FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP1: File Transfer

[75.43] Redes
Primer Cuatrimestre de 2024

Integrantes

Alumno	Padrón
GALIÁN, Tomás Ezequiel	104354
SAEZ, Edgardo Francisco	104896
PUJATO, Iñaki	109131
LARDIEZ, Mateo	107992
ZACARIAS, Victor	107080

Índice

1. Introducción

El trabajo practico consiste en implementar un protocolo que provea Relaible Data Transfer por sobre UDP, que no lo hace. En el mismo se implementa este servicio en un protocolo de aplicacion y provee dos formas de entregar los paquetes: Stop & Wait y Selective Acknowledgement. Esto debe permitir a un cliente comunicarse con el servidor para subir un archivo o para descargar uno.

Para ello se implementó datagramas, interacciones con estos, y los protocolos y su reacción ante estos. Ademas, se utilizaron timeouts y fue testeado con Mininet para simular perdidas.

2. Hipotesis y suposisiones realizadas

Las principales hipotesis fueron las siguientes. Primero se supuso que se iba a tener que crear un datagrama dentro del de UDP para poder llevar campos de control propios para implementar el RDT. Luego se supuso que lo mejor era que estos sean de tamaño fijo para simplificar su interpretacion. Por ulitmo, se tomaron varias ideas de que iba a hacer falta a partir del libro como el numero de acks, la cantidad total, tamaño y otros. A su vez, se confió en el control de integridad de los datos de UDP.

En cuanto a suposisiones, se supuso que una ventana de congestion de 5 iba a ser suficiente y tomamos arbitrariamente el tamaño de los datagramas solo teniendo en cuenta el limite de Mininet.

3. Implementacion

3.1. Stop & Wait

3.1.1. Definicion

El protocolo de Stop&Wait es la variante mas simple de las dos. Consta en enviar en orden los paquetes y no enviar el proximo hasta recibir la confirmacion del mismo mediante un ACK. El cliente comienza enviando un header dando aviso de que comienza la transferencia y espera un ACK.

Una vez recibido comienza un ida y vuelta de datagrams y respuesta de ACK. El ACK debe responder al último paquete enviado y en caso de que no sea asi, se reenvia el paquete. Del otro lado sucede lo mismo pero de forma inversa, si el que llega no es el siguiente, se lo rechaza. De esta manera se logra un avance incremental y ordenado del archivo.

Lo atractivo de Stop&Wait es que es muy simple de implementar y requiere muy poca complejidad de codigo y logica de fondo. Sin embargo, es muy ineficiente y contraproducente en la red ya que cada datagrama como minimo requerira 2 mensajes y cualquier paquete fuera de orden se pierde.

3.1.2. Implementacion

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

- El cliente envia un header que contiene el nombre del archivo que va a subir, cuantos datagrams habrán y su tamaño. Espera por un ACK del server y luego inicia el envio del archivo.
- Al iniciar comienza una iteracion de ambos lados que va creciendo de a uno a medida que la comunicacion es correcta. Es correcta para el server cuando llega el proximo paquete y es correcta para el cliente cuando recibe el ack que corresponde al ultimo que fue enviado.
- A cada envio del datagrama lo sigue de fondo un timeout en el caso de que el paquete se pierda. Si el tiemout se alcanza, se envia nuevamente ese paquete y se repite la iteración. El timeout se corre solo del lado del cliente. Esto es porque el servidor no requiere correrlo tambien ya que si el ack se pierde y no llega, el cliente solo reenviara.
- Eventualmente el servidor habrá recibido la misma cantidad de datagramas que el header aviso que llegarian y se escriben todos los bytes que fue recibidos de todos los datagrmas y se guardan en orden para rehacer el archivo enviado.

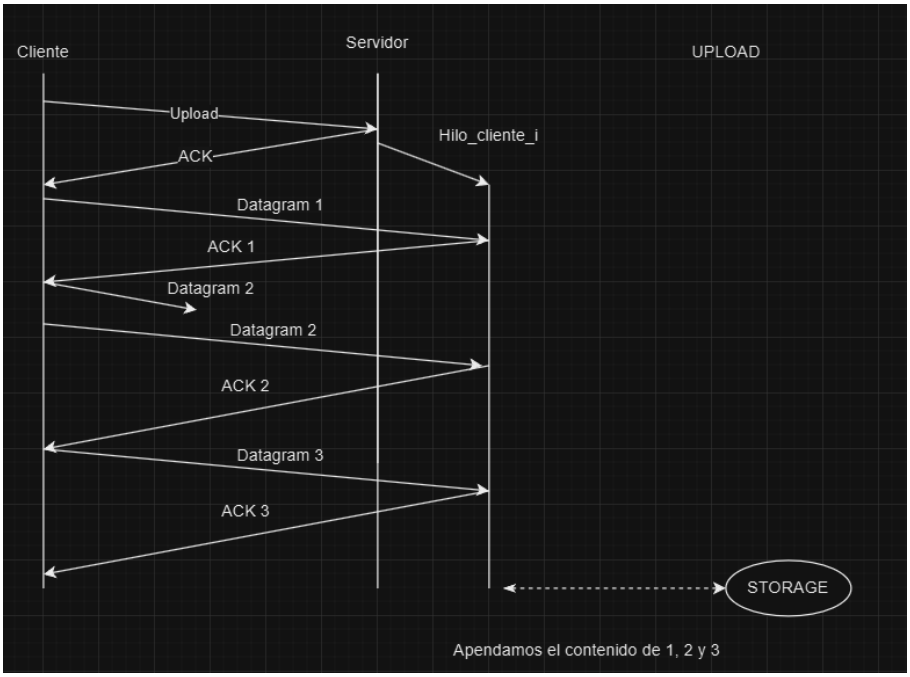

Figura 1: Diagrama del upload en Stop&Wait

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

- El cliente envia un datagrama de tipo header que contiene el nombre del archivo que desea descargar. El mimo espera un header que contiene los mismos datos que en el upload y una vez recibido confirma que esta listo enviando un ACK. Hecho esto se comienza el ida y vuelta de paquetes.
- Nuevamente se avanza de a uno rechazando y aceptando pero invirtiendo los roles. Es ahora el server quien recibe y envia los datagramas y el cliente quien envia los ACKs.
- De esta misma forma tambien se invierte quien corre el timeout y quien no, siendo nuevamente quien envia el paquete quien lo corre.

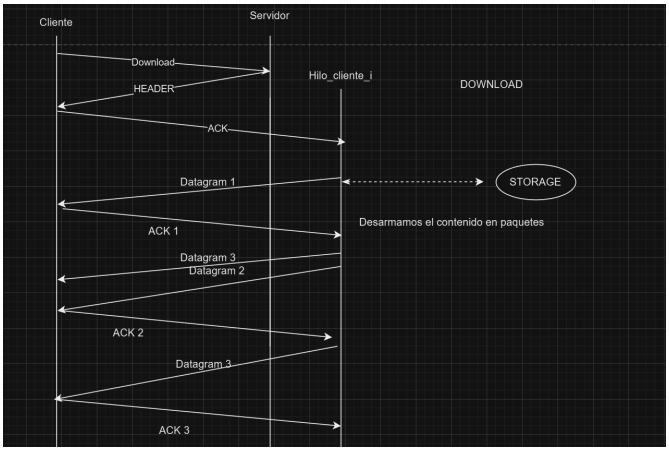

Figura 2: Diagrama del downloan en Stop&Wait

3.2. Selective Acknowledgement

3.2.1. Definicion

El protocolo de Selective Acknowledgment o en su abreviacion SACK sirve para detectar perdidas o vacios en el buffer del receptor. Cuando el receptor no recibe los datagramas en orden, el SACK identifica los faltantes, por lo que el emisor lo entiende y envia los datagramas faltantes inmediatamente.

3.2.2. Implementacion

La forma en la que se implementó SACK es la siguiente:

- El emisor envia N paquetes siendo N el tamaño de la ventana de congestion, todos seguidos. En nuestro caso N es igual a 5. El receptor envia los ACKs de los paquetes uno por uno, siempre y cuando los reciba en orden
- Cada vez que el emisor recibe un ACK, vuelve a enviar el siguiente paquete para mantener siempre la ventana de congestion en 5
- Si se pierde un paquete y no le llega a el receptor, pero le llega el siguiente a este, el receptor va a enviar siempre el ACK del ultimo paquete que le llego en orden y el SACK correspondiente. Si, por ejemplo llegaron los paquetes 0,1,2 y llega el 4 (el 3 se perdio) el receptor va a enviar al emisor ACK=3 y SACK = 4-5. En el SACK, el 4 (la primer posicion) define el primer paquete que le llego luego del hueco que hay, y el 5 (la segunda posicion) define el siguiente paquete que espera recibir.
- Si el emisor envia el paquete 5, el receptor le va a enviar ACK=3, SACK=4-6
- Cuando al emisor le llega el mismo ACK 2 veces (por ejemplo llega 2 veces el 3) detecta que hay un hueco, por lo que envia el paquete 3 nuevamente
- Cuando el emisor detecta el paquete 3, vuelven a estar todos los paquetes en orden por lo que envia ACK=6 y el SACK vacio
- Se utilizan tambien timeouts en caso de que se pierda el ACK que el receptor le envia al emisor. En este caso se reinserta el datagrama mas antiguo que se envio en la lista de datagramas que estan en vuelo, osea los que estan en la ventana de congestion

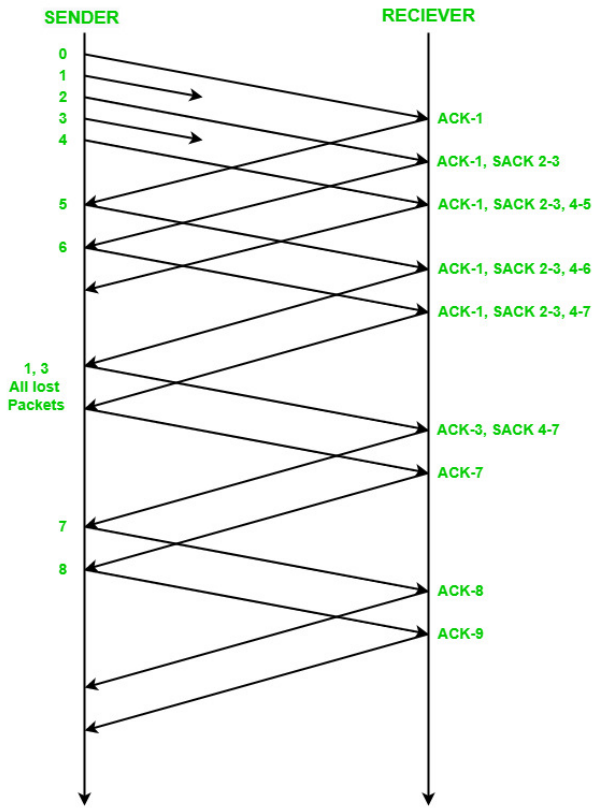

Figura 3: emisor-receptorSACK

Cuando el cliente quiere hacer un upload de un archivo, realiza el mismo procedimiento ya mencionado: envia un header que contiene el nombre del archivo que va a subir, cuantos datagrams habran y su tamaño. Espera por un ACK del server y luego inicia el envio del archivo. En este caso el cliente seria

el emisor y el server el receptor. En caso de que el cliente quiera hacer un download, el seria el receptor y el servidor el emisor

4. Pruebas

Hemos corrido varios casos de prueba pero vamos a resaltar tan solo el envio de un PDF de mas de 5MB. Se corre en ambos escenarios, con y sin perdida de paquetes, usando el algoritmo de Selective Acknowledgement y Stop&Wait.

4.1. Selective Acknowledgement

4.1.1. PDF de 5.2MB en terminal sin perdida

Aqui se corren en terminal sin nada en particular. Notar como al final imprime el Tiempo que tardo y es menos de 2 segundos, es extremadamente rapido porque no pierde paquetes justamente. Notar en el llamado que se corre Selective Acknowledgedment.

```
/usr/bin/python3.8 /home/vic/Escritorio/universidad/redes/tps/tp1/src/start_server.py -H 127.0.0.1 -p 12345 -a sack
[SERVIDOR - Hilo principal] Servidor iniciado en: 127.0.0.1:12345
[SERVIDOR - Hilo #('127.0.0.1', 40114)] Comienza a correr el thread del cliente
[SERVIDOR - Hilo #('127.0.0.1', 40114)] Recibio mensaje de: ('127.0.0.1', 40114)
[SERVIDOR - Hilo #('127.0.0.1', 40114)] Comenzando a recibir el archivo del cliente
--------------------------------------------------------------------------------
[SERVIDOR - Hilo #('127.0.0.1', 40114)] Recibido con éxito el archivo 1.pdf en 1.3292155265808105segs
--------------------------------------------------------------------------------
```

Figura 4: Terminal del servidor al terminar la recepcion

```
/usr/bin/python3.8 /home/vic/Escritorio/universidad/redes/tps/tp1/src/upload.py -H 127.0.0.1 -p 12345 -n 1.pdf -a sack
[Cliente] Iniciando conexion con el servidor: ('127.0.0.1', 12345)
[Cliente] Se desea cargar un archivo al servidor
[Cliente] Número de datagramas a enviar: 3888
[Cliente] Comenzando a enviar el archivo al servidor
[Cliente] Archivo subido correctamente en 1.3133649826049805segs
```

Figura 5: Terminal del cliente al terminar de enviar

4.1.2. PDF de 5.2MB en Mininet

En este caso se corren en Mininet simulando una perdida del 10 %. Para ello nuevamente se corre este archivo y se utiliza el SACK como protocolo. Vemos aca como el tiempo sube a 25 segundos aproximadamente. Esto se debe a que con la perdida habran mas idas y vueltas, pero igualmente es un buen tiempo.

```
root@vic-IdeaPad-3-14IIL05:/home/vic/Escritorio/universidad/redes/tps/tp1# python3 src/start_server.py -H 10.0.0.1 -p 12345 -a sack
[SERVIDOR - Hilo principal] Servidor iniciado en: 10.0.0.1:12345
[SERVIDOR - Hilo #('10.0.0.2', 49153)] Comienza a correr el thread del cliente
[SERVIDOR - Hilo #('10.0.0.2', 49153)] Recibio mensaje de: ('10.0.0.2', 49153)
[SERVIDOR - Hilo #('10.0.0.2', 49153)] Comenzando a recibir el archivo del cliente
--------------------------------------------------------------------------------
[SERVIDOR - Hilo #('10.0.0.2', 49153)] Recibido con éxito el archivo src/1.pdf en 24.655217409133915segs
--------------------------------------------------------------------------------
```

Figura 6: Terminal del servidor al finalizar la recepcion

Figura 10: Ambas terminales al finalizar Mininet

Figura 7: Termianl del cliente al finalizar el upload

4.2. Stop&Wait

4.2.1. PDF de 5.2MB en terminal sin perdida

Aqui se corren en terminal sin nada en particular. Notar como al final imprime el Tiempo que tardo y es menos de 1 segundo, es extremadamente rapido porque no pierde paquetes justamente. Notar en el llamado que se corre Stop&Wait. En este caso SW es mas rapido porque al no haber perdidas la logica simple es mas veloz que la logica compleja de SACK.

```
/usr/bin/python3.8 /home/vic/Escritorio/universidad/redes/tps/tp1/src/start_server.py -H 127.0.0.1 -p 12345 -a sw
[SERVIDOR - Hilo principal] Servidor iniciado en: 127.0.0.1:12345
[SERVIDOR - Hilo #('127.0.0.1', 36196)] Comienza a correr el thread del cliente
[SERVIDOR - Hilo #('127.0.0.1', 36196)] Iniciando conexión con el cliente
[SERVIDOR - Hilo #('127.0.0.1', 36196)] El cliente solicita cargar el archivo: 1.pdf
[SERVIDOR - Hilo #('127.0.0.1', 36196)] Solicitud de carga aceptada
[SERVIDOR - Hilo #('127.0.0.1', 36196)] Iniciando recepción de archivo
[SERVIDOR - Hilo #('127.0.0.1', 36196)] Recibido con éxito el archivo 1.pdf en 0.15749454498291016segs
```

Figura 8: Terminal del servidor al finalizar la recepcion

```
/usr/bin/python3.8 /home/vic/Escritorio/universidad/redes/tps/tp1/src/upload.py -H 127.0.0.1 -p 12345 -n 1.pdf -a sw
[Cliente] Solicitud de carga de archivo enviada al servidor
[Cliente] Solicitud de carga de archivo aceptada por el servidor

Process finished with exit code 0
```

Figura 9: Termianl del cliente al finalizar el upload

4.2.2. PDF de 5.2MB en Mininet

En este caso se corren en Mininet simulando una perdida del 10 %. Para ello nuevamente se corre este archivo y se utiliza el SW como protocolo. Vemos aca como el tiempo sube a 95 segundos aproximadamente. Aca se ve la diferencia de como SACK es mejor cuando efectivamente hay perdida, sacando a relucir las ventajas de su complejidad. Esto se debe a que SW tiene mucha latencia o tiempo de espera y tiene limitacion de envio ya que envia un paquete y espera a recibirlo y si no lo recibe lo vuelve a enviar. Mientras que SACK puede enviar mas de un paquete a la vez y no debe recibir si o si en orden los ACK de los paquetes. Ademas, Stop&Wait tiene mucho tiempo de espera donde no sucede nada entre paquete enviado a espera de recepcion.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

La arquitectura cliente servidor comienza de forma similar a la clasica donde el servidor es pasivo a espera de que un cliente comienze una comunicacion. Para ello, se inicia primero el server en un puerto especifico al cual el cliente se conectara. Una vez iniciado ya puede recibir multiples conexiones de distintos clientes.

Cuando un cliente se conecta, el servidor genera un hilo para ejecutar las requests de este cliente. Todos los datagrams que el servidor recibe los mete en una cola, donde a partir de una funcion 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 redirijidos a donde correspondan. Del lado del cliente, este ni se entera de estas particularidades y simplemente se conecta a la direccion. El servidor sabe a que cliente enviar los paquetes ya que se guarda la direccion de envio del cliente correspondiente cuando le llega el primer paquete.

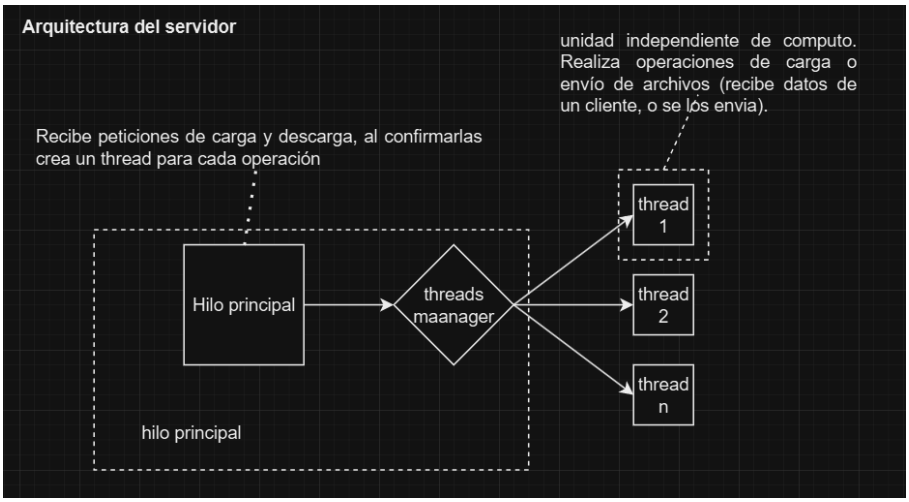

Figura 11: Arquitectura del servidor

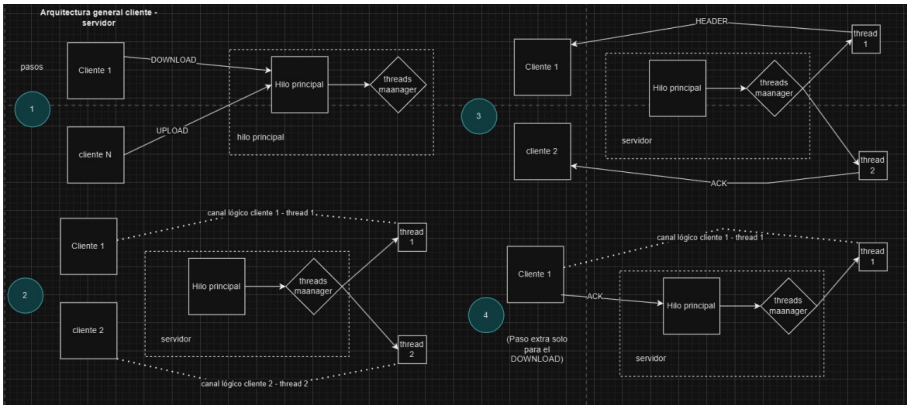

Figura 12: Arquitectura Cliente-Servidor

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

La funcion de un protoclo de capa de aplicacion es proveer a dos aplicaciones de un protoclo de forma que se puedan comunicar sin problema y proveerles de ciertas garantias. Estas garantias dependen de lo que estemos queriendo lograr con nuestra capa y es fundamental cumplirlas para que se pueda dar el sistema de capas.

Los protoclos definen formatos, flujos de informacion y criterios ante problemas que se pueden encontrar. De esta manera nos aseguramos consistencia, por ejemplo HTTP es un protoclo de capa de aplicacion y en cualquier navegador que se usa se lee siempre la misma pagina web ya que indica como leer lo que recibe. El protoclo le provee a todas las computadoras el mismo formato en el que recibira la informacion asegurandose que de esta manera cualquier interaccion de HTTP pueda ser interpretada. Es analogo a un idioma entre dos personas, permitiendo que las palabras tengan o no sentido.

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

En este trabajo particularmente se desarrollo un protocolo que consta de la interpretacion de un datagrama que provea RDT sobre UDP. UDP se ocupa de todo lo que es la logica de envio pero creamos nuestros datagramas para agregarle la confiabilidiad que ofrecen otros protoclos como TCP. Le agregamos chequeos respecto de si el datagrama que acaba de llegar es el esperado, si es que se perdio algun datagrama de ambos lados y que esten todos.

Esto lo logramos a partir del uso de dos estructuras: SWDatagrams y SackDatagrams. Estos se usan para cada algoritmo correspondiente ya que los campos que traen no son los mismos. SACK es para Selective Acknowledgement y SW para Stop&Wait. Principalmente estos datagramas proveen toda la informacion de control que requiere el protocolo para proveer la RDT, ya que es lo unico que nuestro protocolo provee.

En el caso de SW se necesita solo saber el numero de paquete actual y el previo por lo que precisa menos campos de control, mientras que SACK requiere los selective acks y cuantos son, por lo que agrega eso a su datagram haciendolo un poco mas complejo.

Algo importante a denotar es que nuestros datagrams, no importa si son HEADER o ACK o Contenido todos tienen el mismo tamaño por lo que hay campos que a veces estan en uso y a veces estan vacios, pero que permiten mayor facilidad a la hora de leer estos paquetes.

En las siguientes imagenes se pueden ver las estructuras de los datagrams para SW y SACK, notandose la difencia de campos SACK y nro de SACK

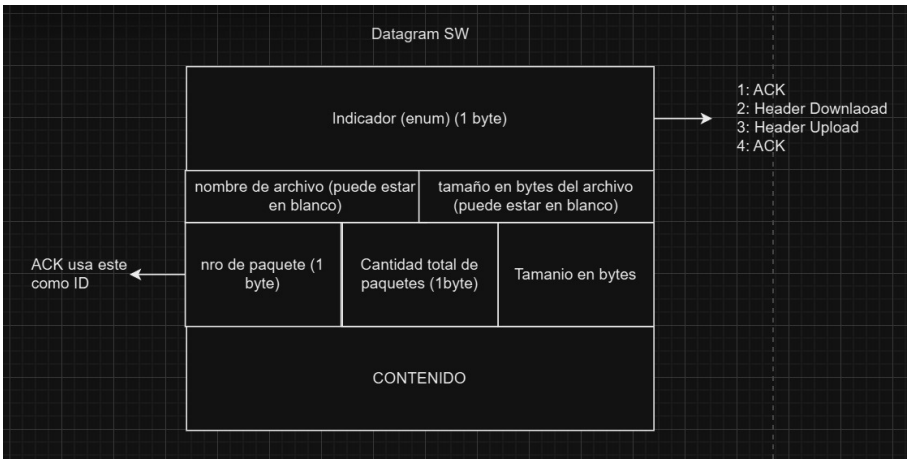

Figura 13: SW Datagram

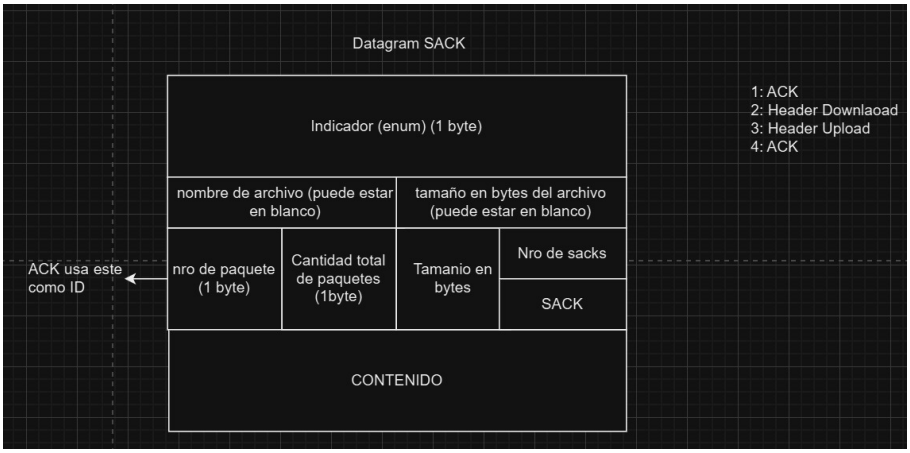

Figura 14: SACK Datagram

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

TCP esta caracterizado por ser fiable, esta orientado a la conexion, controla flujo y congestion y asegura orden en los paquetes. Provee servicio de varias cosas, principalmente Reliable Data Transfer, Control de Congestion y Control de Flujos. Princpalmente es elegido en casos donde no se puede perder informacion y no hay problema de perder un poco de tiempo y eficienca con tal de asegurar la integridad de la informacion, como por ejemplo HTTP.

Por el lado de UDP esta caracterizado por ser rapido y liviano, es connectionless y no asegura ni orden, ni control de flujo ni de congestion. Provee velocidad, comunicacion sin confirmacion y sin garantias y sin conexion confirmada. Si bien no puede confiarse de que va a llegar a destino sigue siendo util donde la velocidad es lo mas 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 perdida, pero es mejor eso a delay por esperar.

6. Dificultades Encontradas

La primera dificultad que encontramos fue en como fijar el tamaño de los datagramas en un lenguaje tipado como lo es Python, una vez que lo logramos la comunicacion se volvio mas facil y simple.

A su vez, a medida que implementamos el SACK (que por definicion nos fue dificil) vimos tambien que ibamos a necesitar hacer modificaciones importantes al datagrama unificado que usabamos hasta el momento. Fue ahi donde decidimos que ibamos a tener dos tipos de datagramas porque considerabamos que la diferencia a SW lo iba a perjudicar tangiblemente por campos que siempre iban a estar en desuso.

Por ulitmo la logica de comunicacion del SACK nos fue un poco dificl mas que nada en cuanto a los peridodos que ya se habian recibido, pero pudimos sacarlo adelante a base de intentos y debuggeo.

7. Conclusion

La conculision principal que podemos encontrar es que efectivamente pudimos lograr crear un protocolo RDT sobre UDP, pero que el algoritmo que le demos trae consigo un precio que debemos saber elegir. Particularmente, SACK es mas eficiente en todo sentido de las comunicaciones, pero esta eficiencia viene a un costo de complejidad que no puede ser obviado. Si nuestra red sufre muy poco trafico y no sera lo normal perder paquetes, entonces es posible que el extra de procesamiento sea mayor a un reenvio ineficiente como el SW. Por el lado de SW vimos que implementar RDT es algo realmente simple, pero sin duda es un algoritmo muy ineficiente y no recomendable de uso excepto en condiciones excepcionales.

Se podria concluir en que si sabemos que se van a perder pocos paquetes en la transferencia de datos, conviene utilizar el algoritmo Stop&Wait, pero si tenemos un alto porcentaje de perdida de paquetes, sin dudas combiente utilizar Selective ACK. En la vida real, siempre triunfara el caso de Selective ACK como protocolo triunfante contra Stop&Wait, pero tampoco debe ser descartado.