INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS (75.43)

Trabajo Práctico N°1: FILE TRANSFER

Alumno	**Padrón**
Camila Fernández Marchitelli	102515
Lucas Aldazabal	107705
Bautista Boselli	107490
Alejo Fábregas	106160
Camilo Fábregas	103740

INTRODUCCIÓN

El objetivo de este trabajo práctico es la creación de una aplicación de red para transferencia de archivos, que implemente el modelo cliente-servidor y un protocolo RDT (Reliable Data Transfer).

Para ello, se deberá emplear una arquitectura donde el cliente inicia la conexión con el servidor, le envía peticiones y el servidor le responde con los datos solicitados. La implementación de esta aplicación deberá ser utilizando sockets UDP de Python, para poder lograr una comunicación entre hosts. También se deberán aplicar algunos principios básicos de la transferencia de datos confiable, para poder asegurarse que la información llegue siempre y de forma segura y sin fallas.

Se deberán implementar dos versiones: una que utilice el protocolo Stop & Wait, y otra que utilice el protocolo Selective Repeat. La funcionalidad de la aplicación deberá ser la siguiente:

- **SERVER:** servidor que pueda almacenar y proveer archivos a sus clientes.
- **UPLOAD:** transferencia de un archivo del cliente hacia el servidor.
- **DOWNLOAD:** transferencia de un archivo del servidor hacia el cliente.

Además, el servidor deberá ser capaz de atender a distintos clientes de forma paralela. Para ello, se deberá aplicar técnicas de programación concurrente, utilizando los threads de Python.

También se deberá utilizar Wireshark para poner a prueba el rendimiento de la solución implementada, ya sea observando distintas métricas o analizando el tráfico de paquetes en la red.

HIPÓTESIS Y SUPOSICIONES REALIZADAS

- Para evitar fragmentación de paquetes se utiliza como tamaño de paquete 1472 bytes que, sumado a los 20 bytes del header del datagrama y los 8 del header UDP, da 1500 que es el MTU (maximum transfer unit) que tanto Ethernet como protocolos punto a punto de la capa de enlace utilizan habitualmente para enviar datagramas, de esta forma evitamos fragmentación de nuestros paquetes.

- Dado la hipótesis anterior, se decidió tomar como tamaño máximo de los datos enviados en un paquete 1400 bytes, de esta forma la implementación queda abierta a una futura expansión del header implementado.

- El tamaño de la window para selective repeat se definió en 15. Este valor se obtuvo al realizar fine tuning con un archivo de 5 MB y un loss del 10%.

- El tamaño máximo del archivo es 4 GB ya que se destinan 4 bytes en el handshake para el tamaño del archivo (Es decir, 32 bits).

- Los valores de los timeouts utilizados en general se obtuvieron al realizar fine tuning con un archivo de 5 MB y un loss del 10%.

IMPLEMENTACIÓN

Responsabilidades

A continuación se detallarán cuáles son las responsabilidades tanto del receptor como del emisor.

Lado receptor:
Interfaz/Config: Cliente/Servidor se generan con sus datos y config
Cliente/Servidor: Hace el handshake para enviar el archivo (es receiver). Recibe la estrategia del servidor y usa el Receiver específico de dicha estrategia
Receptor Estrategia: El Receiver de la estrategia maneja los mensajes, windows, timeouts según la estrategia elegida.
Cliente/Servidor: Guarda lo recibido o imprime por consola en caso de que haya algún error.

Lado emisor:
Interfaz/Config: Cliente/Servidor se generan con sus datos y config
Cliente/Servidor: Hace el handshake para enviar el archivo (es sender). Recibe la estrategia del servidor y usa el Sender específico de la estrategia elegida y genera los chunks de la data binaria a enviar.
Emisor Estrategia: El Sender de la estrategia maneja los mensajes, windows y timeouts según la estrategia.

En ambos casos, el servidor, al recibir el primer handshake hace el nuevo thread.

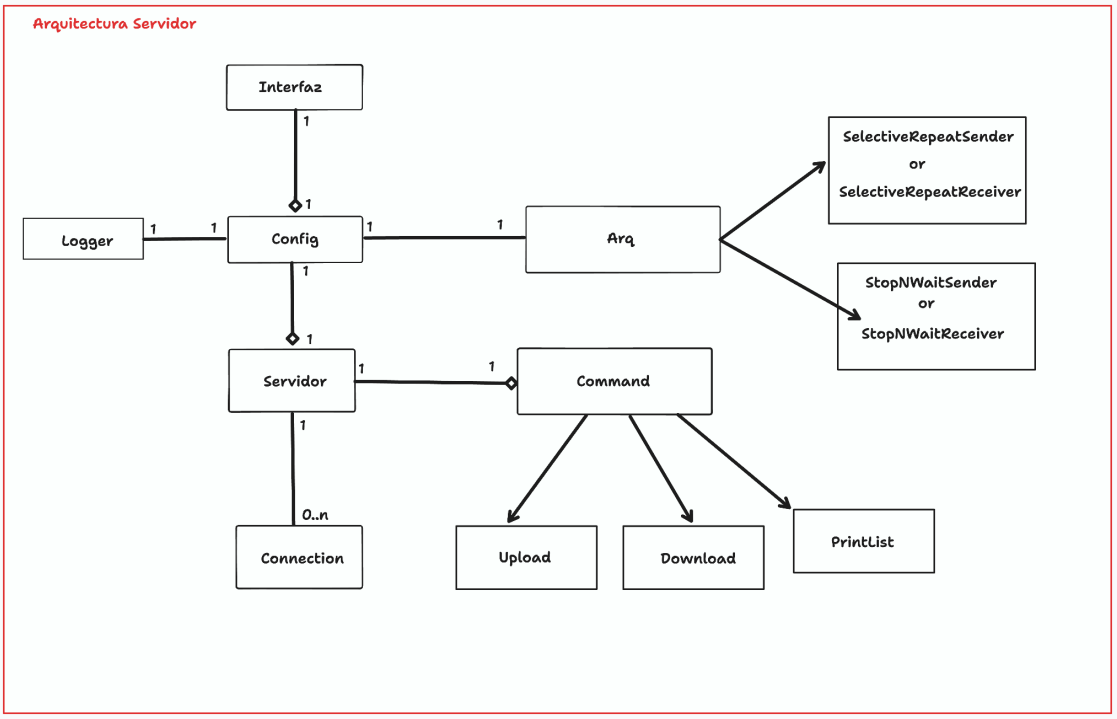
Overview de cómo es la comunicación dependiendo el tipo de comando elegido

A continuación se detallan con dos esquemas como es a gran escala el intercambio de mensajes entre cliente/servidor dependiendo el comando que se haya elegido.
Este intercambio de mensajes es independiente de la estrategia elegida.

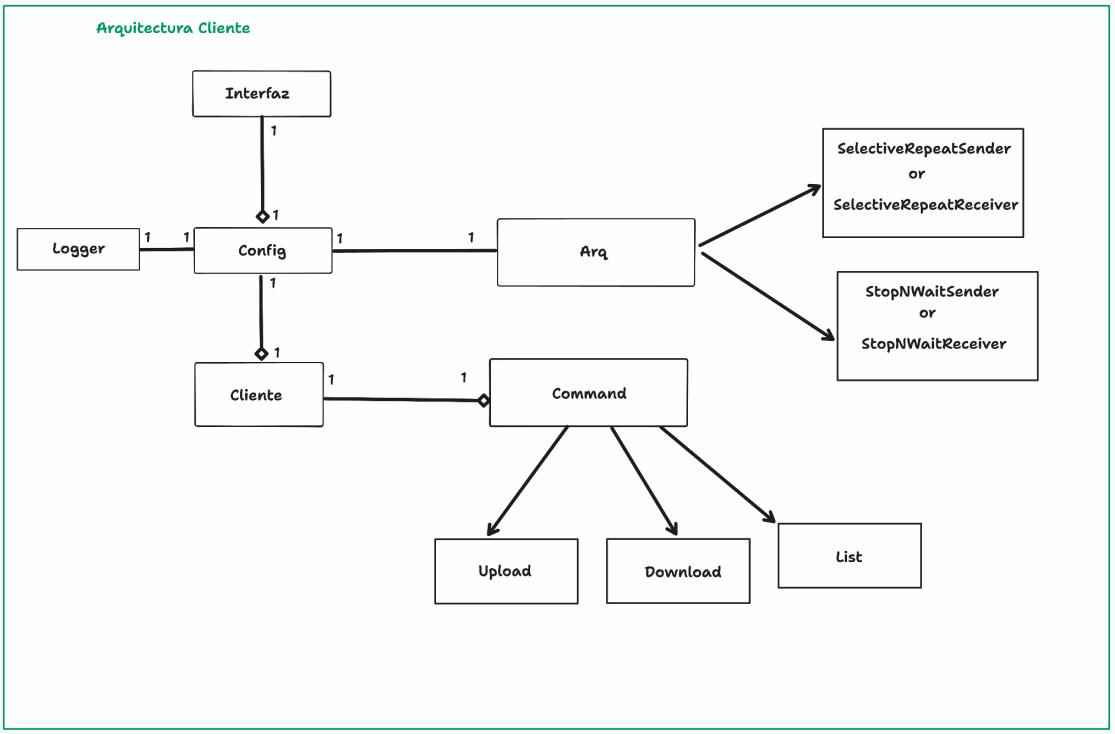
Arquitectura

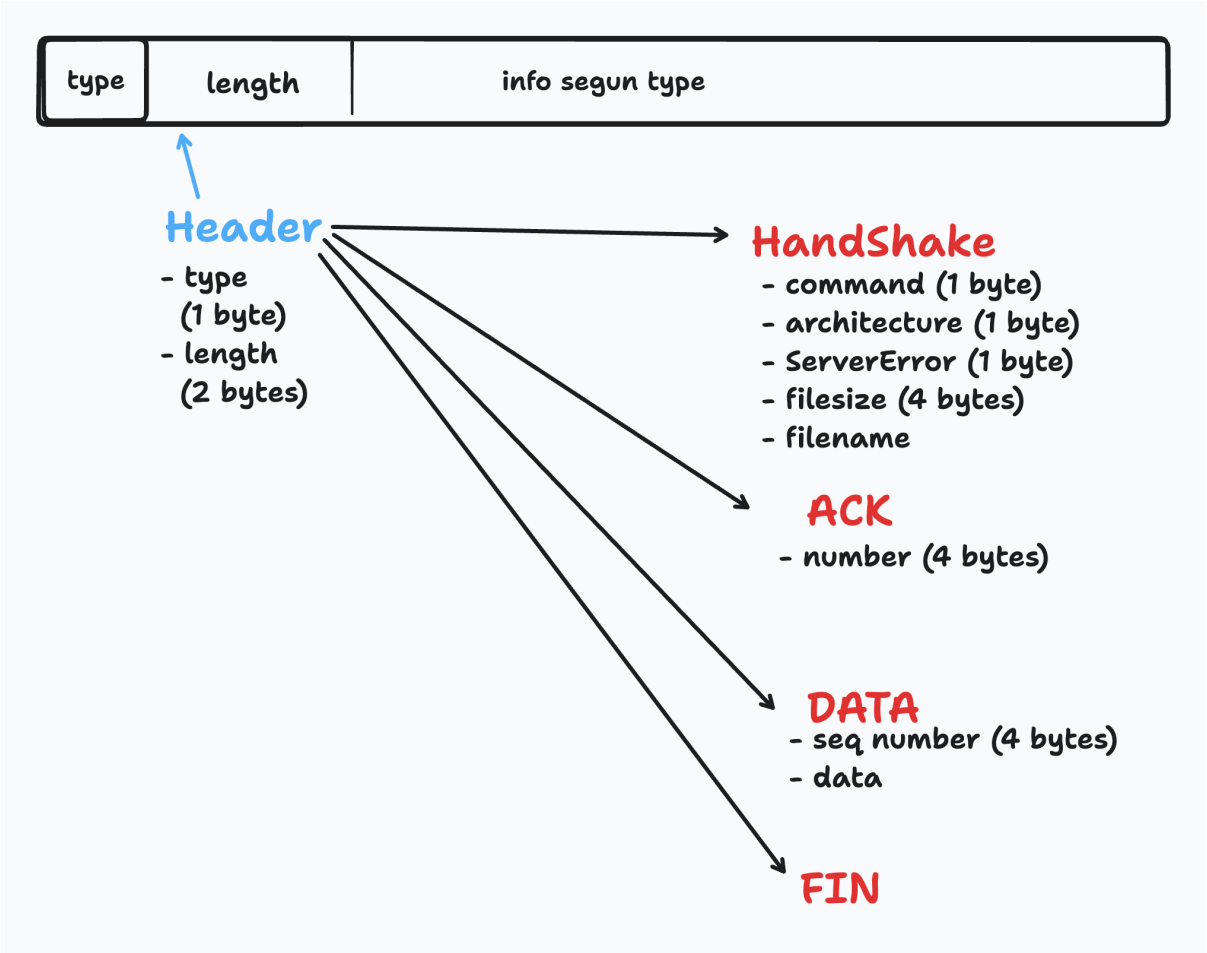
A continuación se mostrarán dos diagramas en donde se puede ver como es la arquitectura del programa desde el lado del cliente y desde el lado del servidor.

Servidor



Cliente





Descripción de las clases:

Interfaz: Es la clase encargada de interactuar con el usuario, la cual tiene las siguientes responsabilidades:

- Guarda todos los parámetros que se ingresen con línea de comandos
- Verifica que se ingresen los parámetros obligatorios
- Imprime el menú de ayuda del programa.

Config: Es la clase encargada de validar los argumentos de la interfaz, y se crea una instancia por cada cliente y una instancia en el servidor. Además cada instancia crea un logger para escribir sobre un archivo (que cambia si la instancia es de un Server o Cliente) todos los eventos que sucedan

Logger: Esta clase se encarga de escribir en un archivo los eventos que sucedan durante la transferencia de mensajes. Este archivo se crea uno para el servidor y otro por cada cliente que se conecte al mismo, se identifican según el momento exacto en

el que se conectan al servidor. Se puede configurar por terminal con quiet o verbose si aparte se imprime por la terminal el detalle de estos archivos mientras se escriben

Server: Es la clase que representa una abstracción del servidor y tiene las siguientes responsabilidades:

- Guarda en un diccionario los clientes que se conecten al mismo usandolos como clave del diccionario y como valor guarda un thread y una cola para manejar la conexión con el cliente
- Guarda una referencia a la instancia de la clase Config
- Guarda una referencia al socket de donde escucha nuevos mensajes y si es uno nuevo lo agrega al diccionario con su respectiva cola/thread y si es uno ya guardado pone en la cola de ese cliente el mensaje escuchado en el socket

Connection: Es la clase que representa nuestra abstracción de la conexión que mantiene el Servidor con un determinado cliente y tiene como responsabilidades:

- Ejecutar el handshake entre el server y cliente, determinando la arquitectura, comando (si es descarga o subida), nombre y tamaño del archivo y el host y puerto del cliente
- Mandar a ejecutar el comando indicado en el handshake

Command: Según los tipos de comandos utilizados (Download/Upload) se crea una instancia de las siguientes clases:

- Upload: Esta clase se instancia en 2 situaciones y lo que hace es: se abre el archivo a subir, se lo guarda en un buffer, y se crean chunks de los datos a enviar.
    - Cuando el comando que el cliente quiere ejecutar es “upload”, esta instancia se crea para el cliente y el envío de datos lo realiza la instancia del Sender que varía con la arquitectura (StopNWaitSender o SelectiveRepeatSender)
    - Cuando el comando que el cliente quiere ejecutar es “download”, esta instancia se crea para el servidor que busca mandar el archivo al cliente, nuevamente el envío de datos depende de las clases previamente mencionadas (StopNWaitSender o SelectiveRepeatSender)
- Download: Esta clase se instancia en 2 situaciones y lo que hace es: recibir los datos enviados, abrir el archivo y escribir en él los datos en orden
    - Cuando el comando que el cliente quiere ejecutar es “download”, esta instancia se crea para el cliente. La recepción de los chunks de datos depende del Receiver que varía con la arquitectura (StopNWaitReceiver o SelectiveRepeatReceiver)

    - Cuando el comando que el cliente quiere ejecutar es “upload”, esta instancia se crea para el servidor que busca recibir los datos enviados por el cliente. Nuevamente la recepción de los chunks de datos depende del Receiver que varía con la arquitectura (StopNWaitReceiver o SelectiveRepeatReceiver)
- List: Este comando es usado para listar en el cliente los archivos que tiene el servidor en su storage.
    - En el cliente se utiliza un PrintList como command que se encarga de hacer la recepción con la arquitectura configurada (StopNWaitReceiver o SelectiveRepeatReceiver) e imprime el texto recibido.
    - En el servidor se utiliza List como command que se encarga de generar los chunks con el os.listdir de python para enviar con la arquitectura configurada (StopNWaitSender o SelectiveRepeatSender) los archivos que tiene en el storage.

Arq: Es la clase que representa el tipo de protocolo que se desea usar para el envío o recepción de datos. Por default, si no se indica se usa siempre “stop and wait”. Dependiendo del tipo de comando elegido, la arquitectura indicada por terminal y el rol que tenga que cumplir el server/cliente, la arq puede ser:

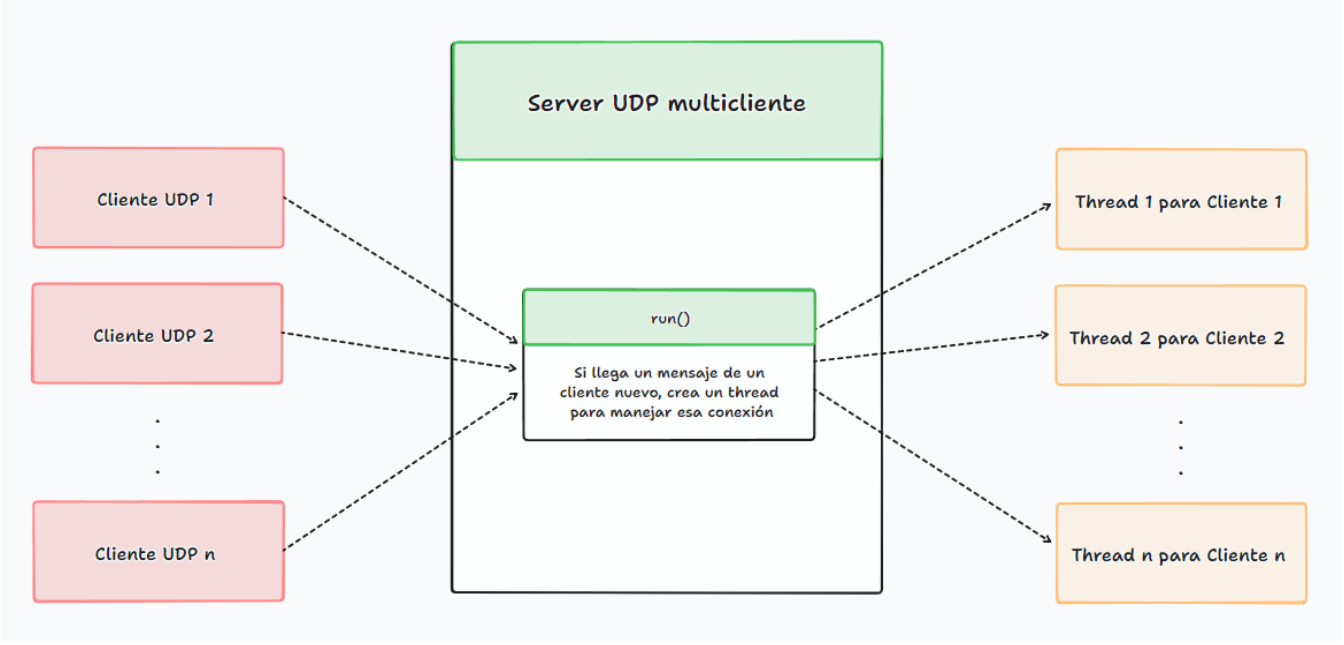
- **SelectiveRepeatSender**
- **SelectiveRepeatReceiver**
- **StopNWaitSender**
- **StopNWaitReceiver**
    - Donde si el rol del cliente/servidor es mandar datos entonces será un Sender y luego según la arquitectura sera **StopNWaitSender** o **SelectiveRepeatSender**
    - Si el rol del cliente/servidor es recibir los datos entonces será un Receiver y luego según la arquitectura será **Stop Wait Receiver** o **SelectiveRepeatReceiver**

Cliente: Esta clase es nuestra representación de un cliente que se conecta al servidor. Cada cliente distinto implica una nueva instancia de esta clase, cuyas responsabilidades son:

- Realizar el handshake con el servidor, indicando entre otras cosas el comando a realizar
- Escuchar mensajes del servidor mediante un socket
- Mandar a ejecutar el comando indicado en el handshake

Multithreading

Para que el servidor soporte múltiples clientes concurrentemente, es decir que puedan subir o descargar archivos al mismo tiempo, se implementó una solución con multithreading para dicho fin.
El servidor tiene un diccionario de clientes, que tiene como clave a la dirección IP del cliente y como valor a una tupla con (thread, queue). En ese thread se va a ejecutar la conexión de ese cliente, y se va a utilizar la queue para que el servidor se pueda comunicar con los threads de cada cliente.
En el servidor se ejecuta un loop para recibir constantemente mensajes por el socket con socket.recvfrom(). Esta operación devuelve una tupla con (mensaje, dirección). Entonces, cuando llega un mensaje utilizamos la dirección IP en el diccionario para encontrar el thread y la queue correspondientes a ese cliente. Se utiliza la queue para enviarle al thread del cliente el mensaje que llegó por el socket. Así, cada thread de un cliente recibe por su queue los mensajes enviados por ese cliente, para que se administren por separado en ese thread. De esa forma, mantenemos conexiones separadas para cada cliente, gestionando el envío y recepción de mensajes de forma independiente.

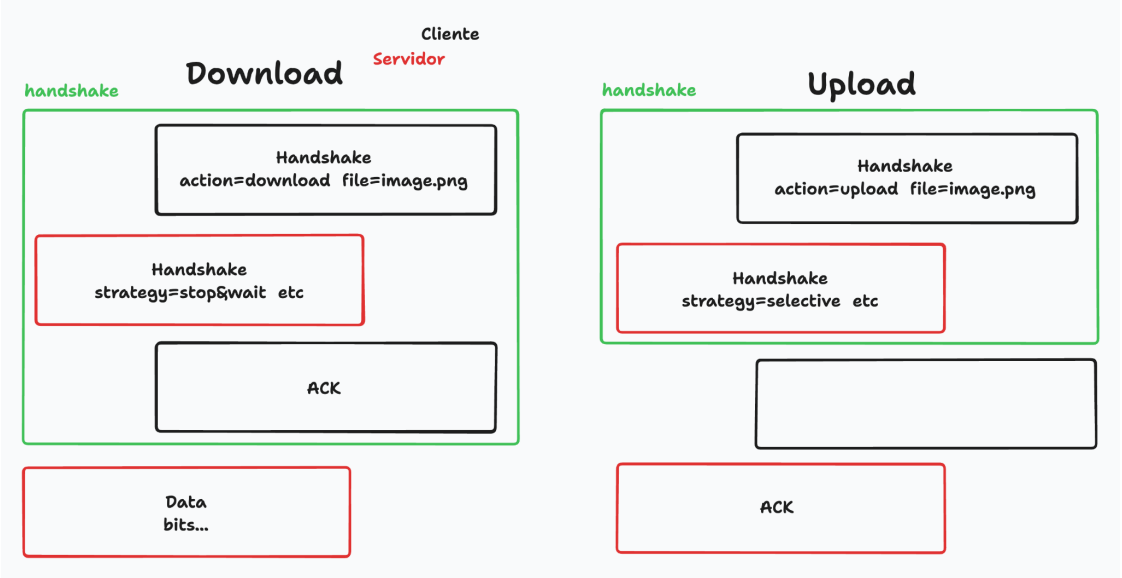


Funcionamiento de RDT

Nuestro sistema de transferencia de datos confiable funciona de la siguiente manera:

Handshake

Para el handshake utilizamos un mensaje “handshake” en común tanto para el cliente como para el servidor. Siempre el que comienza el proceso es el cliente, mandando un handshake informando el comando que desea ejecutar, nombre del archivo y filesize de este si el es el sender. Este mensaje lo envía en un loop hasta recibir el handshake del servidor y puede confirmar que este lo recibió y procesó. El servidor por su parte, cuando recibe el handshake actualiza sus valores con los parámetros recibidos y completa los que estaban incompletos (arquitectura a utilizar o filesize si este es el sender). El servidor también entra en un loop enviando su handshake hasta recibir un mensaje del cliente que sea tipo “data”, donde entiende que el cliente ya recibió el handshake y comenzó con la subida de un archivo; o cuando recibe un “ack” con número de secuencia 2^16-1, este “ack” es enviado por el cliente cuando este es el que va a recibir y lo usa para justamente informarle al servidor que recibió el handshake de su parte y está listo para empezar a recibir los datos.



Envío y recepción

En stop & wait cada paquete tiene su propio número de secuencia y el transmisor los envía de a uno, cuando el receptor recibe el paquete, este manda un ACK con el número de secuencia del último paquete correcto recibido. Si un paquete (ya sea un paquete de datos o un ACK de ese paquete) se llegase a perder, al ser stop & wait el transmisor no enviará nada más hasta recibir un ACK o que se acabe el timeout, en cuyo caso se envía el mismo paquete. Esto se realiza hasta que llegue el ACK correcto para el paquete enviado. Cuando el transmisor recibe el ACK para el paquete que acaba de enviar sabe que puede continuar y manda el siguiente paquete. Si recibe otro ACK vuelve a enviar el mismo paquete ya que esto implicaría que el último se perdió.

En selective repeat trabajamos con una window de tamaño fijo *m* que es igual tanto para el transmisor como para el receptor. Al igual que en stop & wait cada paquete tendrá su propio número de secuencia.
Mientras aún queden paquetes por enviar, el transmisor envia *m* paquetes donde *m* es el tamaño de la window y los marca como enviados pero no acknowledgeados y se les setea un timer para el timeout. Luego a medida que el receptor los recibe, este manda un ACK con el número acorde al paquete que recibió. La window solo se mueve en el caso de que el primer paquete de la window actual haya sido acknowledgeado.
Si se pierde algún paquete, cuando el timer de ese paquete llegue a un determinado valor este marcará al mismo como no enviado, de esta forma el transmisor volverá a transmitirlo hasta que este sea acknowlegeado y cuando esto ocurra, el paquete se marcará como acknowlegeado y cuando se intente actualizar la window se anulara el timer del timeout y se moverá la window.

Fin

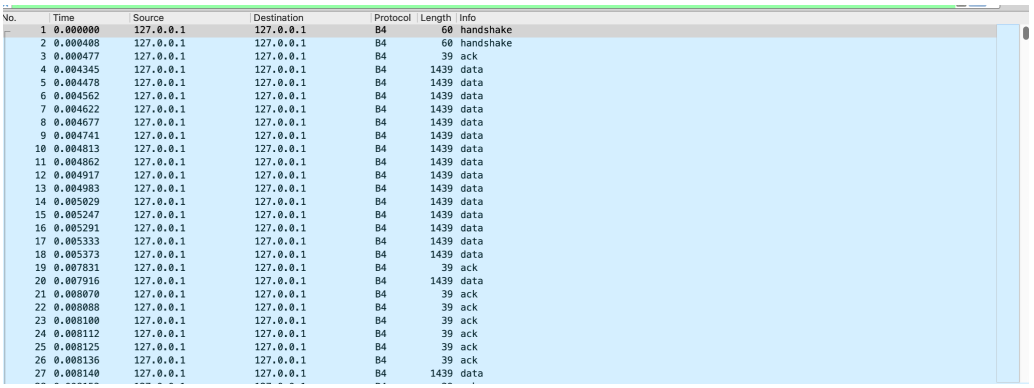
Para el fin, cuando una estrategia termina de enviar y/o recibir todos los mensajes, entra en el proceso de finalización. El sender comienza enviando un mensaje de tipo “fin”, y si le llega algún “ack” lo ignora y vuelve a enviar “fin” en loop hasta recibir una respuesta también de tipo “fin”. El receiver una vez termina de recibir lo que hace es comenzar a escuchar los mensajes recibidos esperando el “fin” para reenviarlo y terminar, pero también podría ocurrir que reciba un “data” si es que el “ack” se perdió y en este caso envía el “ack” correspondiente y sigue en loop esperando el mensaje “fin”.
Ambos tienen la posibilidad de haber perdido el paquete de fin del otro y que este se haya desconectado haciendo que quede esperando un mensaje de este tipo que nunca va a llegar, por lo que el fin tiene un tope de timeouts seteado en 10 para asumir el fin de forma implícita.

WIRESHARK

Para poder analizar mejor el protocolo, desarrollamos un plugin para wireshark en LUA que agrega soporte a nuestro protocolo.

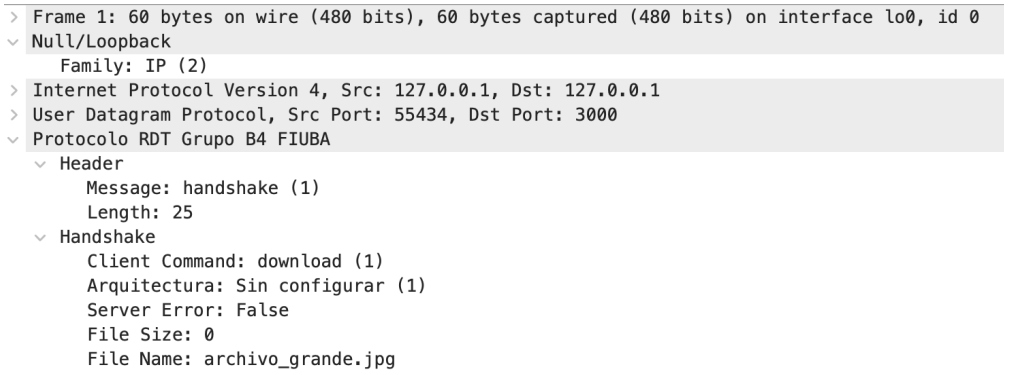
Instalación

Para instalar el protocolo es necesario buscar en la sección de “Ayuda” > “Acerca de Wireshark” > “Carpetas” > “Plugins en Lua personales” el directorio donde debe guardarse el archivo “b4.lua” encontrado en la raíz del proyecto. Una vez copiado, reiniciar el wireshark y automáticamente ya reconocerá los paquetes de nuestro protocolo con la abreviatura “B4”



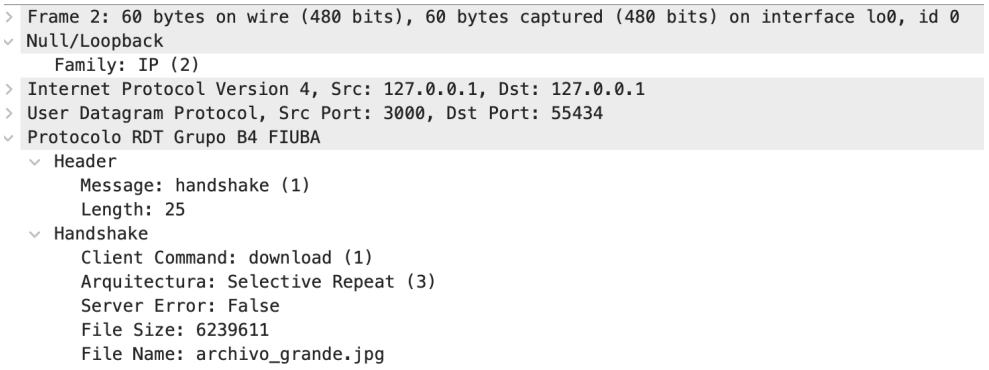
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	B4	60	handshake
2	0.000408	127.0.0.1	127.0.0.1	B4	60	handshake
3	0.000477	127.0.0.1	127.0.0.1	B4	39	ack
4	0.004345	127.0.0.1	127.0.0.1	B4	1439	data
5	0.004478	127.0.0.1	127.0.0.1	B4	1439	data
6	0.004562	127.0.0.1	127.0.0.1	B4	1439	data
7	0.004622	127.0.0.1	127.0.0.1	B4	1439	data
8	0.004677	127.0.0.1	127.0.0.1	B4	1439	data
9	0.004741	127.0.0.1	127.0.0.1	B4	1439	data
10	0.004813	127.0.0.1	127.0.0.1	B4	1439	data
11	0.004862	127.0.0.1	127.0.0.1	B4	1439	data
12	0.004917	127.0.0.1	127.0.0.1	B4	1439	data
13	0.004983	127.0.0.1	127.0.0.1	B4	1439	data
14	0.005029	127.0.0.1	127.0.0.1	B4	1439	data
15	0.005247	127.0.0.1	127.0.0.1	B4	1439	data
16	0.005291	127.0.0.1	127.0.0.1	B4	1439	data
17	0.005333	127.0.0.1	127.0.0.1	B4	1439	data
18	0.005373	127.0.0.1	127.0.0.1	B4	1439	data
19	0.007831	127.0.0.1	127.0.0.1	B4	39	ack
20	0.007916	127.0.0.1	127.0.0.1	B4	1439	data
21	0.008070	127.0.0.1	127.0.0.1	B4	39	ack
22	0.008088	127.0.0.1	127.0.0.1	B4	39	ack
23	0.008100	127.0.0.1	127.0.0.1	B4	39	ack
24	0.008112	127.0.0.1	127.0.0.1	B4	39	ack
25	0.008125	127.0.0.1	127.0.0.1	B4	39	ack
26	0.008136	127.0.0.1	127.0.0.1	B4	39	ack
27	0.008140	127.0.0.1	127.0.0.1	B4	1439	data

Vista general de una descarga vía nuestro protocolo

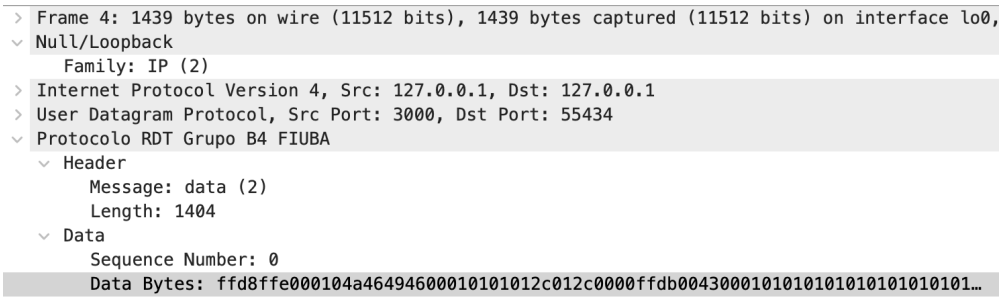


```
> Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface lo0, id 0
v Null/Loopback
    Family: IP (2)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 55434, Dst Port: 3000
v Protocolo RDT Grupo B4 FIUBA
  v Header
      Message: handshake (1)
      Length: 25
  v Handshake
      Client Command: download (1)
      Arquitectura: Sin configurar (1)
      Server Error: False
      File Size: 0
      File Name: archivo_grande.jpg
```

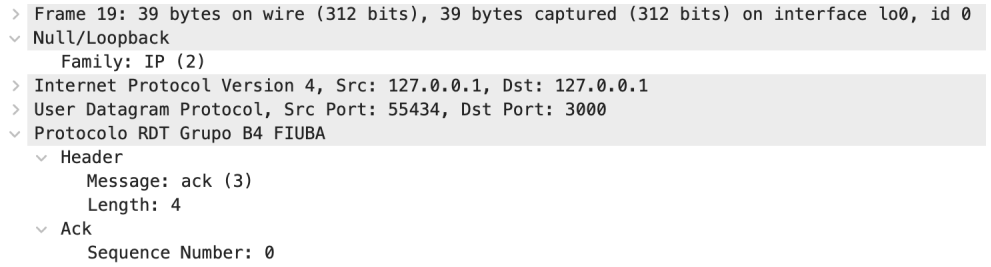
Handshake desde el cliente con algunos valores sin completar



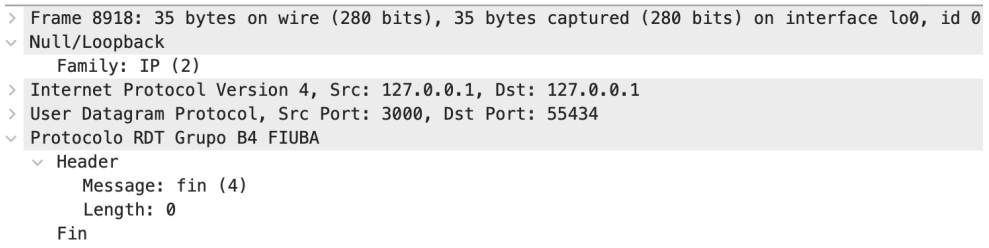
Handshake desde el servidor con los datos completados



Data con seq number 0 y 1400 bytes de datos



Ack con seq number 0



Fin

PRUEBAS

```
python3 start-server -H 127.0.0.1 -s files -a sr -v

El servidor esta listo para recibir
[127.0.0.1:62171] === iniciando el handshake ===
[127.0.0.1:62171]   <<  HANDSHAKE
[127.0.0.1:62171]   >>  HANDSHAKE
[127.0.0.1:62171]   <<  ACK
[127.0.0.1:62171]
[127.0.0.1:62171]   client:         127.0.0.1, 62171
[127.0.0.1:62171]   command:        download
[127.0.0.1:62171]   architecture:   selectiveRepeat
[127.0.0.1:62171]   filename:       prueba.jpeg
[127.0.0.1:62171]   filesize:       5595
[127.0.0.1:62171] === handshake finalizado ===

[127.0.0.1:62171] >>  DATA 0
[127.0.0.1:62171] >>  DATA 1
[127.0.0.1:62171] >>  DATA 2
[127.0.0.1:62171] >>  DATA 3
[127.0.0.1:62171] <<  ACK 0
[127.0.0.1:62171] <<  ACK 1
[127.0.0.1:62171] <<  ACK 2
[127.0.0.1:62171] <<  ACK 3
[127.0.0.1:62171] >>  FIN
[127.0.0.1:62171] <<  FIN
[127.0.0.1:62171] Selective Repeat Terminado
[127.0.0.1:62171] DESCARGA files/prueba.jpeg en 0.610ms (8955.498KB/s)
^C
Servidor desconectado con éxito.
```

```
❯
python3 download -H 127.0.0.1 -n prueba.jpeg -d archivo_pr
ueba.jpg -v

=== iniciando el handshake ===
  >>  HANDSHAKE
  <<  HANDSHAKE
  >>  ACK

  command:        download
  architecture:   selectiveRepeat
  filename:       prueba.jpeg
  filesize:       5595
=== handshake finalizado ===

<<  DATA 0
>>  ACK 0
<<  DATA 1
>>  ACK 1
<<  DATA 2
>>  ACK 2
<<  DATA 3
>>  ACK 3
<<  FIN
>>  FIN
Selective Repeat Terminado

DESCARGA ./archivo_prueba.jpg en 0.437ms (12502.520KB/s)
```

Envío de archivo pequeño sin pérdidas

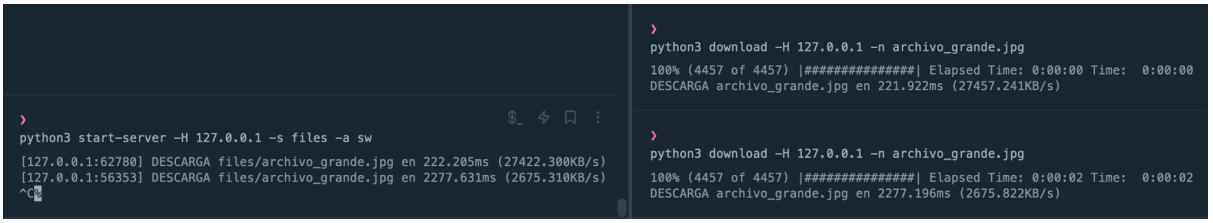
```
❯
python3 start-server -H 127.0.0.1 -s files -a sr -v

El servidor esta listo para recibir
[127.0.0.1:53928] === iniciando el handshake ===
[127.0.0.1:53928]   <<  HANDSHAKE
[127.0.0.1:53928]   >>  HANDSHAKE
[127.0.0.1:53928]   <<  ACK
[127.0.0.1:53928]
[127.0.0.1:53928]   client:         127.0.0.1, 53928
[127.0.0.1:53928]   command:        download
[127.0.0.1:53928]   architecture:   selectiveRepeat
[127.0.0.1:53928]   filename:       prueba.jpeg
[127.0.0.1:53928]   filesize:       5595
[127.0.0.1:53928] === handshake finalizado ===

[127.0.0.1:53928] >>  DATA 0
[127.0.0.1:53928] >>  DATA 1
[127.0.0.1:53928] >>  DATA 2
[127.0.0.1:53928] >>  DATA 3
[127.0.0.1:53928] <<  ACK 0
[127.0.0.1:53928] >>  DATA 3
[127.0.0.1:53928] <<  ACK 3
[127.0.0.1:53928] >>  DATA 1
[127.0.0.1:53928] >>  DATA 2
[127.0.0.1:53928] >>  DATA 1
[127.0.0.1:53928] >>  DATA 2
[127.0.0.1:53928] <<  ACK 1
[127.0.0.1:53928] <<  ACK 2
[127.0.0.1:53928] >>  FIN
[127.0.0.1:53928] <<  FIN
[127.0.0.1:53928] Selective Repeat Terminado
[127.0.0.1:53928] DESCARGA files/prueba.jpeg en 208.416ms (26.216KB/s)
^C
Servidor desconectado con éxito.
```

```
❯
python3 download -H 127.0.0.1 -n prueba.jpeg -d archivo_pr
ueba.jpg -v

=== iniciando el handshake ===
  >>  HANDSHAKE
  <<  HANDSHAKE
  >>  ACK

  command:        download
  architecture:   selectiveRepeat
  filename:       prueba.jpeg
  filesize:       5595
=== handshake finalizado ===

<<  DATA 0
>>  ACK 0
<<  DATA 2
>>  ACK 2
<<  DATA 3
>>  ACK 3
<<  DATA 3
>>  ACK 3
<<  DATA 1
>>  ACK 1
<<  DATA 1
>>  ACK 1
<<  DATA 2
>>  ACK 2
<<  FIN
>>  FIN
Selective Repeat Terminado

DESCARGA ./archivo_prueba.jpg en 208.184ms (26.245KB/s)
```

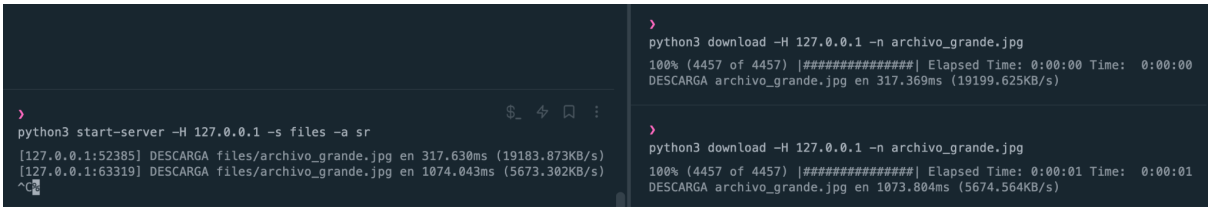
Envío de archivo pequeño con pérdidas al 30%

Comparativa entre Stop&Wait y Selective Repeat

Para hacer la comparación, vamos a primero hacer una descarga de un archivo binario de 5 MB sin pérdida y luego la misma operación pero con pérdidas del 10%



Stop & Wait



Selective Repeat

Como se puede ver, sin pérdidas es más rápido el Stop & Wait por su simplicidad y no estar continuamente generando y cancelando timers. Pero a la hora de perder paquetes, aunque sean tan solo 10%, Selective Repeat es el doble de rápido por hacer el envío a ráfagas haciendo que la window siga avanzando a pesar de que se puedan perder paquetes.

PREGUNTAS A RESPONDER

1. Describa la arquitectura Cliente-Servidor.

La arquitectura cliente-servidor es un modelo de diseño de software en el que las tareas se reparten entre el host proveedor de recursos o servicios, llamado servidor, y el demandante, llamado cliente. Un cliente inicia la comunicación y realiza peticiones al servidor (de dirección fija y conocida), quien le da respuesta. Los clientes no se comunican entre sí, sino que lo hacen mediante el servidor.

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

La función de un protocolo de capa de aplicación es establecer un método de comunicación común entre dos procesos (aplicaciones) que corren en distintos hosts. Este puede ser, por ejemplo, el protocolo SMTP para correo electrónico o el protocolo HTTP para la web. Para eso, se definen reglas y formatos para el intercambio de datos, por ejemplo:

- Tipo de mensaje: petición o respuesta
- Sintaxis de los mensajes (campos obligatorios, opcionales)
- Valores posibles para esos campos

En resumen, lo que se busca es proporcionar una interfaz de comunicación para las aplicaciones.

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

La descripción del protocolo de aplicación desarrollado en este trabajo se puede leer 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?

Protocolo TCP

TCP ofrece un servicio orientado a la conexión confiable. Garantiza la entrega ordenada y sin errores de datos entre dos dispositivos en una red. Ofrece control de flujo, retransmisión de datos perdidos y control de congestión.

Características

- Establece una conexión confiable antes de enviar datos y la cierra después de la transmisión.
- Es un protocolo fiable. Garantiza que los paquetes lleguen en el orden correcto y sin errores.
- Provee control de flujo: Evita que el emisor sature al receptor con datos enviados demasiado rápido. El emisor regula la transmisión para que el receptor no reciba todo junto y así no pueda procesarlo
- Provee control de congestión: Regula la transmisión para controlar la congestión en los enlaces

Es apropiado usar TCP cuando:

- Aplicaciones que requieren entrega confiable de datos, como transferencia de archivos, correo electrónico, navegación web y transacciones en línea.
- Situaciones en las que la integridad de los datos y el orden de entrega sean importantes

Protocolo UDP

UDP es un protocolo sin conexión y no garantiza la entrega confiable de datos. Ofrece un servicio de envío de datagramas independientes que pueden llegar en cualquier orden o incluso perderse.

Características

- Sin conexión: No establece una conexión antes de enviar paquetes y no mantiene un estado de conexión.
- No garantiza la entrega: Los paquetes pueden perderse o llegar en un orden diferente al que se enviaron.
- Mayor velocidad: Debido a su simplicidad y falta de verificación de errores, UDP es más rápido que TCP.

Es apropiado usar UDP cuando:

- Aplicaciones en tiempo real que pueden tolerar pérdida ocasional de datos, como videoconferencias, transmisión de audio en tiempo real y videojuegos en línea.
- Situaciones en las que la latencia es crítica y la velocidad de transmisión es prioritaria.

- Aplicaciones que implementan su propio control de errores y recuperación de datos.

Conclusión: TCP es un protocolo más completo, pero no podríamos definir cual es mejor o peor ya que dependerá de la aplicación en donde se quiera usar y es algo que lo definirá el desarrollador.

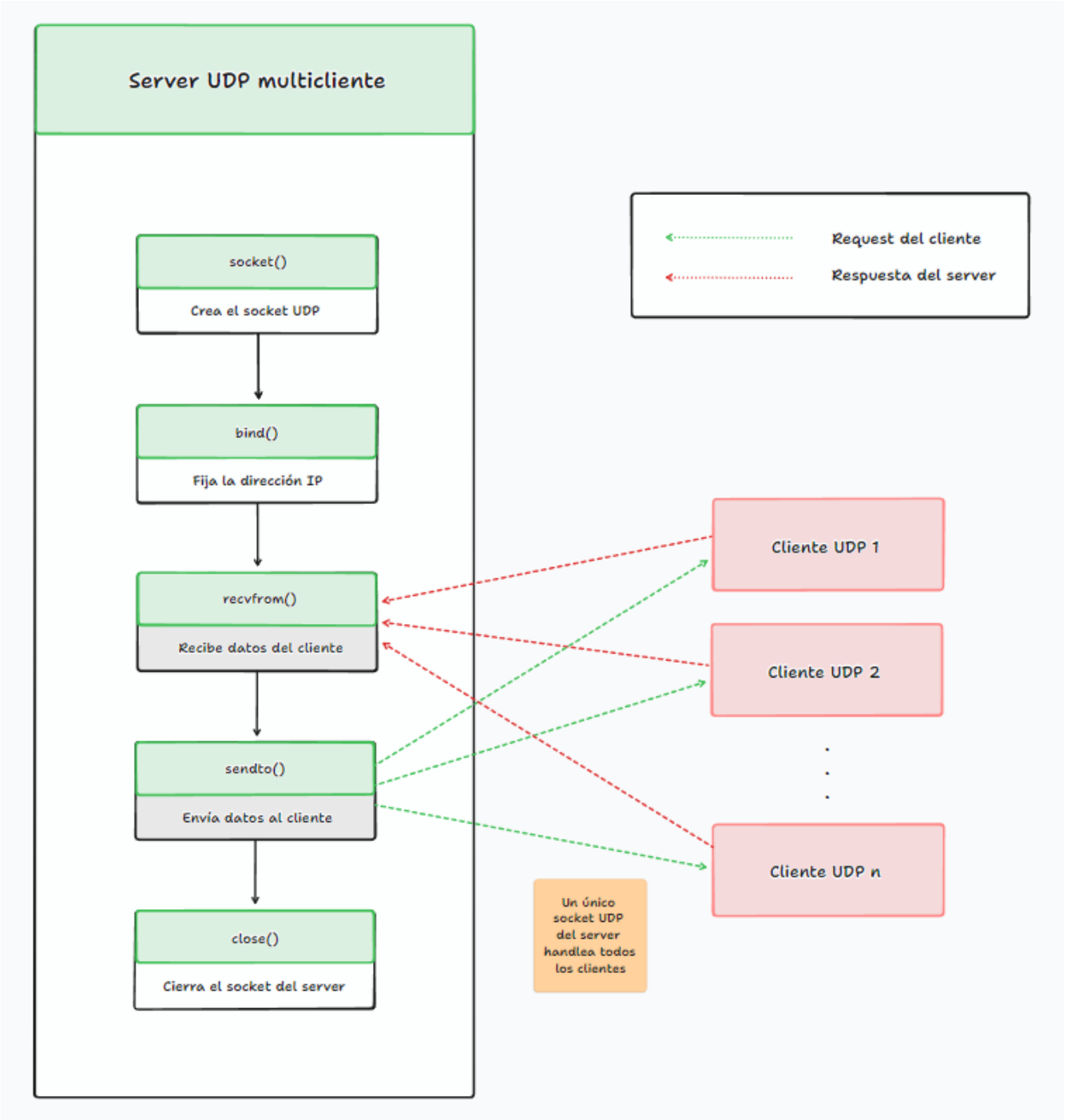
DIFICULTADES ENCONTRADAS

Un problema que tuvimos que enfrentar fue cómo implementar una solución para poder atender a múltiples clientes al mismo tiempo.

El objetivo inicial era crear un hilo para cada cliente, y así poder manejar las conexiones por separado. El problema es que estamos utilizando el protocolo UDP, por lo que el servidor tiene un único socket, y no se manejan conexiones individuales con cada cliente. Por esto, tenemos que manejar las conexiones con cada cliente de forma manual.

Si estuviésemos utilizando el protocolo TCP, la solución sería bastante simple ya que sólo habría que crear un thread por cada cliente y pasarle el socket correspondiente a la conexión del servidor con ese cliente. No pudimos realizar esto ya que el protocolo UDP nos limita a que el servidor tenga un único socket.

Para superar esta limitación, utilizamos la dirección IP de los clientes para filtrar los mensajes recibidos en el socket. Así, podemos distinguir de qué cliente es el mensaje que llega por el socket, y logramos manejar conexiones independientes con cada uno usando threads que reciben los mensajes del cliente correspondiente.


Server UDP multicliente
socket()
Crea el socket UDP
bind()
Fija la dirección IP
recvfrom()
Recibe datos del cliente
sendto()
Envía datos al cliente
close()
Cierra el socket del server
Request del cliente
Respuesta del server
Cliente UDP 1
Cliente UDP 2
Cliente UDP n
Un único socket UDP del server handlea todos los clientes

CONCLUSIÓN

El presente trabajo práctico nos permitió adquirir conocimientos sobre cómo funcionan los sockets UDP, además de poder implementar un modelo cliente-servidor utilizando Python. Asimismo, la implementación de un protocolo RDT nos dió la posibilidad de estudiar otros temas como la conformación de un paquete, el manejo de paquetes perdidos y el reenvío de los mismos, etcétera.
Este trabajo práctico permitió aplicar los temas vistos en clase, en un escenario real.

Por otro lado, pudimos observar las diferencias entre los protocolos Stop & Wait y Selective Repeat, principalmente en el rendimiento:

En primer lugar, notamos que en un entorno ideal, sin pérdida de paquetes, Stop & Wait es más rápido que Selective Repeat en alrededor de un 50%. Stop & Wait tardó 200 ms en enviar un archivo de aproximadamente 5 MB, mientras que Selective Repeat tardó 300 ms en enviar el mismo archivo.
Cuando enviamos el mismo archivo pero con una pérdida de paquetes del 10% mediante Comcast, observamos que Selective Repeat fue un 100% más rápido que Stop & Wait.
Creemos que Stop & Wait es más rápido que Selective Repeat cuando no hay pérdida de paquetes ya que es un protocolo más liviano, con menor overhead, por lo que si el entorno es ideal, su operación es más veloz. En cambio, si introducimos pérdida de paquetes, Selective Repeat tiene mucho mejor rendimiento ya que es un protocolo mucho mejor preparado para estos casos en los que el entorno no es ideal. Al enviar paquetes “de a ráfagas” mediante la window, se pierde mucho menos tiempo en el caso de que haya un paquete que no llegó a destino, ya que mientras tanto se siguen enviando otros paquetes. Por el contrario, Stop & Wait frena completamente el envío de paquetes si uno no llegó a destino (no se responde con ACK) hasta que se el mismos se reenvíe con éxito, por lo que es mucho más lento en estos casos.