FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1

[TA048] Redes
Primer cuatrimestre de 2025

Alumno	Padron	Email
AVALOS, Victoria	108434	vavalos@fi.uba.ar
CASTRO MARTINEZ, Jose Ignacio	106957	jcastrom@fi.uba.ar
CIPRIANO, Victor	106593	vcipriano@fi.uba.ar
DEALBERA, Pablo Andres	106858	pdealbera@fi.uba.ar
DIEM, Walter Gabriel	105618	wdiem@fi.uba.ar

Contents

1 Introducción

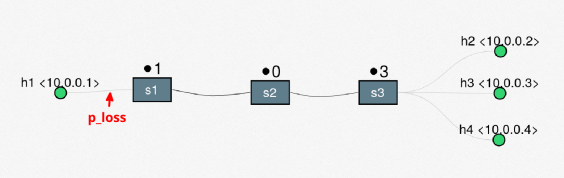
En el presente trabajo práctico se desarrolla una aplicación de red para la transferencia de archivos, basada en una arquitectura cliente-servidor, con el objetivo principal de implementar mecanismos de transferencia de datos confiable. Para ello, se trabajará sobre la capa de transporte, utilizando específicamente el protocolo UDP, lo cual requiere diseñar e implementar soluciones personalizadas que garanticen la entrega de datos confiable. En este marco, se explorarán los principios del concepto de Reliable Data Transfer (RDT), implementando dos de sus variantes: Stop-and-Wait y Go-Back-N (GBN). La comunicación entre procesos se llevará a cabo mediante la interfaz de sockets de Python, y se utilizará la herramienta Mininet para simular diferentes condiciones de red y evaluar el comportamiento de las implementaciones, incluyendo escenarios con pérdida de paquetes.

2 Hipótesis y suposiciones realizadas

- La carga/descarga no va a conservar la metadata del archivo. Es decir, si yo descargo un archivo, ese archivo va a tener metadata como si yo hubiera creado el archivo desde cero usando 'touch archivo'.
- Si el cliente utiliza otro protocolo para comunicarse con el server, el server debe rechazar este pedido. (PROTOCOL MISSMATCH). El header tendrá un campo dedicado a esto.
- El argumento de FILENAME sera opcional, en caso de no estar, se utiliza el nombre original del archivo.
- Por simplicidad, vamos a guardar todos los archivos en DIRPATH sin ningún nivel de subdirectorios.
- Si no se provee un DIRPATH para el storage del servidor, se utiliza el directorio actual.
- Se verifica que hayan 100 megas disponibles en el disco para realizar un upload al server.
- Si se cancela la descarga sin haberse finalizado correctamente, el archivo se borra.

3 Implementación

3.1 Topología



La topología es una red lineal con 1 host servidor conectado a 3 switches en serie cuyo ultimo switch esta conectado a n hosts clientes. El primer enlace (el conectado entre el servidor y el primero switch) tiene configurado un packet loss del 10% configurado de forma simetrica 5% de cada lado del enlace.

3.2 Especificación del protocolo Stop-and-Wait

3.2.1 General

1. Tamaño máximo de payload

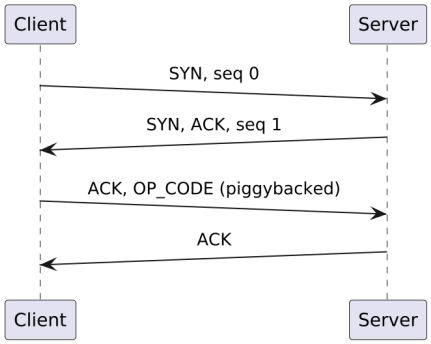
El tamaño máximo de payload es el tamaño máximo de un datagrama UDP menos el tamaño de los headers de IP, UDP y SAW.

Como el MTU que usamos en mininet es 1500, el tamaño máximo de payload es:

$\text{HISTORICAL}_{\text{MTU}} = 1500\ \text{MAX}_{\text{IPHEADERSIZE}} = 60\ \text{UDP}_{\text{HEADERSIZE}} = 8\ \text{SAW}_{\text{PROTOCOLHEADERSIZE}} = 6$

$\text{FILE}_{\text{CHUNKSIZE}} = \text{HISTORICAL}_{\text{MTU}}\ \text{MAX}_{\text{IPHEADERSIZE}}\ \text{UDP}_{\text{HEADERSIZE}}\ \text{SAW}_{\text{PROTOCOLHEADERSIZE}}$

3.2.2 Handshake



La idea es usar este handshake para inicialización de recursos del servidor y check de protocolo.

Se usa la nomenclatura S para mencionar al servidor y C para el cliente.
Mensajes para caso Download y caso Upload:

1. C → S: con flag SYN para declarar una solicitud de conexión y el protocolo

Flujo normal (mismo protocolo):

2. S → C: con flag de SYN y ACK para declarar que se acepta la conexión y el puerto donde se va a escuchar el resto.
3. C → S: con flag ACK al mismo welcoming socket.

Flujo de error (distinto protocolo):

2. S → C: con flag FIN para denegar la conexión por usar un protocolo distinto.

Se hace una transferencia de puerto para que el welcoming socket se encargue solamente de establecer conexiones y el nuevo puerto maneje la transferencia de datos del archivo. El último ACK de parte del cliente asegura que se recibió el puerto donde se tiene que comunicar y es seguro hacer el cambio de socket.

3.2.3 Etapa de configuración y Transferencia

El cliente ya sabe que tiene que comunicarse con el nuevo puerto.
Se envía primero la configuración para saber si la operación es valida y tener en cuenta casos de error, y luego se hace la transferencia.

Mensajes para caso Download y caso Upload:

1. C → S: se declara la operación (OP), que puede ser download (0) o upload (1)
2. S → C: ACK de la operación (no falla)

Continuación de mensajes para caso Download:

3. Mensaje 3 C → S: filename

Flujo Normal:

4. S → C: ACK + comienzo de datos (piggybacked)
5. C → S: ACK
6. S → C: continuacion de datos

Flujo de error (no existe un archivo con ese nombre):

4. S → C: FIN, se termina la conexión

Continuación de mensajes para caso Upload:

3. C → S: filename

Flujo de error (ya existe un archivo con ese nombre):

4. S → C: FIN, se termina la conexión

Flujo normal:

4. S → C: ACK
5. C → S: filesize

Flujo de error (archivo es más grande que el tamaño máximo o [TODO] no hay más espacio en disco):

6. S → C: FIN, se termina la conexión

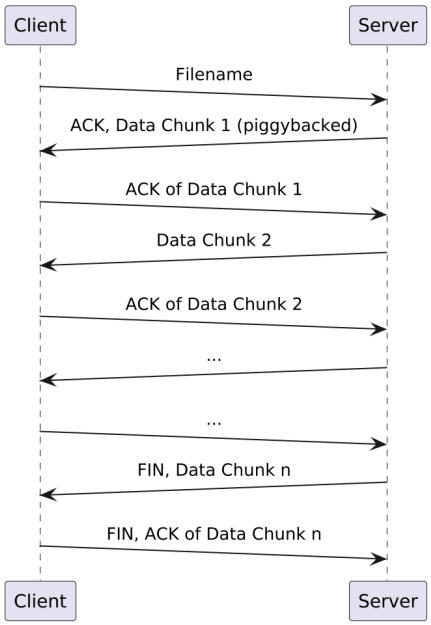
Flujo normal:

6. S → C: ACK
7. C → S: comienzo de datos
8. S → C: ACK
9. C → S: continuacion de datos

3.2.4 Cierre

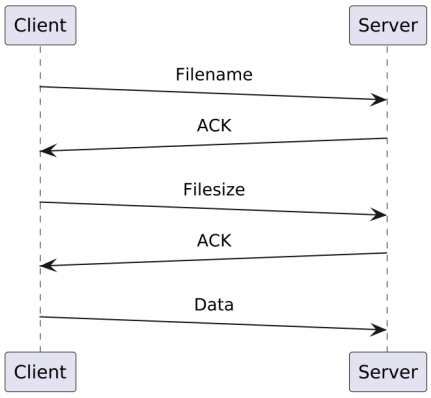
El flag FIN va piggybacked con la última data para que sea más eficiente. El receptor confirma con un ACK + FIN para que el emisor sepa que le llego la informacion, y por si este se pierde está el último ACK para confirmar el cierre de parte del emisor.

1. Mensajes para caso Download



(a) S → C: ultima data, va piggybacked el flag FIN
(b) C → S: ACK + FIN
(c) S → C: ACK

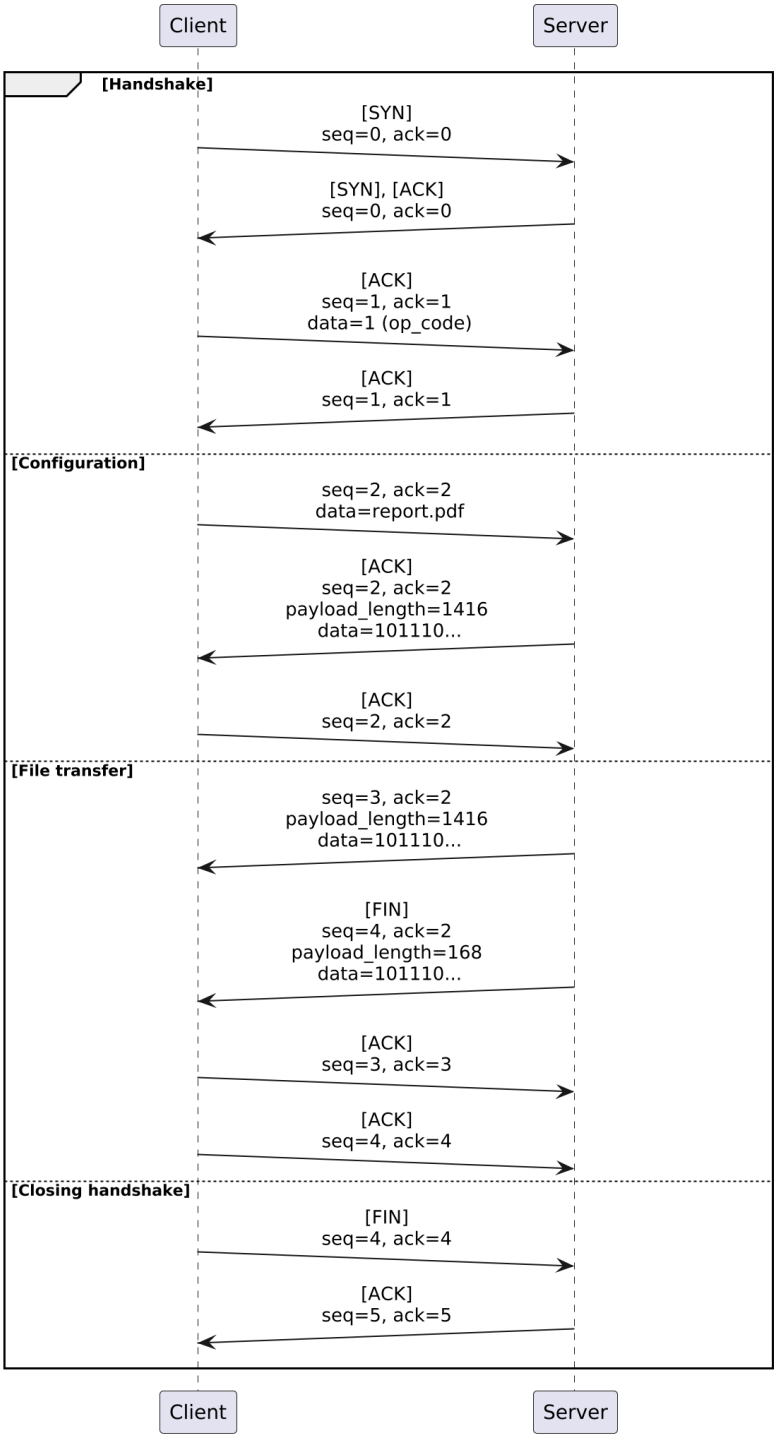
2. Mensajes para caso Upload:



(a) C → S: ultima data, va piggybacked el flag FIN
(b) S → C: ACK + FIN
(c) C → S: ACK

3.3 Especificación del protocolo Go-Back-N

3.3.1 Ciclo de Vida



Go-Back-N protocol with window of
2 packets. Client downloads file
(report.pdf of size 3000 bytes) from server

3.3.2 Análisis del ciclo de vida de una transferencia con Go-Back-N

El siguiente análisis describe el comportamiento de una transferencia de archivos mediante el protocolo Go-Back-N (GBN) con una ventana de tamaño 2. En este caso, el cliente descarga un archivo (‘report.pdf‘) de 3000 bytes desde el servidor. El ciclo de vida de la request se puede dividir en cuatro fases principales:

1. Establecimiento de la conexión (Handshake):
   - El cliente inicia la conexión enviando un paquete con las banderas ‘SYN‘, con ‘seq=0‘ y ‘ack=0‘.
   - El servidor responde con un paquete combinado ‘SYN, ACK‘ manteniendo los mismos valores de secuencia y acuse.
   - El cliente confirma la recepción enviando un paquete ‘ACK‘ con ‘seq=1‘ y ‘ack=1‘, incluyendo una operación de configuración (data=1 op_{code}).
   - El servidor responde con un ‘ACK‘ para confirmar la recepción del mensaje de configuración.
2. Configuración:
   - El cliente envía un paquete con ‘seq=2‘, ‘ack=2‘ y ‘data=report.pdf‘, indicando el nombre del archivo solicitado.
   - El servidor responde con un paquete de datos con ‘seq=2‘, ‘ack=2‘, una longitud de carga útil de 1416 bytes y los primeros bits del archivo.
   - El cliente confirma la recepción con un ‘ACK‘ correspondiente.
3. Transferencia del archivo:
   - El servidor envía el segundo fragmento de datos (‘seq=3‘, ‘ack=2‘), también de 1416 bytes.
   - Posteriormente, se envía un paquete con la bandera ‘FIN‘ (‘seq=4‘, ‘ack=2‘, ‘$\text{payload}_{\text{length}}$=168‘), marcando el fin de la transferencia.
   - El cliente responde con dos ‘ACK‘, uno para cada paquete recibido correctamente: ‘seq=3, ack=3‘ y ‘seq=4, ack=4‘.
4. Cierre de la conexión (Closing handshake):
   - El cliente envía un ‘FIN‘ para finalizar su lado de la comunicación (‘seq=4, ack=4‘).
   - El servidor responde con un ‘ACK‘ final (‘seq=5, ack=5‘), completando el cierre de la conexión de manera ordenada.

4 Pruebas

En esta sección mostraremos capturas de diferentes casos de uso de la aplicación.

4.1 Casos de error

- Protocol Mismatch

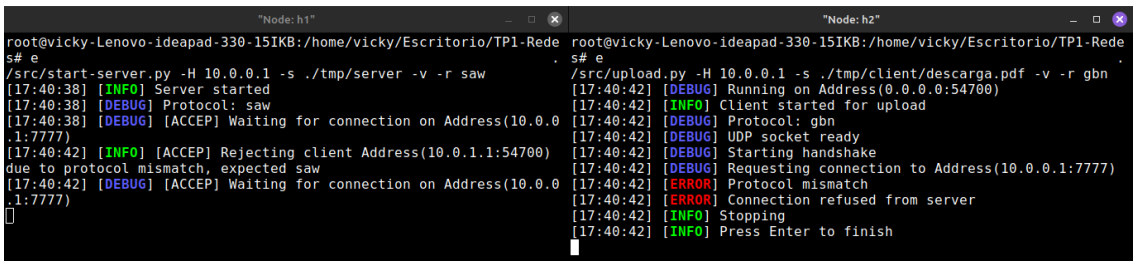


Figure 1: Ejemplo de protocol mismatch.

En caso de que un cliente intente conectarse con un servidor utilizando un protocolo diferente al suyo, el servidor lo rechazará. En la imagen se puede observar un ejemplo en el que un servidor que utiliza Stop & Wait rechaza a un cliente que hace una petición con Go Back N.

- Archivo ya existente.

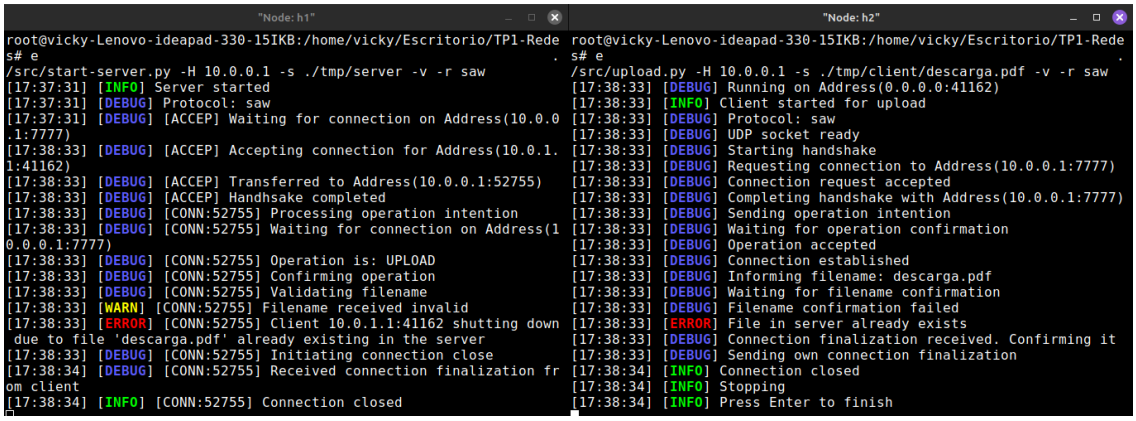


Figure 2: Ejemplo de upload de un archivo que ya existe en el servidor.

Para ambos protocolos, si el cliente intenta subir un archivo que el servidor ya tiene, se rechaza.

4.2 Stop & Wait

Para mostrar el funcionamiento de Stop & Wait, mostraremos las capturas de las operaciones upload y download de un archivo pequeño de 5kB a modo de ejemplo primero sin pérdida de paquetes, y luego con una pérdida del 10% utilizando mininet.

- Upload

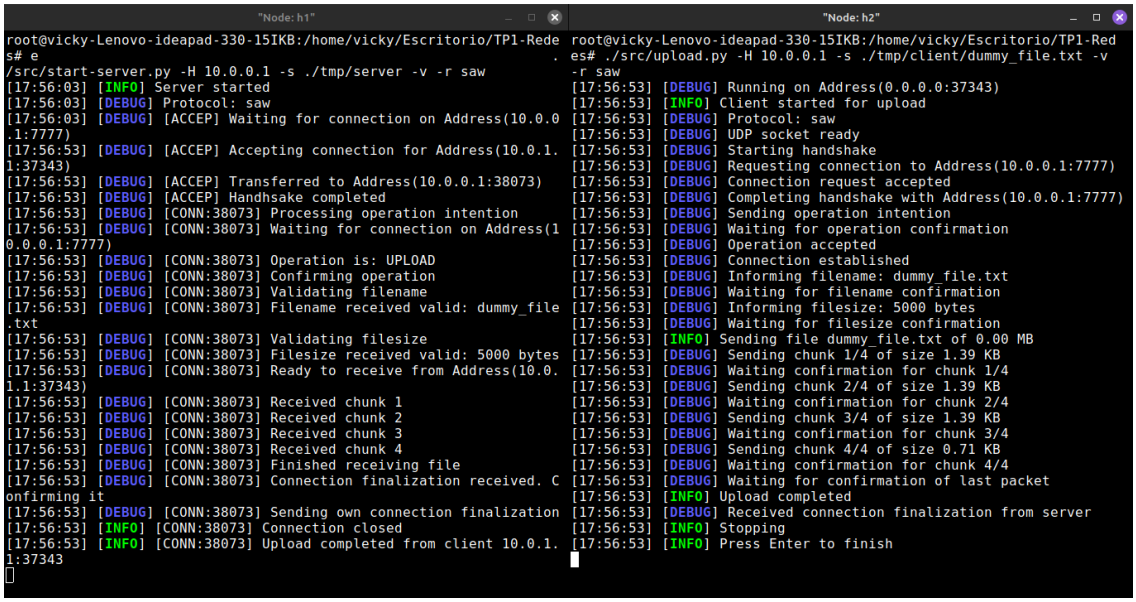


Figure 3: Captura de los logs de Upload con Stop and Wait.

No.	Time	Source	Destination	Protocol	Length	SQN SAW	Info
3	0.001276884	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[SYN]
4	0.001583021	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[SYN] [ACK]
5	0.004823484	10.0.1.1	10.0.0.1	SAW	50	1 (SAW)	[ACK] SEQ=1 Data(2)
6	0.006517231	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
7	0.009580066	10.0.1.1	10.0.0.1	SAW	62	0 (SAW)	SEQ=0 Data(14)
8	0.010016107	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
9	0.011770878	10.0.1.1	10.0.0.1	SAW	52	1 (SAW)	SEQ=1 Data(4)
10	0.012769219	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
11	0.014334757	10.0.1.1	10.0.0.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
12	0.015314610	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
13	0.015678432	10.0.1.1	10.0.0.1	SAW	1474	1 (SAW)	SEQ=1 Data(1426)
14	0.016514919	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
15	0.016722570	10.0.1.1	10.0.0.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
16	0.017761222	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
17	0.017989342	10.0.1.1	10.0.0.1	SAW	770	1 (SAW)	[FIN] SEQ=1 Data(722)
18	0.022569726	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
19	0.022816041	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
20	0.023322291	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[FIN]

Figure 4: Captura de wireshark de Upload con Stop and Wait.

- Download

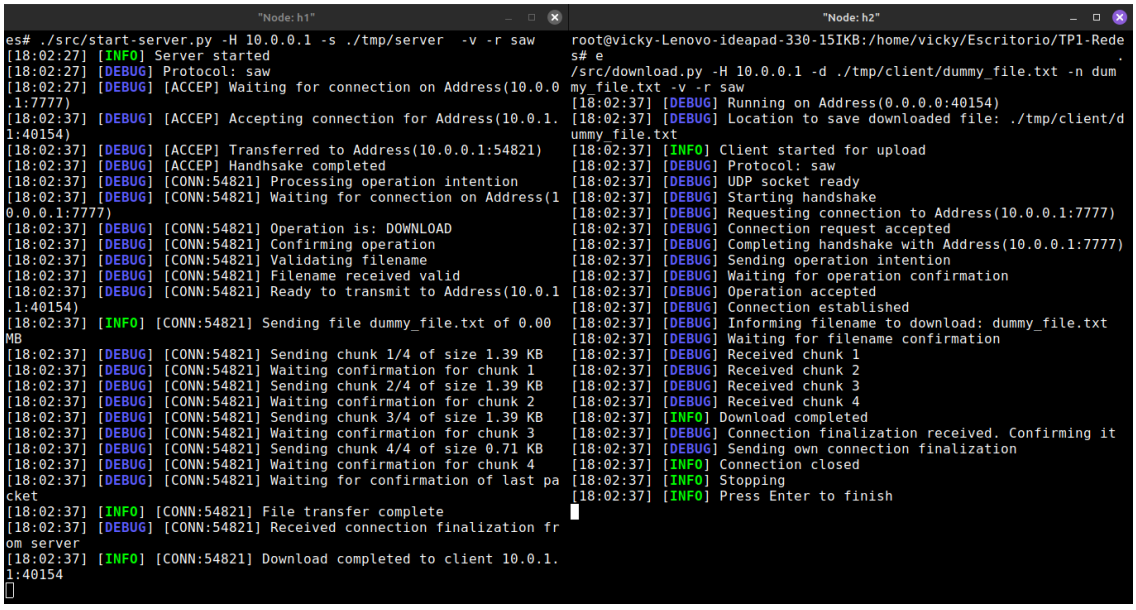


Figure 5: Captura de los logs de Download con Stop and Wait.

No.	Time	Source	Destination	Protocol	Length	SQN SAW	Info
3	0.000889970	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[SYN]
4	0.001228074	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[SYN] [ACK]
5	0.002993334	10.0.1.1	10.0.0.1	SAW	50	1 (SAW)	[ACK] SEQ=1 Data(2)
6	0.005225142	10.0.0.1	10.0.1.1	SAW	48	1 (SAW)	[ACK]
7	0.007731745	10.0.1.1	10.0.0.1	SAW	62	0 (SAW)	SEQ=0 Data(14)
8	0.008911627	10.0.0.1	10.0.1.1	SAW	1474	0 (SAW)	[ACK] SEQ=0 Data(1426)
9	0.010986390	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
10	0.011167391	10.0.0.1	10.0.1.1	SAW	1474	1 (SAW)	SEQ=1 Data(1426)
11	0.012880075	10.0.1.1	10.0.0.1	SAW	48	1 (SAW)	[ACK]
12	0.013667967	10.0.0.1	10.0.1.1	SAW	1474	0 (SAW)	SEQ=0 Data(1426)
13	0.014523036	10.0.1.1	10.0.0.1	SAW	48	0 (SAW)	[ACK]
14	0.016145495	10.0.0.1	10.0.1.1	SAW	770	1 (SAW)	[FIN] SEQ=1 Data(722)
15	0.017319266	10.0.1.1	10.0.0.1	SAW	48	1 (SAW)	[ACK]
16	0.017797019	10.0.0.1	10.0.1.1	SAW	48	0 (SAW)	[ACK]
17	0.020146182	10.0.1.1	10.0.0.1	SAW	48	1 (SAW)	[FIN]

Figure 6: Captura de wireshark de Download con Stop and Wait.

4.2.1 Tabla de Datos de Wireshark

No	Time	Src	Dst	Proto	Len	Type	SEQ	ACK	SYN	FIN	SrcPort	DstPort
1	0.000000000	10.0.1.1	10.0.0.1	SAW	48	Stop-and-Wait	0	False	True	False	52515	0
2	0.000191297	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	True	True	False	41367	0
3	0.002208402	10.0.1.1	10.0.0.1	SAW	50	Stop-and-Wait	1	True	False	False	52515	2
4	0.002801150	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	1	True	False	False	41367	0
5	0.004355272	10.0.1.1	10.0.0.1	SAW	53	Stop-and-Wait	0	False	False	False	52515	5
6	0.004722710	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	True	False	False	41367	0
7	0.005754904	10.0.1.1	10.0.0.1	SAW	52	Stop-and-Wait	1	False	False	False	52515	4
8	0.005879502	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	1	True	False	False	41367	0
9	0.006562696	10.0.1.1	10.0.0.1	SAW	1474	Stop-and-Wait	0	False	False	False	52515	1426
10	0.006634214	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	True	False	False	41367	0
11	0.006819155	10.0.1.1	10.0.0.1	SAW	1474	Stop-and-Wait	1	False	False	False	52515	1426
12	0.006887880	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	1	True	False	False	41367	0
...	...	...	...	...	...	...	...	...	...	...	...	.....
384	0.036747322	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	1	True	False	False	41367	0
385	0.036806828	10.0.1.1	10.0.0.1	SAW	1474	Stop-and-Wait	0	False	False	False	52515	1426
386	0.036860606	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	True	False	False	41367	0
387	0.037000220	10.0.1.1	10.0.0.1	SAW	1474	Stop-and-Wait	1	False	False	False	52515	1426
388	0.037084310	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	1	True	False	False	41367	0
389	0.037217987	10.0.1.1	10.0.0.1	SAW	363	Stop-and-Wait	0	False	False	True	52515	315
390	0.037459011	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	True	False	False	41367	0
391	0.037511183	10.0.0.1	10.0.1.1	SAW	48	Stop-and-Wait	0	False	False	True	41367	0
392	0.037580397	10.0.1.1	10.0.0.1	SAW	48	Stop-and-Wait	1	True	False	False	52515	0

4.3 Análisis de la comunicación Stop-and-Wait

La tabla presentada muestra una traza de paquetes intercambiados entre dos nodos de mininet (10.0.1.1 y 10.0.0.1) utilizando el protocolo desarrollado con el esquema Stop-and-Wait. Este protocolo garantiza la entrega ordenada y libre de errores mediante el envío secuencial de paquetes, esperando una confirmación (ACK) por cada uno antes de continuar con el siguiente.

El ciclo de vida de la comunicación puede dividirse en tres fases:

1. Establecimiento de la conexión:

- El cliente (10.0.1.1) inicia la conexión enviando un paquete con la bandera 'SYN' activada.
- El servidor (10.0.0.1) responde con un paquete que contiene tanto 'SYN' como 'ACK', indicando aceptación.
- Finalmente, el cliente responde con un 'ACK', completando el procedimiento de handshake.

2. Transferencia de datos:

- Una vez establecida la conexión, el cliente comienza a enviar datos, alternando los números de secuencia (SEQ) entre 0 y 1. Este comportamiento es característico del protocolo Stop-and-Wait.
- Por cada paquete de datos enviado, el servidor responde con un paquete de confirmación ('ACK') para indicar que ha recibido correctamente el contenido.
- El campo 'Len' refleja el tamaño de los datos transportados, y los puertos de origen y destino se utilizan para mantener la sesión activa entre los procesos involucrados.

3. Finalización de la conexión:

- El cliente inicia el cierre de la sesión enviando un paquete con la bandera 'FIN' activada.
- El servidor responde primero con un 'ACK', y luego con su propio paquete 'FIN', indicando que también desea cerrar la conexión.
- Finalmente, el cliente responde con un 'ACK', completando el cierre de la comunicación de manera ordenada.

En resumen, esta captura de paquetes evidencia el funcionamiento correcto de una implementación del protocolo Stop-and-Wait, en donde cada paquete enviado es seguido por una respuesta de confirmación, y el inicio y cierre de la conexión se realizan mediante el protocolo RDT.

5 Preguntas a Responder

5.1 Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es una de dos arquitecturas más comunes. En esta arquitectura hay un *host* (*end system*) llamado *server* que esta siempre encendido que pasivamente escucha *requests* de otros *hosts* llamados *clients* que son agentes activos que inician la comunicación con el *server*.

Un ejemplo de esta arquitectura es una aplicacion Web donde hay un *Web server* que escucha *requests* de navegadores web. El navegador web es el cliente que inicia la comunicación y el *Web server* es el servidor que responde a los *requests*. Estos mensajes tienen el formato de Capa de Aplicación HTTP.

5.1.1 Caracteristicas

- Los clientes son agentes activos que inician la comunicación.
- Los clientes no se comunican entre si.
- Los clientes no necesitan estar encendidos todo el tiempo ni tener una IP fija.
- Los servidores son pasivos y siempre están encendidos.
- Los servidores **deben** tener una IP fija bien conocida (*well-known IP address*) que se puede resolver con un nombre de dominio DNS (*domain name*).
- Los servidores pueden tener múltiples clientes conectados al mismo tiempo.

5.1.2 Ventajas

- Diseño simple usando protocolos sin estado como HTTP donde el servidor no necesita mantener informacion sobre clientes ya que se puede guardar informacion del cliente en *cookies* del cliente y estos se transmitidos en *headers* HTTP.
- Puede soportar un gran numero de clientes.

5.1.3 Desventajas

- Un solo punto de falla. Si el servidor se cae, el servicio se cae.
- El servidor debe estar encendido todo el tiempo.
- Gran costo para escalar, ya que a medida de que el servicio tiene mas usuarios, el servidor debe tambien aumentar su capacidad de procesar mas clientes.

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

Un protocolo de capa de aplicación especifica cómo los procesos de una aplicación, que se ejecutan en diferentes sistemas finales, intercambian mensajes entre sí. Este tipo de protocolo define:

- Los tipos de mensajes que se envían, como mensajes de solicitud y de respuesta.
- La sintaxis de los mensajes, es decir, la estructura de los campos dentro de cada mensaje y cómo se separan o identifican esos campos.
- La semántica de los campos, indicando qué significa la información contenida en cada uno.
- Las reglas de comunicación, que establecen cuándo un proceso debe enviar un mensaje y cómo debe reaccionar al recibir uno.

En resumen, los protocolos de capa de aplicación aseguran que las aplicaciones puedan comunicarse correctamente y coordinarse en la red, haciendo posible servicios como el correo electrónico, la web o la transferencia de archivos.

5.3 Detalle el protocolo de aplicación desarrollado en este trabajo.

5.4 La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP.

5.4.1 ¿Qué servicios proveen dichos protocolos?

Ambos protocolos proveen los siguientes servicios:

- **Multiplexación/Demultiplexación**: son los mecanismos que permiten extender el servicio de entrega de IP entre dos end systems a un servicio de entrega entre dos procesos que se ejecutan en esos sistemas. Dichos mecanismos permiten identificar a qué proceso pertenece cada ssegmento recibido.
- **Chequeo de integridad**: se verifica que no haya errores en los datos mediante un campo de checksum en los headers de ambos protocolos.

UDP no realiza ninguna otra función extra. Por lo tanto, su servicio es:

- **No confiable**: no garantiza que la entrega de los paquetes sea exitosa, ni tampoco que lleguen en orden.
- **Sin conexión**: cada paquete datagrama se envía de manera independiente, sin garantías de que el receptor esté listo o incluso disponible.

Por su parte, TCP ofrece las siguientes funcionalidades adicionales:

- **Orientado a la conexión**: antes de que un proceso de aplicación pueda comenzar a enviar datos a otro, ambos procesos deben comunicarse entre sí; es decir, deben enviarse algunos segmentos preliminares para establecer los parámetros de la transferencia de datos subsiguiente. Se trata de una conexión lógica con un estado en común que reside en TCP de los hosts.
- **Transferencia de datos confiable**: garantiza la entrega, el orden y la no corrupción de los datos. Esto lo logra mediante timers, numeros de secuencia y ACKs (flags que indican que un paquete fue entregado correctamente).
- **Control de congestión**: festiona que no se saturen los enlaces. Es más bien un servicio para la red.
- **Control de flujo**: para eliminar la posibilidad de que el remitente desborde el búfer del receptor. Hace coincidir la velocidad a la que el remitente envía con la velocidad a la que la aplicación receptora lee.

5.4.2 ¿Cuáles son sus características?

Algunas de las características de UDP son las siguientes:

- **Pequeño overhead de header por paquete**: UDP posee un header pequeño (8 bytes) en comparación con TCP (20 bytes)
- **Sin estado de conexión**: UDP no mantiene un estado de conexión en los end systems, por lo que no rastrea ningún parámetro. Por esta razón, un servidor dedicado a una aplicación específica generalmente puede admitir muchos más clientes activos cuando la aplicación se ejecuta mediante UDP en lugar de TCP.
- **Sin retraso por conexión**: UDP no induce ningún retraso para establecer una conexión, a diferencia de TCP que posee un handshake de tres pasos.

Por su parte, TCP posee las siguientes características:

- **Full-duplex**: dada una conexión TCP entre dos hosts, digamos A y B, la información puede fluir de A a B al mismo tiempo que fluye información de B a A.
- **Conexión point-to-point**: la conexión de TCP únicamente se puede establecer entre un único remitente y un único receptor, no admite multicasting.
- **Three-Way Handshake**: para establecer la conexión mencionada anteriormente se realiza un procedimiento donde se envían tres segmentos.

5.4.3 ¿Cuando es apropiado utilizar cada uno?

Ninguno de estos protocolos es mejor que el otro. Para decidir cuál de ellos utilizar, se deben tener en cuenta las necesidades de la aplicación. Debido a las características mencionadas anteriormente, UDP resulta más apropiado para aplicaciones que requieran mayor velocidad sin que sea tan sensible a algunas pérdidas de paquetes, por ejemplo plataformas de streaming, y si se tiene un servidor dedicado a una aplicación específica que necesita poder admitir muchos más clientes activos. Por otro lado, TCP es más ventajoso para las aplicaciones que necesitan un transporte confiable de los datos. Algunos ejemplos son el email y la web.

6 Dificultades Encontradas

7 Conclusión

8 Anexo: Fragmentacion IPv4

8.1 Consideraciones iniciales

Se propuso inicialmente una red de topología lineal que una a un host servidor, tres switches conectados en serie y un host cliente donde haya pérdida de paquetes en el enlace próximo al host receptor y que el switch central tenga la capacidad de fragmentar paquetes IPv4. Sin embargo, los switches en mininet no tienen la capacidad de fragmentar paquetes, por lo que al forzar fragmentación (es decir, reducir el MTU de alguna de sus interfaces) causaría una pérdida total de todo paquete de tamaño mayor al MTU.

La solución propuesta fue sustituir ese switch central por un "router" central, un tipo de nodo de mininet similar a los hosts pero que además cuenta con la capacidad de hacer forwarding de paquetes IPv4. Conceptualmente un switch, al ser un elemento de Link Layer, no debe tener la capacidad de fragmentar paquetes de otra capa de red.

Otro de los conflictos presentados es la comunicación bidireccional. Reducir el MTU de alguna interfaz de un router implica que se reduce sobre el enlace de esta interfaz. Y si este enlace está

conectado a un router y a un switch, la comunicación andará bien en sentido router->switch pero se perderán los paquetes en el sentido switch->router. Por lo que si se desea una comunicación bidireccional en la red y se requiere reducir el MTU de alguna interfaz de la topología esta NO debe estar vinculada a un switch, de lo contrario se perderá todo paquete que exceda el MTU reducido.

Se decidió arbitrariamente que basta con tener comunicación unidireccional (C->S) para poder provocar el fenómeno de fragmentación de paquetes IPv4. Se define una pérdida de 20% de paquetes sobre el enlace próximo al servidor, un MTU de 800 en la interfaz saliente del router (s2-eth0) y un tamaño de paquetes de 1000B (arbitrariamente, basta con que sea ligeramente superior al MTU reducido) para estudiar este fenómeno.

8.2 Análisis

8.2.1 Proceso de fragmentación

Habiendo definido la red como se mencionó anteriormente, basta generar tráfico de cliente a servidor usando iperf y capturar los paquetes recibidos y enviados por el router (s2) con Wireshark para estudiar este fenómeno. Por ejemplo:

Paquetes capturados en interfaz de entrada del router (s2-eth1): No. Time Source Destination Protocol Length Info ... 23 1.971521 10.0.1.1 10.0.0.1 TCP 74 54360 → 5001 [SYN] Seq=0 Win=43000 Len=0 MSS=1000 $SACK_{PERM}$ TSval=2993150612 TSecr=0 WS=512 24 1.991994 10.0.0.1 10.0.1.1 TCP 74 5001 → 54360 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 $SACK_{PERM}$ TSval=2142019792 TSecr=2993150612 WS=512 25 1.995183 10.0.1.1 10.0.0.1 TCP 66 54360 → 5001 [ACK] Seq=1 Ack=1 Win=43008 Len=0 TSval=2993150645 TSecr=2142019792 26 1.995677 10.0.1.1 10.0.0.1 TCP 126 54360 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=43008 Len=60 TSval=2993150646 TSecr=2142019792 27 1.995841 10.0.0.1 10.0.1.1 TCP 66 5001 → 54360 [ACK] Seq=1 Ack=61 Win=43520 Len=0 TSval=2142019813 TSecr=2993150646 28 1.996044 10.0.0.1 10.0.1.1 TCP 94 5001 → 54360 [PSH, ACK] Seq=1 Ack=61 Win=43520 Len=28 TSval=2142019813 TSecr=2993150646 29 1.996235 10.0.1.1 10.0.0.1 TCP 2042 54360 → 5001 [PSH, ACK] Seq=61 Ack=1 Win=43008 Len=1976 TSval=2993150646 TSecr=2142019792 30 1.996274 10.0.1.1 10.0.0.1 TCP 2042 54360 → 5001 [PSH, ACK] Seq=2037 Ack=1 Win=43008 Len=1976 TSval=2993150646 TSecr=2142019792 ...

Paquetes capturados en interfaz de salida del router (s2-eth0): No. Time Source Destination Protocol Length Info ... 16 1.893458 10.0.1.1 10.0.0.1 TCP 74 54360 → 5001 [SYN] Seq=0 Win=43000 Len=0 MSS=1000 $SACK_{PERM}$ TSval=2993150612 TSecr=0 WS=512 17 1.911142 10.0.0.1 10.0.1.1 TCP 74 5001 → 54360 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 $SACK_{PERM}$ TSval=2142019792 TSecr=2993150612 WS=512 18 1.914359 10.0.1.1 10.0.0.1 TCP 66 54360 → 5001 [ACK] Seq=1 Ack=1 Win=43008 Len=0 TSval=2993150645 TSecr=2142019792 19 1.914855 10.0.1.1 10.0.0.1 TCP 126 54360 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=43008 Len=60 TSval=2993150646 TSecr=2142019792 20 1.915001 10.0.0.1 10.0.1.1 TCP 66 5001 → 54360 [ACK] Seq=1 Ack=61 Win=43520 Len=0 TSval=2142019813 TSecr=2993150646 21 1.915201 10.0.0.1 10.0.1.1 TCP 94 5001 → 54360 [PSH, ACK] Seq=1 Ack=61 Win=43520 Len=28 TSval=2142019813 TSecr=2993150646 22 1.915411 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6401) [Reassembled in #23] 23 1.915413 10.0.1.1 10.0.0.1 TCP 278 54360 → 5001 [ACK] Seq=61 Ack=1 Win=43008 Len=988 TSval=2993150646 TSecr=2142019792 24 1.915415 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6402) [Reassembled in #25] 25 1.915416 10.0.1.1 10.0.0.1 TCP 278 54360 → 5001 [PSH, ACK] Seq=1049 Ack=1 Win=43008 Len=988 TSval=2993150646 TSecr=2142019792 26 1.915444 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6403) [Reassembled in #27] 27 1.915445 10.0.1.1 10.0.0.1 TCP 278 54360 → 5001 [ACK] Seq=2037 Ack=1 Win=43008 Len=988 TSval=2993150646 TSecr=2142019792 28 1.915447 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6404) [Reassembled in #29] 29 1.915448 10.0.1.1 10.0.0.1 TCP 278 54360 → 5001 [PSH, ACK] Seq=3025 Ack=1 Win=43008 Len=988 TSval=2993150646 TSecr=2142019792 30 1.915467 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6405) [Reassembled

in #31] ...

Se observa que tras establecer la comunicación inicial, se envían paquetes de un tamaño que lleva a forzar que el router fragmente los paquetes TCP recibidos en varios paquetes TCP/IPv4, esto según que tan grande sea la relación tamaño de paquete/MTU del enlace. Se logra identificar que paquete se fragmentó según los números de secuencia que muestra Wireshark en su columna de información, y para paquetes que no tienen número de secuencia coincidente indica que se requirió fragmentar más de una vez.

Si se hace una suma del tamaño de los fragmentos hasta que coincidan paquetes de ambas tablas, se observa como el router termina enviando más bytes de los que recibió. Esto debido a que el reensamblado de paquetes se produce incluyendo encabezados de todas las capas de red.

8.2.2 Funcionamiento de TCP ante la pérdida de un fragmento

TCP es un protocolo de transporte que maneja la retransmisión de datos tras pérdida y detección a partir de ACKs duplicados. Este fenómeno es muy simple de visualizar en Wireshark, donde se señala en la columna de información cuando un paquete corresponde a información retransmitida (Retransmission/Fast Retransmission)

No. Time Source Destination Protocol Length Info ... 42 2.002269 10.0.1.1 10.0.0.1 TCP 2042 54360 → 5001 [PSH, ACK] Seq=16857 Ack=1 Win=43008 Len=1976 TSval=2993150652 TSecr=2142019792 41 2.002234 10.0.1.1 10.0.0.1 TCP 2042 54360 → 5001 [PSH, ACK] Seq=14881 Ack=1 Win=43008 Len=1976 TSval=2993150652 TSecr=2142019792 43 2.002311 10.0.1.1 10.0.0.1 TCP 1054 [TCP Fast Retransmission] 54360 → 5001 [ACK] Seq=2037 Ack=1 Win=43008 Len=988 TSval=2993150652 TSecr=2142019792 44 2.002843 10.0.0.1 10.0.1.1 TCP 86 [TCP Dup ACK 34#4] 5001 → 54360 [ACK] Seq=29 Ack=2037 Win=41984 Len=0 TSval=2142019818 TSecr=2993150646 SLE=5001 SRE=8953 SLE=3025 SRE=4013 45 2.002884 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=4013 Ack=1 Win=43008 Len=988 TSval=2993150653 TSecr=2142019792 46 2.003232 10.0.1.1 10.0.0.1 TCP 66 54360 → 5001 [ACK] Seq=18833 Ack=29 Win=43008 Len=0 TSval=2993150653 TSecr=2142019792 47 2.005101 10.0.0.1 10.0.1.1 TCP 86 [TCP Dup ACK 34#5] 5001 → 54360 [ACK] Seq=29 Ack=2037 Win=41984 Len=0 TSval=2142019821 TSecr=2993150646 SLE=5001 SRE=9941 SLE=3025 SRE=4013 48 2.005135 10.0.1.1 10.0.0.1 TCP 1054 54360 → 5001 [ACK] Seq=18833 Ack=29 Win=43008 Len=988 TSval=2993150655 TSecr=2142019821 49 2.005341 10.0.0.1 10.0.1.1 TCP 94 [TCP Dup ACK 34#6] 5001 → 54360 [ACK] Seq=29 Ack=2037 Win=41984 Len=0 TSval=2142019822 TSecr=2993150646 SLE=5001 SRE=11917 SLE=18833 SRE=19821 SLE=3025 SRE=4013 50 2.005364 10.0.1.1 10.0.0.1 TCP 1054 [TCP Fast Retransmission] 54360 → 5001 [ACK] Seq=2037 Ack=29 Win=43008 Len=988 TSval=2993150655 TSecr=2142019822 51 2.005371 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=4013 Ack=29 Win=43008 Len=988 TSval=2993150655 TSecr=2142019822 52 2.005376 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=11917 Ack=29 Win=43008 Len=988 TSval=2993150655 TSecr=2142019822 53 2.005382 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=12905 Ack=29 Win=43008 Len=988 TSval=2993150655 TSecr=2142019822 54 2.005429 10.0.0.1 10.0.1.1 TCP 94 5001 → 54360 [ACK] Seq=29 Ack=4013 Win=40448 Len=0 TSval=2142019822 TSecr=2993150655 SLE=12905 SRE=13893 SLE=5001 SRE=11917 SLE=18833 SRE=19821 ...

Este mecanismo garantiza que no se pierden datos durante la comunicación en el caso de pérdida de paquetes. Para esta experiencia, como se fuerza la fragmentación de paquetes IPv4 antes del enlace con pérdida de paquetes, en su mayoría lo que se retransmite son fragmentos IPv4 perdidos, esto salvo que se pierdan datos referentes a la conexión como los ACKs de handshake/cierre de conexión u otros paquetes como la actualización de ventanas de transmisión. Dicho esto, el fenómeno de retransmisión de paquetes es el mismo para todo tipo de paquete del protocolo.

Capturando el output del comando iperf (ejecutado durante 3 segundos) se puede observar el resultado final de la comunicación en cliente y servidor:

Cliente:

Client connecting to 10.0.0.1, TCP port 5001 MSS req size 1000 bytes (per TCP_{MAXSEG}) TCP

window size: 85.3 KByte (default)

---

[1] local 10.0.1.1 port 54360 connected with 10.0.0.1 port 5001 (icwnd/mss/irtt=9/988/33411) [ID] Interval Transfer Bandwidth [1] 0.0000-1.0000 sec 130 KBytes 1.07 Mbits/sec [1] 1.0000-2.0000 sec 23.2 KBytes 190 Kbits/sec [1] 2.0000-3.0000 sec 104 KBytes 854 Kbits/sec [1] 0.0000-6.0606 sec 258 KBytes 348 Kbits/sec

Servidor:

Server listening on TCP port 5001 TCP window size: 85.3 KByte (default)

---

[1] local 10.0.0.1 port 5001 connected with 10.0.1.1 port 54360 (icwnd/mss/irtt=9/988/20275) [ID] Interval Transfer Bandwidth [1] 0.0000-7.9822 sec 222 KBytes 228 Kbits/sec

8.2.3 Funcionamiento de UDP ante la pérdida de un fragmento

TCP no es un protocolo orientado a la transferencia confiable de datos, por lo que NO cuenta con ningún mecanismo tras pérdida de paquetes, solamente envía sin garantía de recepción.

En estas capturas de Wireshark se evidencia como ocurre el fenómeno de fragmentación, donde al solo fragmentar en 2 un paquete UDP en Wireshark se capturará cerca del doble de paquetes en cada interfaz (o más según la relación tamaño/MTU):

Paquetes capturados en interfaz de entrada del router (s2-eth1): No. Time Source Destination Protocol Length Info ... 9 1.125290 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 10 1.126623 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 12 1.126743 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 11 1.126713 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 13 1.126777 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 ...

Paquetes capturados en interfaz de salida del router (s2-eth0): ... No. Time Source Destination Protocol Length Info 14 1.125302 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 15 1.126628 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=bd6e) [Reassembled in #16] 16 1.126633 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 17 1.126698 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=bd6f) [Reassembled in #18] 13 1.125298 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=bd6d) [Reassembled in #14] 18 1.126699 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 19 1.126727 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=bd70) [Reassembled in #20] 20 1.126729 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 21 1.126763 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=bd71) [Reassembled in #22] 22 1.126765 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 ...

Sin embargo, al haber pérdida de paquetes cercano al servidor, esta no puede observarse capturando sobre el router sino sobre el servidor, donde se observaría que tantos paquetes se pierden a partir de que tantos envió el cliente. Otra forma más sencilla de observar este fenómeno es con el output del comando iperf ejecutado sobre UDP, donde explícitamente señala que cantidad de paquetes se perdieron.

Cliente:

Client connecting to 10.0.0.1, UDP port 5001 Sending 1000 byte datagrams, IPG target: 800.00 us (kalman adjust) UDP buffer size: 208 KByte (default)

---

[1] local 10.0.1.1 port 55090 connected with 10.0.0.1 port 5001 [ID] Interval Transfer Bandwidth [1] 0.0000-1.0000 sec 1.19 MBytes 10.0 Mbits/sec [1] 1.0000-2.0000 sec 1.19 MBytes 9.97 Mbits/sec [1] 2.0000-3.0000 sec 1.20 MBytes 10.0 Mbits/sec [1] 0.0000-3.0003 sec 3.58 MBytes 10.0 Mbits/sec [1] Sent 3753 datagrams [1] Server Report: [ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams [1] 0.0000-3.0052 sec 2.91 MBytes 8.13 Mbits/sec 0.047 ms 698/3753 (19%) [1] 0.0000-3.0052 sec 3 datagrams received out-of-order

Servidor:

Server listening on UDP port 5001 UDP buffer size: 208 KByte (default)

[1] local 10.0.0.1 port 5001 connected with 10.0.1.1 port 55090 [ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams [1] 0.0000-3.0052 sec 2.91 MBytes 8.13 Mbits/sec 0.047 ms 698/3753 (19%) [1] 0.0000-3.0052 sec 3 datagrams received out-of-order

Donde se observa una pérdida en torno al 20% definido inicialmente.

8.2.4 Aumento de tráfico al reducirse el MTU mínimo de la red.

Por último, el fenómeno más sencillo de comprobar. Como se mencionó anteriormente, ya que el router fragmenta paquetes de ambos protocolos de transporte en función de su tamaño y el MTU de la interfaz de salida, siempre va a enviar más paquetes de los que recibe, por lo que se observa un aumento notorio del tráfico en la red. Para esta experiencia, se observa un aumento trabajando con UDP:

Paquetes capturados en interfaz de entrada del router (s2-eth1): ... No. Time Source Destination Protocol Length Info ... 3762 4.119552 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 3763 4.120970 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 3764 4.121056 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 3765 4.134780 10.0.1.1 10.0.0.1 UDP 1042 55090 → 5001 Len=1000 3766 4.140611 10.0.0.1 10.0.1.1 UDP 170 5001 → 55090 Len=128

Paquetes capturados en interfaz de salida del router (s2-eth0): ... No. Time Source Destination Protocol Length Info 7519 4.121043 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=cc13) [Reassembled in #7520] 7520 4.121046 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 7521 4.134785 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=UDP 17, off=0, ID=cc14) [Reassembled in #7522] 7522 4.134792 10.0.1.1 10.0.0.1 UDP 266 55090 → 5001 Len=1000 7523 4.140574 10.0.0.1 10.0.1.1 UDP 170 5001 → 55090 Len=128

Y se observa también un aumento en el tráfico de la red trabajando con TCP:

Paquetes capturados en interfaz de salida del router (s2-eth1): No. Time Source Destination Protocol Length Info ... 495 9.978010 10.0.0.1 10.0.1.1 TCP 78 [TCP Window Update] 5001 → 54360 [ACK] Seq=29 Ack=227301 Win=531968 Len=0 TSval=2142027795 TSecr=2993158628 SLE=228289 SRE=229277 496 9.978020 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=227301 Ack=29 Win=43008 Len=988 TSval=2993158628 TSecr=2142027795 497 9.985551 10.0.0.1 10.0.1.1 TCP 78 5001 → 54360 [FIN, ACK] Seq=29 Ack=227301 Win=531968 Len=0 TSval=2142027802 TSecr=2993158628 SLE=228289 SRE=229277 498 9.985578 10.0.1.1 10.0.0.1 TCP 66 54360 → 5001 [ACK] Seq=229277 Ack=30 Win=43008 Len=0 TSval=2993158635 TSecr=2142027802 499 10.551816 10.0.1.1 10.0.0.1 TCP 1054 [TCP Retransmission] 54360 → 5001 [ACK] Seq=227301 Ack=30 Win=43008 Len=988 TSval=2993159202 TSecr=2142027802

Paquetes capturados en interfaz de salida del router (s2-eth0): No. Time Source Destination Protocol Length Info ... 820 9.897189 10.0.1.1 10.0.0.1 TCP 278 [TCP Retransmission] 54360 → 5001 [ACK] Seq=227301 Ack=29 Win=43008 Len=988 TSval=2993158628 TSecr=2142027795 821 9.904702 10.0.0.1 10.0.1.1 TCP 78 5001 → 54360 [FIN, ACK] Seq=29 Ack=227301 Win=531968 Len=0 TSval=2142027802 TSecr=2993158628 SLE=228289 SRE=229277 822 9.904745 10.0.1.1 10.0.0.1 TCP 66 54360 → 5001 [ACK] Seq=229277 Ack=30 Win=43008 Len=0 TSval=2993158635 TSecr=2142027802 823 10.471001 10.0.1.1 10.0.0.1 IPv4 810 Fragmented IP protocol (proto=TCP 6, off=0, ID=6534) [Reassembled in #824] 824 10.471020 10.0.1.1 10.0.0.1 TCP 278 [TCP Retransmission] 54360 → 5001 [ACK] Seq=227301 Ack=30 Win=43008 Len=988 TSval=2993159202 TSecr=2142027802