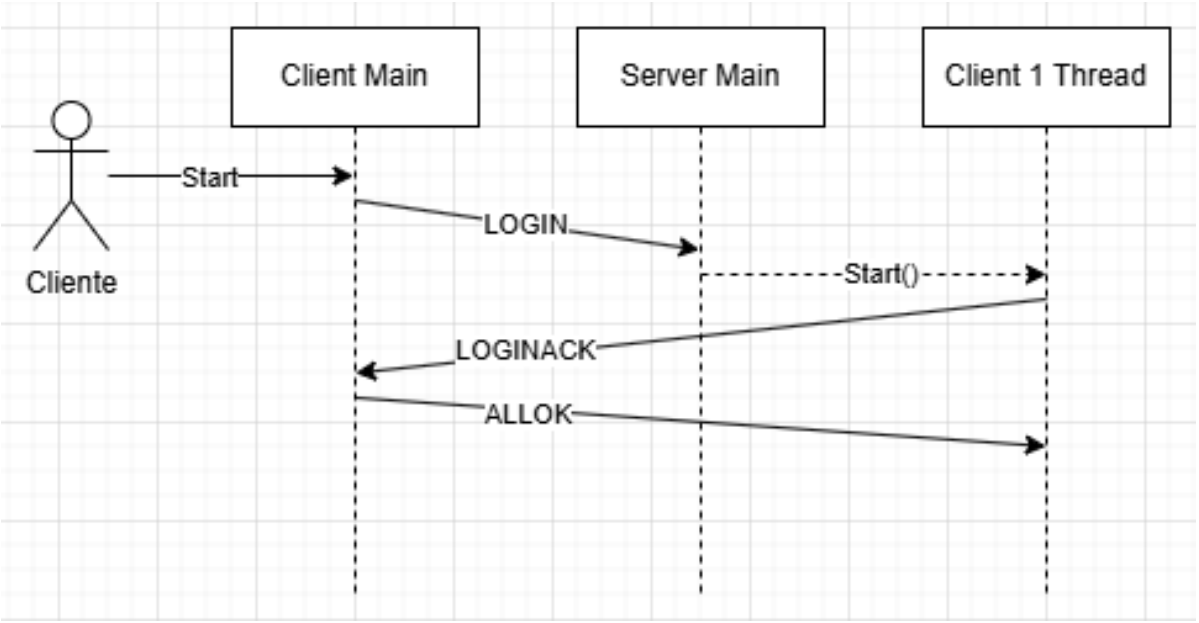
Hipótesis y suposiciones realizadas

1. **Red no confiable**: se asume que el canal de comunicación basado en UDP puede sufrir pérdidas, duplicaciones o desorden, por lo que el protocolo debe implementar mecanismos propios de confiabilidad
2. **Ambiente de ejecución**: Las pruebas y ejecuciones se realizaron en un entorno de red local y una simulada con Mininet, donde el comportamiento de pérdida pudo ser reproducido artificialmente para validar los mecanismos de retransmisión.
3. **Direcciones IP y puertos conocidos**: se presupone que el cliente y el servidor conocen de antemano las direcciones IP y puertos para establecer la conexión, sin necesidad de un protocolo de descubrimiento
4. **No hay corrupción de datos**: se parte de la hipótesis de que los datos recibidos están íntegros, es decir, que si un paquete es recibido, su contenido es válido.
5. **Un único cierre por conexión**: se asume que cada conexión entre cliente y servidor solo realiza un único proceso de cierres evitando cierres concurrentes o duplicados.
6. **Tiempos de espera y número de reintentos razonables**: el protocolo se basa en tiempos de espera y reintentos fijos, bajo la suposición de que estos son suficientes para lidiar con pérdidas ocasionales.
7. **Fiabilidad a nivel de aplicación**: se asume que la lógica del protocolo implementa la confiabilidad necesaria para garantizar una terminación ordenada, dado que UDP, por sí mismo, no ofrece garantías de entrega ni de orden.

Implementación

Handshake

Ambos protocolos utilizan un Handshake tanto para iniciar la conexión (3-way handshake) como también para terminar la conexión sin errores (4-way handshake).
En el **3-way handshake**, se envía el nombre del archivo que se va a subir o descargar del servidor, y la acción (o "modo") de interacción con el servidor (upload o download, subir o bajar un archivo).



El **4-way handshake** no requiere información adicional, sólo los FIN Y ACK estándar según el RFC 793.

Stop & Wait

Definición

El protocolo Stop & Wait es un algoritmo de control de flujo y confiabilidad, que garantiza la entrega ordenada y libre de errores de datos. Su funcionamiento se basa en una lógica simple: el emisor envía un paquete y espera una confirmación (ACK) antes de enviar el siguiente.

Una vez enviado un paquete, el emisor inicia un temporizador y permanece inactivo hasta recibir el ACK correspondiente. Si el ACK no llega antes de que expire el temporizador, se retransmite el mismo paquete. Cada paquete incluye un número de secuencia que permite al receptor identificar duplicados y confirmar correctamente el paquete recibido.

El receptor, por su parte, responde con un ACK cuando recibe un paquete válido y descarta duplicados si recibe un paquete con el mismo número de secuencia que el anterior.

Este protocolo garantiza una entrega ordenada y confiable, pero es ineficiente en redes con alta latencia o ancho de banda elevado. Por esta razón, aunque es ideal para sistemas simples

o con bajo tráfico, no se utiliza en aplicaciones que requieren alta velocidad de transmisión.

Su principal ventaja es la simplicidad de implementación, ya que requiere poca lógica de control, pero esta misma característica limita su desempeño en entornos más exigentes.

Implementación

La forma en la que se implementó Stop & Wait para el upload una vez establecida la conexión es la siguiente:

- Al iniciar, el cliente comienza leyendo los primeros bytes del archivo, y envía un paquete con número de secuencia 0 y los datos leídos de ese archivo. Si se recibe un ACK con el mismo número de secuencia, se leen secuencialmente los bytes del archivo, se incrementa el número de secuencia, y se envía el segundo paquete. Esto se repite secuencialmente hasta que se termina de leer el archivo siempre y cuando se reciban los ACK correspondientes.

- A cada envió del datagrama le corresponde un timeout, para contemplar el caso en el que el paquete se pierda. Si el timeout se alcanza (es decir, no recibimos el ACK con el número de secuencia correcto dentro de un intervalo determinado de tiempo), se envía nuevamente el mismo paquete y se repite la iteración. El timeout se corre solo del lado del cliente. El servidor no requiere correrlo también ya que si el ACK se pierde, el cliente lo reenvía.

- Cuando el cliente termina de enviar todo el archivo y recibir sus respectivos ACK, el cliente manda un último paquete al servidor que sirve de notificación de que la transferencia terminó. Una vez recibido el ACK de este paquete, el cliente inicia el **4-way handshake** para terminar la conexión de forma correcta.

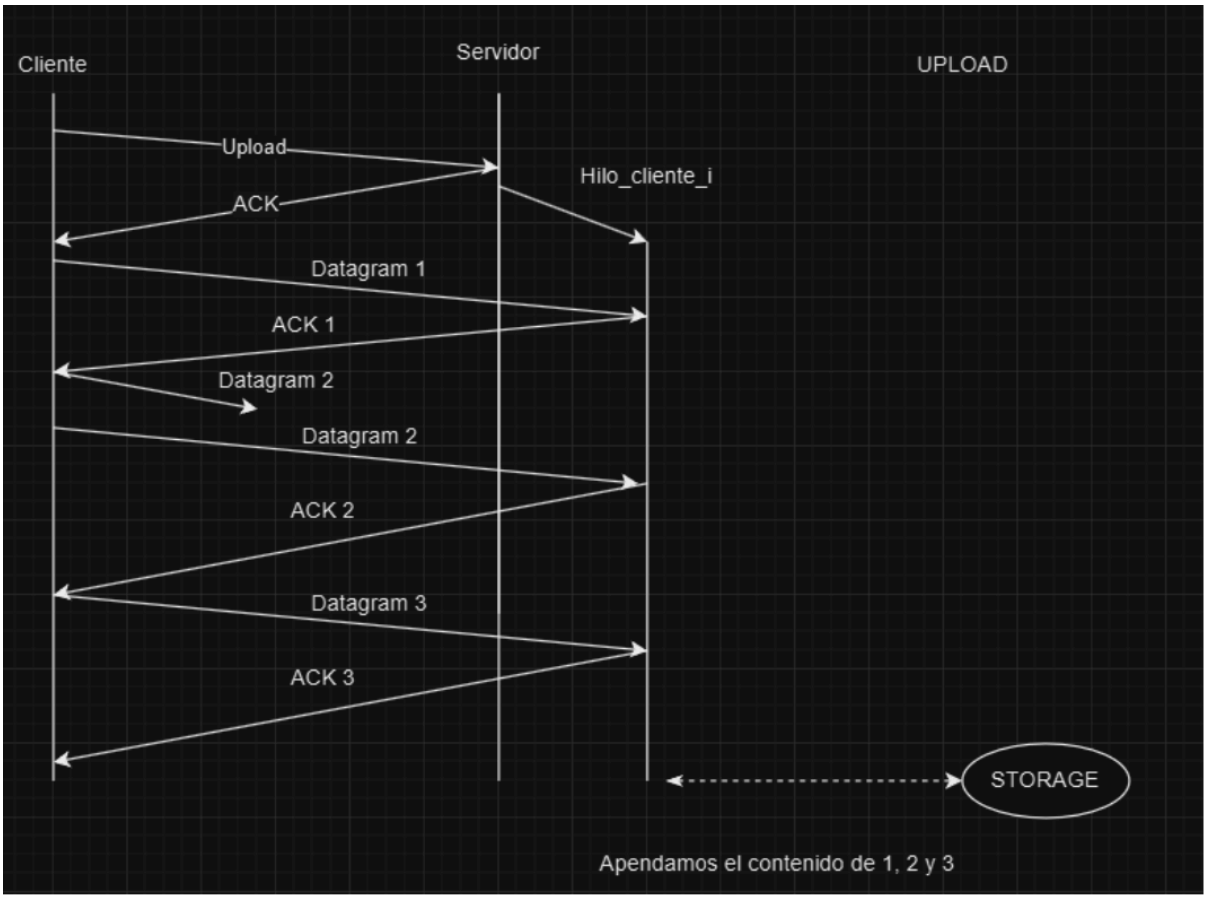


Figura 1: Diagrama de Upload en Stop & Wait

La forma en la que se implementó Stop & Wait para el download una vez establecida la conexión es la siguiente:

- Nuevamente se avanza secuencialmente rechazando y aceptando paquetes, pero invirtiendo los roles. Es ahora el servidor quien recibe ACKs del cliente y envía la información del archivo en cuestión, y es el cliente quien envía los ACKs indicando que recibió la información correctamente.
- De esta misma forma también se invierte quien corre el timeout, siendo el servidor el encargado de esto pues es quien está enviando la información del archivo al cliente.

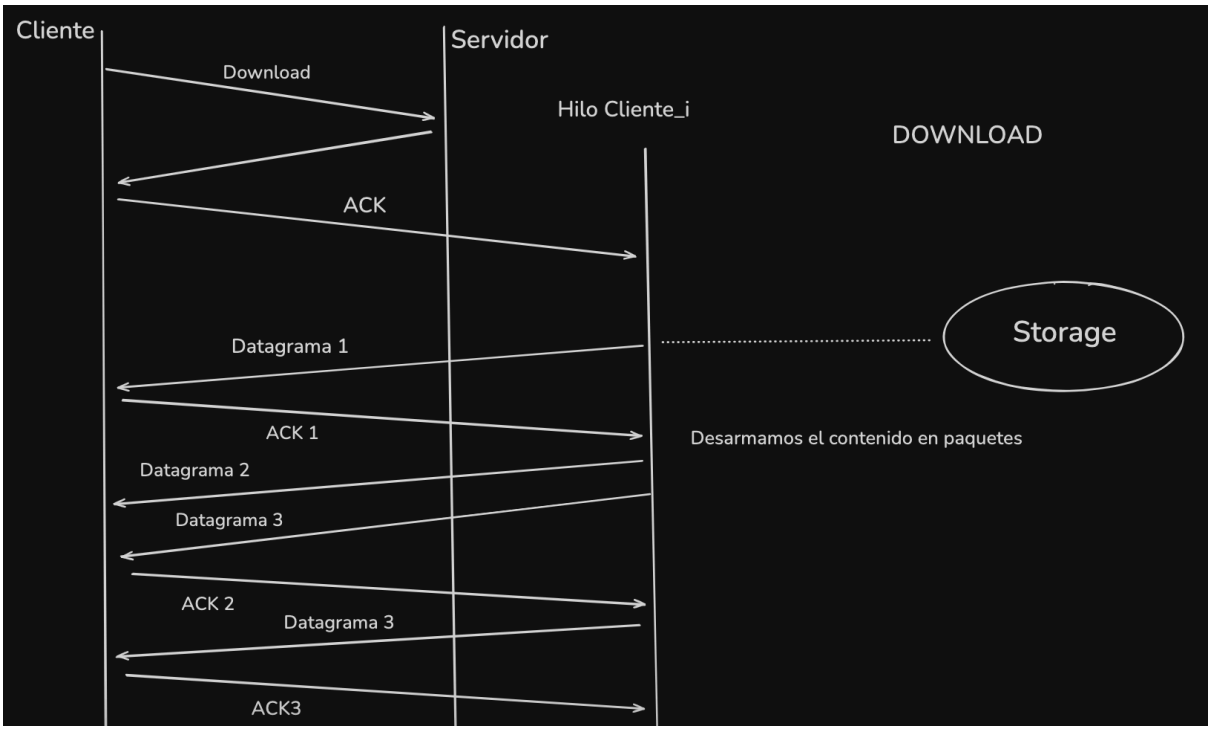


Figura 2: Diagrama de Download en Stop & Wait

Selective Repeat

Definición

El protocolo Selective Repeat es un algoritmo de control de flujo y confiabilidad utilizado en redes de computadoras que permite la transmisión simultánea de múltiples paquetes dentro de una ventana deslizante. A diferencia de otros protocolos como Stop & Wait o Go-Back-N, Selective Repeat permite que los paquetes lleguen fuera de orden y sean almacenados temporalmente por el receptor, sin requerir la retransmisión de todos los paquetes posteriores ante una pérdida.

Cada paquete enviado incluye un número de secuencia, y el receptor confirma su recepción mediante ACKs individuales. Si un paquete se pierde o se corrompe, solo ese paquete es retransmitido, lo que reduce el tráfico de retransmisión innecesaria y mejora el aprovechamiento del ancho de banda.

El emisor mantiene un buffer con los paquetes no reconocidos y gestiona temporizadores independientes para cada uno. El receptor, por su parte, también mantiene un buffer para almacenar los paquetes que llegan fuera de orden, y los entrega en secuencia a la capa superior una vez que se han recibido correctamente.

Este protocolo es más complejo que los mencionados, ya que requiere mayor capacidad de almacenamiento y lógica en ambos extremos, pero ofrece una mayor eficiencia en entornos con alta latencia o pérdidas frecuentes, debido a su capacidad para minimizar la retransmisión y mantener un flujo de datos constante.

Implementación

La implementación usando Selective Repeat para el upload es la siguiente:

- **Ventana de envío**: el cliente mantiene un buffer con los paquetes enviados pero no confirmados
- **Temporizadores individuales**: cada paquete tiene su propio temporizador para manejar retransmisiones
- **Recepción de ACKs**: el cliente procesa ACKs fuera de orden y libera el espacio en la ventana para nuevos envíos

La implementación usando Selective Repeat para el download es la siguiente:

- **Buffer de recepción**: Los paquetes que llegan fuera de orden son almacenados temporalmente
- **ACKs individuales**: Cada paquete recibido correctamente genera un ACK con su número de secuencia
- **Reconstrucción del archivo**: Una vez que los paquetes faltantes se completaron, el archivo se ensambla en el orden correcto

Pruebas

Mininet

Se utiliza mininet para simular distintas condiciones de red utilizando una topología propuesta por la cátedra.

- Topografía de la red: se propuso una *topología lineal* de 2 hosts conectados a través 3 switches (mytopo.py).

  $ sudo mn --custom mytopo.py --topo mytopo

- Se lanza cliente (upload/download) en el host h1 y el servidor en el host h2.}

  mininet> h2 python3 ./src/start-server.py -H 10.0.0.1 -p 11111 -s ./serverfiles -v &

  mininet> h1 python3 ./src/upload.py -H 10.0.0.1 -p 11111 -s /dir/file -n file_name

Condiciones de red a simular

- Fragmentación.
- Pérdida de paquetes.
- Tráfico de red UDP/ TCP.

Pasos a realizar

- Se procede a *forzar la pérdida de paquetes* del switch 3, el cual está conectado a h2.

  mininet> sh tc qdisc add dev s3-eth2 root netem loss 10%

- Se *reducirá el valor de MTU (Maximum Transmission Unit)* de la interfaz s2-eth2.

  Primero examinamos valores iniciales de MTU:

  mininet> sh ifconfig

  Ahora se procede a modificar el valor utilizando la herramienta **ifconfig**

  mininet> sh ifconfig s2-eth2 mtu 1000

- Procedemos a generar tráfico con la herramienta **iperf** teniendo en cuenta de que queremos forzar la fragmentación de paquetes.

  Tráfico UDP (-u):

  Envía datagramas del tamaño exacto que especifiques (-l)

  mininet> h2 iperf -s -u & #Servidor UDP

  mininet> h1 iperf -c h2 -u -b 10M -l 3000 -t 5

  Trafico TCP:

Divide automáticamente los datos en segmentos (MSS- *Maximum Segment Size*)

TCP no permite definir el tamaño del segmento manualmente
(como sí hace UDP con -l).

En su lugar, negocia el MSS (Maximum Segment Size) durante el handshake (normalmente MTU - 40 bytes para headers).

Para que **TCP intente enviar paquetes grandes** (ignorando el MSS), se usa -M para deshabilitar el ajuste automático:

```
mininet> h2 iperf -s &                          #Servidor TCP
mininet> h1 iperf -c h2 -b 10M -M 1400 -t 5     #MSS= 1400
```

Wireshark

Dada las condiciones de red simuladas a través de nuestra topología podemos capturar el tráfico en las interfaces correspondientes para analizar empíricamente los siguientes fenómenos:

1. **Fragmentación**

   Generamos tráfico teniendo en cuenta el largo de los datagramas y el valor de MTU según la condición de red que queremos generar.

```
mininet> h2 wireshark &
mininet> h2 iperf -s -u &


#No se genera fragmentación
mininet> h1 iperf -c h2 -u -b 10M -l 1000 -t 5
#Forzamos la fragmentación
mininet> h1 iperf -c h2 -u -b 10M -l 3000 -t 5
```

2. **Pérdida de paquetes**

   Procedemos a capturar paquetes en h1 y h2 y luego generar tráfico entre estos.

```
mininet> h2 wireshark &
mininet> h1 wireshark &
```

- **Para TCP**: Buscamos retransmisiones.

```
mininet> h2 iperf -s &
mininet> h1 iperf -c h2 -b 10M -t 5
```

- **Para UDP**: Esperamos paquetes perdidos (h2 recibirá menos datagramas de los enviados por h1).

```
mininet> h2 iperf -s &
```

```
mininet> h1 iperf -c h2 -u -b 10M -l 1000 -t 5
```

3. **Medición de tráfico**

Se genera tráfico con iperf y se observa que llegan más paquetes que cuando no se fuerza la fragmentación (prueba: análisis 1 (fragmentación)).

Preguntas a responder

Describa la arquitectura Cliente-Servidor

Es una arquitectura en la cual el servidor es pasivo a la espera de que un cliente entable una comunicación. Para ello, el servidor se inicia primero en un puerto específico al cual el cliente se conectará. Una vez iniciado ya puede recibir múltiples conexiones de distintos clientes.

Cuando un cliente se conecta, el servidor genera un hilo para ejecutar las requests de este cliente. Todos los datagramas que el servidor recibe los mete en una cola, donde a partir de una función los datagramas se encolan en la cola del thread del cliente correspondiente. De esta manera si bien el socket pertenece al hilo principal solo el hilo particular del cliente es quien recibe los paquetes para interpretarlos.

Cuando un cliente quiere hacer un upload o download, el servidor se ocupa de comunicar estos mensajes con el hilo que corresponda y que los mensajes sean redirigidos a donde correspondan. Del lado del cliente, este ni se entera de estas particularidades y simplemente se conecta a la dirección. El servidor sabe a qué cliente enviar los paquetes ya que se guarda la dirección de envió del cliente correspondiente cuando le llega el primer paquete.

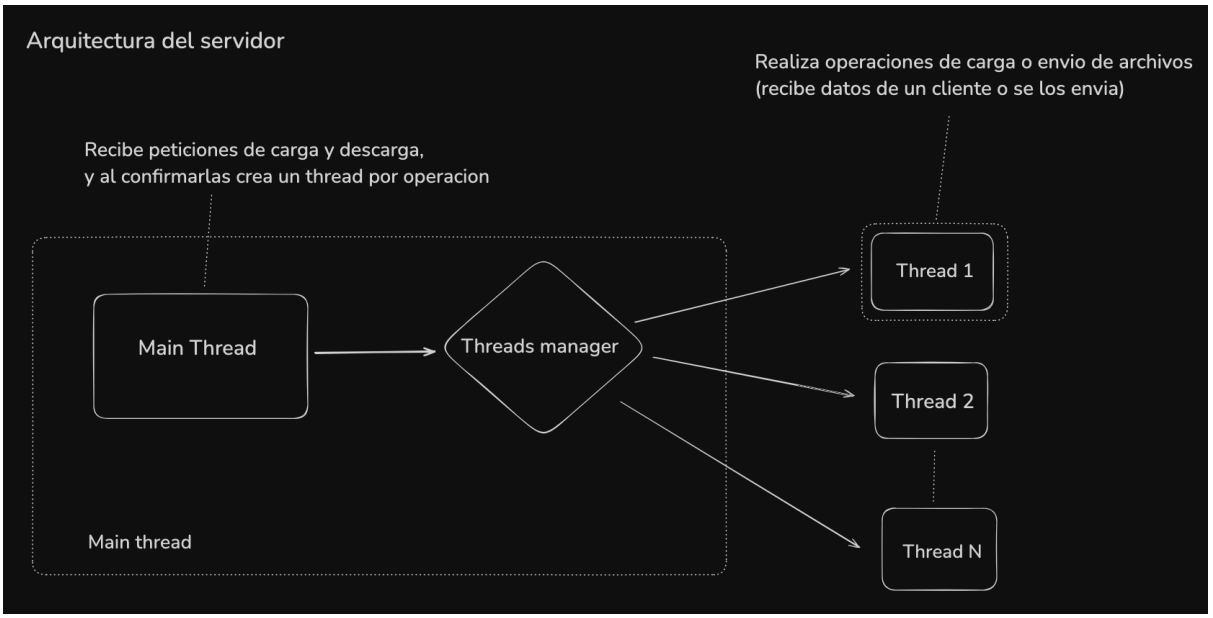


Figura 3: Arquitectura del servidor

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

Su función principal es proveer a dos aplicaciones de un protocolo de forma que se puedan comunicar sin problema y proveerles ciertas garantías. Dichas garantías dependen de lo que estemos queriendo lograr con nuestra capa.

Los protocolos definen formatos, flujos de información y criterios ante problemas que se pueden encontrar. De esta manera nos aseguramos consistencia, por ejemplo HTTP es un protocolo de capa de aplicación y en cualquier navegador que se usa se lee siempre la misma página web ya que indica cómo leer lo que recibe. El protocolo provee a todas las computadoras el mismo formato en el que recibirá la información asegurándose que de esta manera cualquier interacción de HTTP pueda ser interpretada.

Detalle el protocolo de aplicación desarrollado en este trabajo

Descrito anteriormente en el informe.

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?

TCP está caracterizado por ser fiable (es decir, garantiza que los datos lleguen completos), está orientado a la conexión, controla flujo y congestión y asegura orden en los paquetes. Provee servicio de varias cosas, principalmente Reliable Data Transfer, Control de Congestión y Control de Flujos. Es apropiado usarlo en casos donde no se quiere perder información y no hay problema de perder un poco más de tiempo y eficiencia con tal de asegurar la integridad de los paquetes, como por ejemplo HTTP.

Por otro lado, UDP está caracterizado por ser rápido y liviano. Es *connectionless*(es decir, no establece una conexión previa entre el emisor y el receptor antes de enviar los datos), no asegura ni orden, ni control de flujo, ni de congestión. Provee velocidad, comunicación sin confirmación y sin garantías y sin conexión confirmada. Si bien no puede confiarse de que va a llegar a destino sigue siendo útil donde la velocidad es lo más importante y no hay problema de perder algunos paquetes, esto se usa en DNS o en videollamadas en tiempo real donde el lag es esta pérdida, pero es mejor eso a delay por esperar.

Dificultades encontradas

1. **Fiabilidad sobre un protocolo no confiable**: Como UDP no garantiza entrega, orden ni integridad de los datos, fue necesario implementar manualmente todas las funciones de control de errores, confirmaciones, temporización y retransmisión. Aumentando la complejidad del sistema.
2. **Simulación de pérdida y retraso de paquetes**: para poder probar adecuadamente los protocolos, fue necesario simular condiciones de red adversas. Esto trajo problemas adicionales de sincronización y depuración, ya que errores intermitentes y no deterministas eran difíciles de reproducir y diagnosticar
3. **Cierre seguro de conexión**: Asegurar que el cierre de la conexión ocurriera correctamente en presencia de pérdidas de paquetes y duplicaciones fue complejo y requirió implementar reintentos y confirmaciones explícitas en ambas direcciones
4. **Depuración de redes no deterministas**: El comportamiento asíncrono y las condiciones adversas de red generaron errores difíciles de rastrear. Fue por esto que un sistema detallado de logs fue esencial para poder entender y analizar el flujo de mensajes y estados

Conclusión

La conclusión principal a la que hemos llegado es que definitivamente, el algoritmo Selective Repeat(SR) es mucho más eficiente que Stop & Wait(SW), ya que:

- Aprovecha mejor el ancho de banda disponible, ya que puede enviar múltiples paquetes, antes de necesitar confirmación de ACKs.
- Esto último, también ayuda a reducir el tiempo de espera.
- En términos de retransmisión de datos, hemos observado que en SR solo se reenvían los paquetes perdidos, en cambio, en SW si un paquete se pierde, se reenvía ese único paquete.
- Por consiguiente, esto ayuda a reducir el impacto de la pérdida de paquetes.
- En cuestiones de rendimiento(Throughput), SR tiene mucho mejor rendimiento sostenido, a diferencia de SW, cuyo rendimiento es muy bajo en redes de alta latencia o alta pérdida.

A su vez, también concluimos que SR nos fue más complejo de implementar ya que requiere ventanas deslizantes, uso de buffers, etc. Sin embargo, no se debe descartar del todo SW, ya que puede ser útil en redes muy simples o confiables, como redes locales con muy baja latencia y sin pérdida. O también en dispositivos con recursos muy limitados, como IoT, microcontroladores, etc.