Introducción

Este trabajo práctico tiene como objetivo crear una aplicación de red para transferir archivos entre cliente y servidor. Se implementarán dos operaciones fundamentales: UPLOAD (enviar archivos del cliente al servidor) y DOWNLOAD (descargar archivos del servidor al cliente). Se tendrán en cuenta los protocolos TCP y UDP para la comunicación. TCP ofrece un servicio confiable orientado a la conexión, mientras que UDP es sin conexión y menos confiable. Se implementarán versiones de UDP con protocolo Stop & Wait y Selective Repeat con el objetivo de lograr una transferencia confiable al utilizar el protocolo.

Hipótesis y suposiciones realizadas

- No tendremos paquetes corruptos, UDP lo valida previamente.
- El archivo a descargar siempre se encuentra presente en el servidor.
- Si un archivo a subir ya se encuentra en el servidor, se reemplaza su contenido.
- El tamaño maximo de mensaje sera de 1GB.

Implementación

Cliente

La funcionalidad del cliente se divide en dos aplicaciones de línea de comandos: **upload** y **download**.

Upload:

El comando `upload` envía un archivo al servidor para ser guardado con el nombre asignado.

`python3 upload.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-s FILEPATH] [-n FILENAME]` **faltaría ver como pasarle el protocolo que quiero.**

Donde cada flag indica:

- `-h, --help`: Imprime el mensaje de "help"
- `-v, --verbose`: Incrementa en uno la verbosidad en cuanto al sistema de logueo del servidor.
- `-q, --quiet`: Decrementa en uno la verbosidad en cuanto al sistema de logueo.
- `-H, --host`: Indica la dirección IP del servicio.
- `p, --port`: Indica el puerto.
- `-s, --src`: Indica el path del archivo a subir.
- `-n, --name`: Nombre del archivo a subir.

En la implementación, esta operación sigue los siguientes pasos:

- Primero, se establece una conexión utilizando un Socket en el host "localhost" y el puerto proporcionado como parámetro. Luego, se envía una estructura de

Metadata al servidor **(Describirlo en el detalle del protocolo implementado)**. Una vez que se ha enviado la Metadata, comienza la transferencia real del archivo. Para garantizar una transferencia eficiente, el archivo se divide en segmentos más pequeños, cada uno con un tamaño máximo definido (por ejemplo, `MAX_MESSAGE_SIZE = 1024`). Estos segmentos se envían uno por uno servidor hasta que se haya enviado todo el archivo. Es importante destacar que si el tamaño del archivo no es un múltiplo exacto del tamaño máximo del mensaje, el último segmento puede ser más pequeño. Una vez que se ha enviado todo el archivo, se espera una respuesta del servidor para confirmar si la transferencia se realizó correctamente. Esta respuesta puede contener información sobre si el archivo se recibió correctamente o si hubo algún problema durante la transferencia, ayudandonos a garantizar la entrega.

Download:

El comando `download` descarga un archivo especificado desde el servidor.

```
python3 download.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-d FILEPATH]
[-n FILENAME] [-P PROTOCOL]
```

Donde todos los flags indican lo mismo que con el comando anterior, con la diferencia de `-d, --dst` que indica el path de destino del archivo a descargar.

En nuestra implementación, sin importar el protocolo, esta operación sigue los siguientes pasos:

Al igual que `upload`, para comenzar la transferencia de archivos, primero necesitamos establecer un socket utilizando el protocolo adecuado. Luego, el cliente enviará la estructura Metadata al servidor. Una vez que el servidor recibe la Metadata, puede determinar el tamaño del archivo que se va a transferir el cual es enviado de vuelta al cliente para que este último conozca cuántos bytes esperar durante la descarga.

Esta información permite al cliente prepararse para recibir la descarga completa del archivo, asegurándose de manejar correctamente cada segmento para reconstruir el archivo original en su sistema local. Una vez que se ha completado la descarga de todos los segmentos, el cliente confirma la integridad de la transferencia.

Servidor

El servidor debe estar preparado para recibir un mensaje que indica que comienza una nueva conexión, es decir que tiene que estar ejecutándose como proceso antes de que el cliente trate de iniciar el contacto. Es por eso, que el primer comando a ejecutar es el `start-server`.

```
start-server.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-s DIRPATH]
```

Donde los flags indican:

- `-h/--help`: Imprime el mensaje de "help"
- `-v/--verbose`: Incrementa en uno la verbosidad en cuanto al sistema de logueo del servidor
- `-q/--quiet`: Decrementa en uno la verbosidad en cuanto al sistema de logueo.
- `-H/--host`: Indica la dirección IP del servicio
- `-p/--port`: Indica el puerto
- `-s/--storage`: El path en el que se almacenan los archivos.

El servidor va a proveer el servicio de subida y bajada de archivos. Para ello seguirá los siguientes pasos:

Cuando el servidor recibe el comando "start-server", crea un nuevo servidor y comienza a escuchar en el puerto especificado para nuevas conexiones entrantes. Una vez que el servidor está creado y escuchando, permanece a la espera de una conexion. Cuando se establece una nueva conexión, el servidor acepta la conexión y crea un nuevo hilo para manejarla. Esto permite al servidor seguir esperando nuevas conexiones mientras maneja la conexión actual en un hilo separado. Este enfoque de subprocesos múltiples asegura que el servidor pueda manejar múltiples conexiones simultáneamente. Una vez que se ha establecido la conexión y se ha creado un nuevo hilo para manejarla, el servidor espera recibir la Metadata correspondiente a la operación de "UPLOAD" o "DOWNLOAD".

Para **UPLOAD**, hay que encargarse de recibir un archivo:

- Recibirá el archivo de a segmentos de tamaño `MAX_MESSAGE_SIZE = 1024`, por lo tanto, teniendo en cuenta que ya conoce su tamaño, va a iterar hasta saber que consiguió el archivo completo. Una vez que finalizó, se encarga de mandar un mensaje al cliente de que finalizó correctamente el comando.

Para **DOWNLOAD**, hay que encargarse de envíar un archivo:

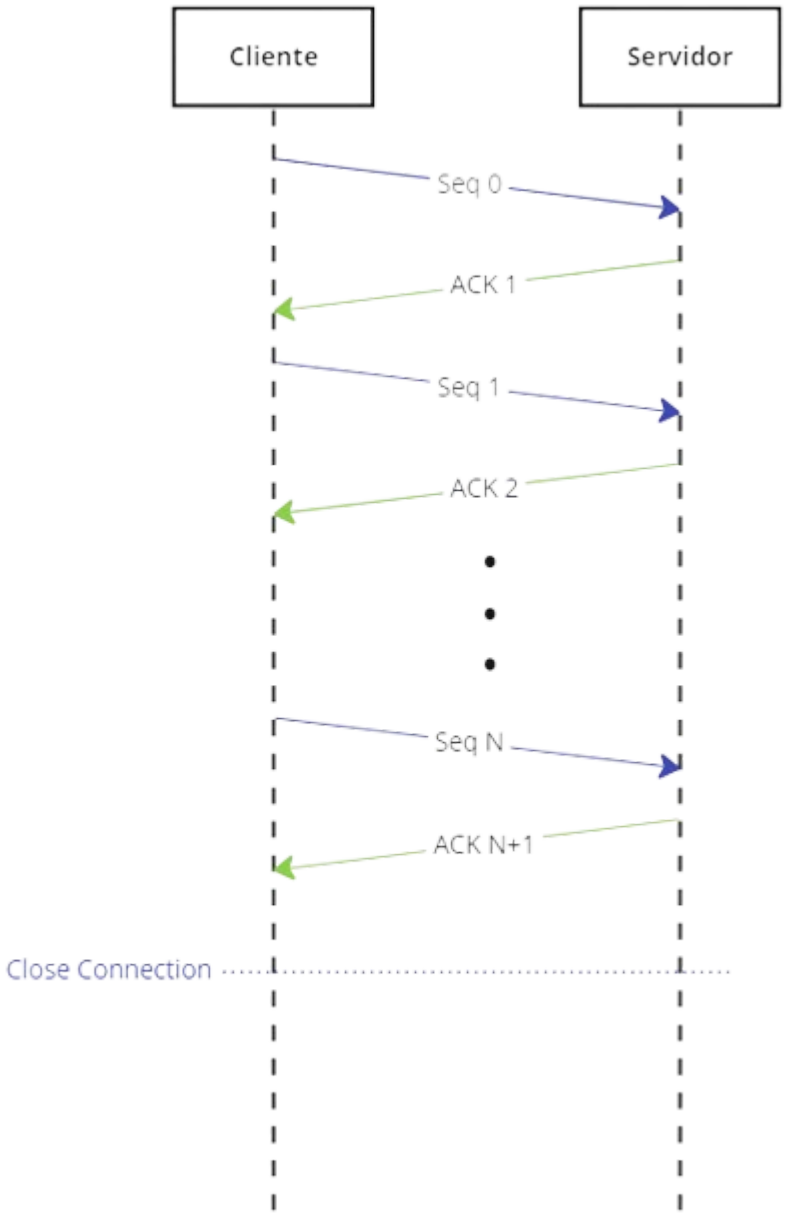
- Una vez que el servidor ubica el archivo solicitado, le envía al cliente el tamaño del archivo. Luego, le envía el archivo al cliente de a segmentos de a `MAX_MESSAGE_SIZE = 1024`. Y al finalizar, espera que el cliente le mande un mensaje indicando que recibió el archivo correctamente.

Handshake

A continuación detallaremos el funcionamiento del handshake para cada uno de los protocolos

Stop and Wait

1. **Envío del ACK (Acknowledgement) por el Cliente:**
    - En el método `receive(self, is_server)`, el cliente (o el servidor si `is_server` es `True`) envía un mensaje de tipo ACK (`ACK_TYPE`) al otro extremo del canal de comunicación, indicando que está listo para recibir datos.
    - Este mensaje incluye el número de secuencia actual (`self.seq_num`) para la sincronización.
2. **Transferencia de Datos:**
    - Después de enviar el ACK, el cliente (o el servidor) espera recibir datos del otro extremo.
    - El cliente espera recibir un paquete de datos del servidor en el método `receive`. Si se recibe un paquete dentro del tiempo de espera (`TIMEOUT`), se envía un ACK de confirmación de recepción.
    - El servidor, por otro lado, espera recibir un ACK del cliente en el método `send`. Si se recibe un ACK válido, se procede a enviar el siguiente paquete de datos. Si no, se continúa retransmitiendo el paquete hasta que se reciba el ACK o se alcance el número máximo de intentos (`MAX_TRIES`).
3. **Acknowledgement de finalización de transferencia:**
    - El cliente y el servidor continúan este intercambio de datos y ACK hasta que se completa la transferencia del archivo o hasta que ocurra un error.



Si hay pérdida de paquetes durante la transferencia de datos, el protocolo Stop-and-Wait manejaría esta situación de la siguiente manera:

1. **Pérdida de ACK (Acknowledgement):**

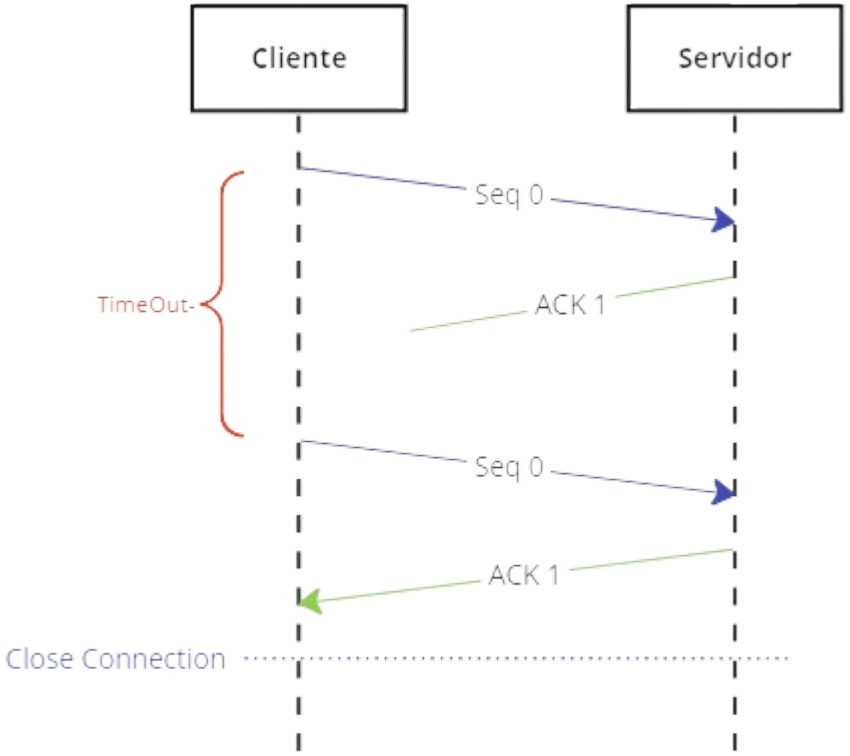
- Si el cliente envía un paquete de datos al servidor o al cliente y no recibe un ACK de confirmación dentro del tiempo de espera especificado (`TIMEOUT`), asume que el paquete se perdió en el camino y retransmite el mismo paquete.
- El cliente continuará retransmitiendo el paquete hasta que reciba un ACK válido o alcance el número máximo de intentos (`MAX_TRIES`).

2. **Manejo de Retransmisiones:**

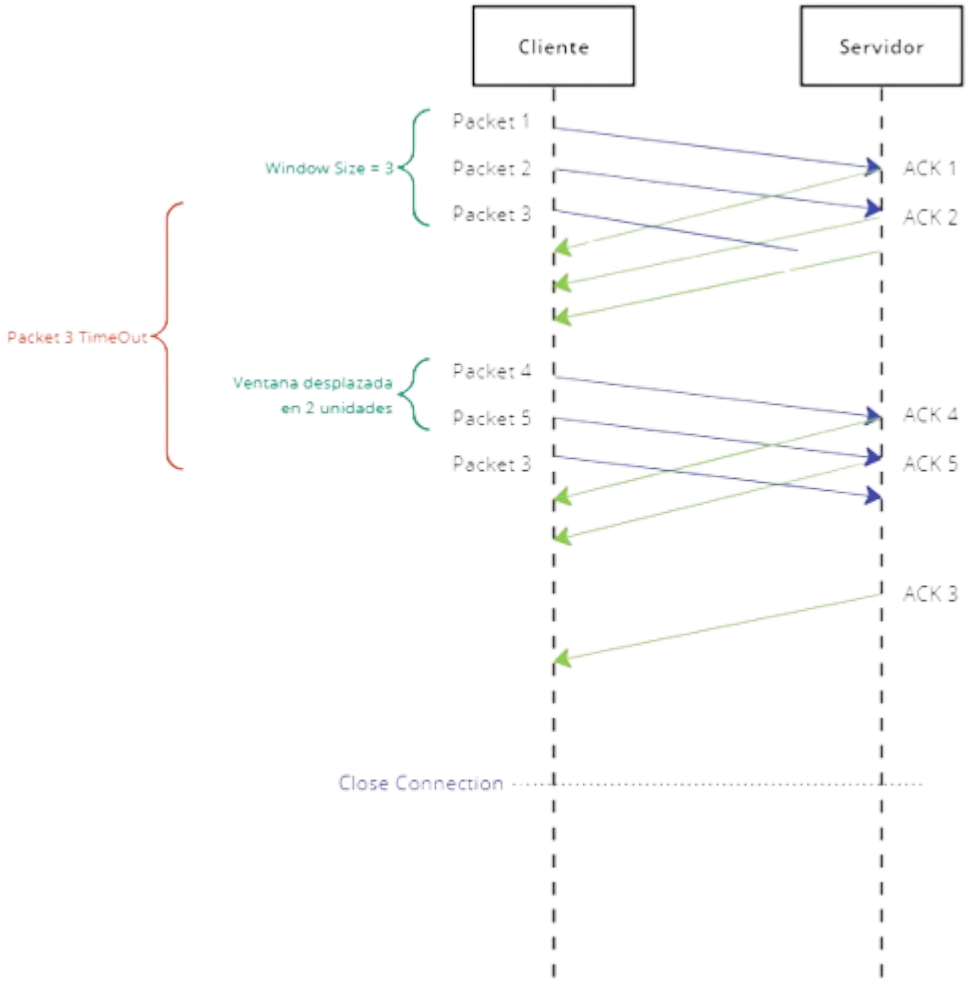
- Tanto el cliente como el servidor mantendrán un contador de intentos (`self.tries`) para rastrear el número de retransmisiones de paquetes.
- Si el número de intentos alcanza el límite máximo (`MAX_TRIES`), se considera que ha ocurrido un error en la comunicación y se finaliza la transferencia con un

mensaje de error.

Este protocolo maneja la pérdida de paquetes mediante la retransmisión de paquetes perdidos y el seguimiento de intentos para evitar la congestión de la red. Esto asegura que la transferencia de datos sea confiable incluso en entornos donde pueda haber pérdida de paquetes.

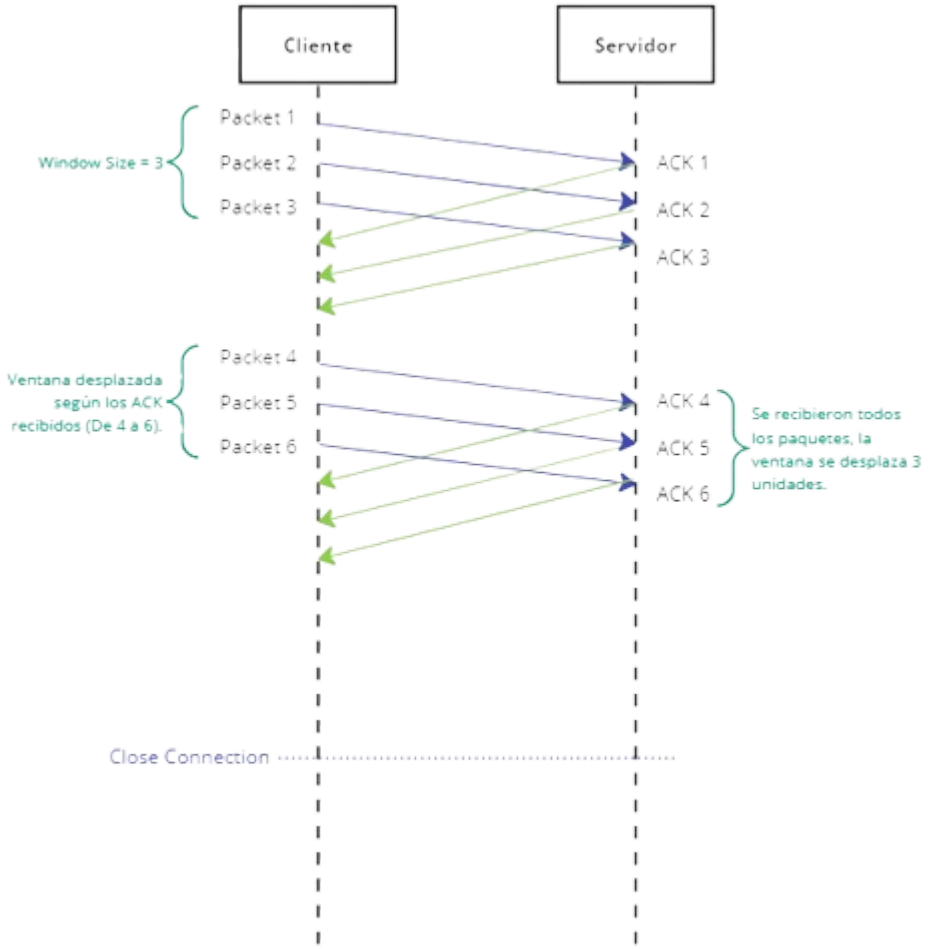


- **~~ESCRIBIR PARTE DE PÉRDIDA DE PAQUETE~~ **

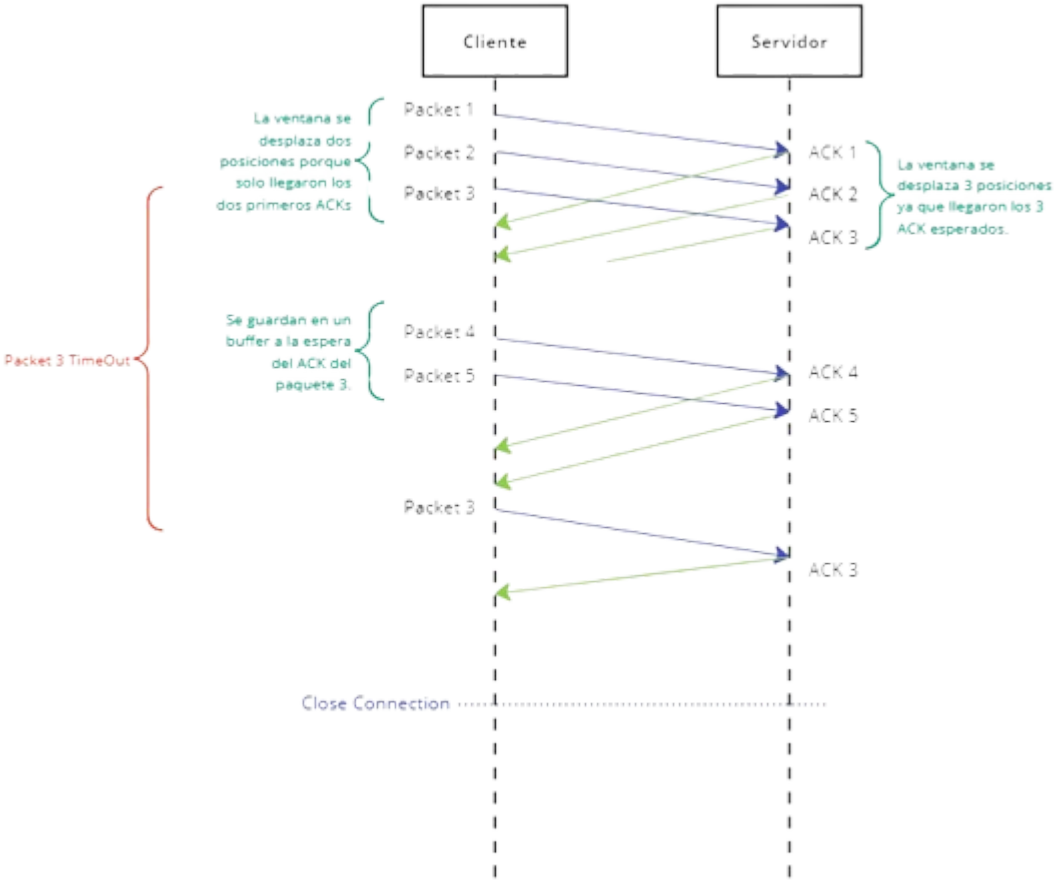


Selective Repeat

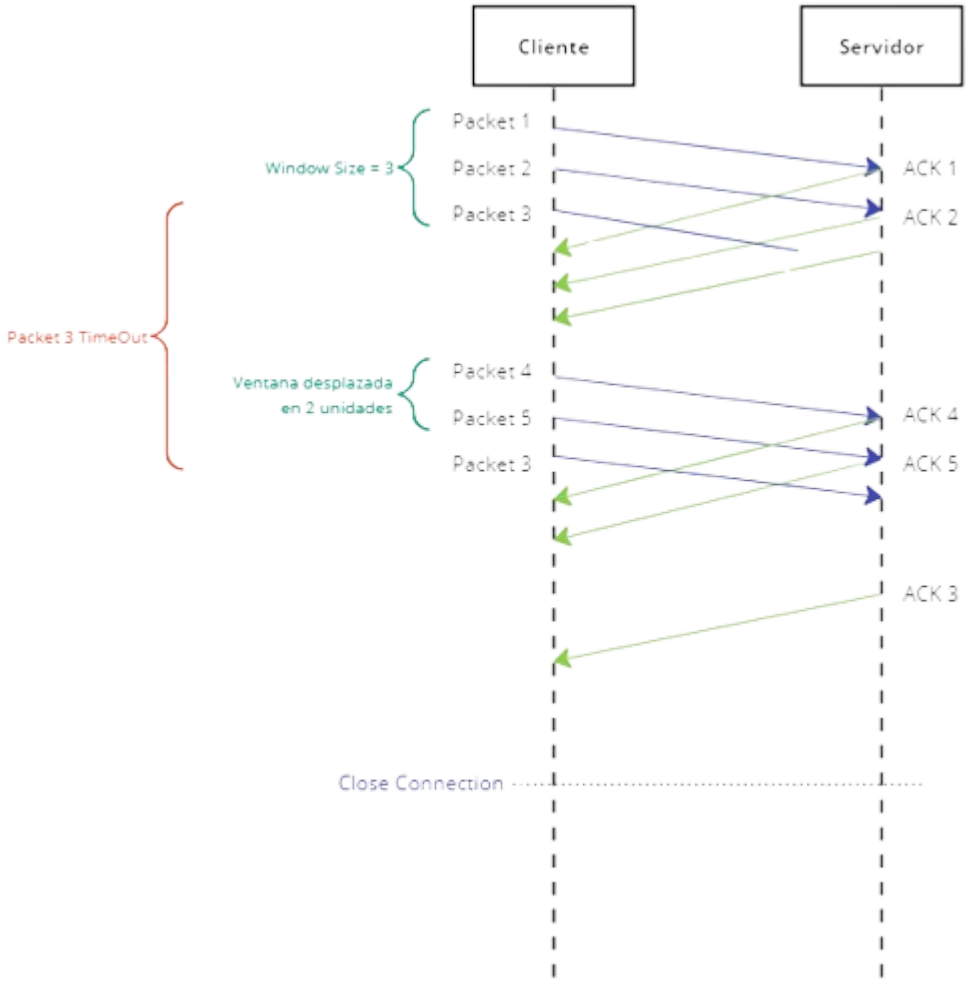
```
~~En esta sección se explicará la implementación realizada para selective repeat.~~
```



- ~~**ESCRIBIR PARTE DE PÉRDIDA DE ACK**~~



- ~~**ESCRIBIR PARTE DE PÉRDIDA DE PAQUETE**~~



Cierre de conexión

```
~~En esta sección se explicará que se implementará para finalizar la conexión, NO
con un Ctrl + C.~~
```

Pruebas

- Para probar la pérdida de paquetes hicimos uso de Mininet, para esto creamos una topología de cuatro Host y un Router (`topology.py`), donde podemos probar pérdida de paquetes de ambos lados de la conexión, y la conexión de múltiples clientes a un mismo servidor. Para ello debemos ejecutar los siguientes comandos:

```
  sudo python3 topology.py    #Ejecuta mininet con la topologìa creada por
nosotros.

  link s1 h1 up   #Establece la conexión entre el switch y el host y la pone
```

```
en estado activo.

  link s1 h2 up   #Establece la conexión entre el switch y el cliente y la
pone en estado activo (Ejecutar para todos los clientes que quisieramos
probar).

  s1 tc qdisc add dev s1-eth2 root netem loss 10%    #Establece perdida de
paquetes del 10% para el switch s1 salida eth2.

  s1 tc qdisc del dev s1-eth2 root netem loss 10%    #Elimina la regla
establecida para la pèrdida de paquetes.

  xterm h1   #Abre una ventana de terminal gráfica para el host h1 donde
podemos iniciar el servidor.

  xterm h2   #Abre una ventana de terminal gráfica para el cliente donde
podemos iniciar, por ejemplo, el cliente 1.
```

- Para constatar que los mensajes de nuestro protocolo son como exactamente se describen en este informe, deberíamos capturar paquetes con Wireshark, por lo que creamos un plugin que parsea los bytes de los mensajes de la capa de aplicación a los campos de nuestro protocolo (Ver nombre de archivo). Para probarlo debemos seguir los sigueintes pasos:

    1. Dentro de Wireshark ir a Ayuda -> Acerca de Wireshark -> Carpetas
    2. Clickear en el item Complementos personales de Lua.
    3. Copiar y pegar el disector en esa carpeta y ya Wireshark lo utilizará en las proximas capturas a realizar.

Análisis

~~En esta sección se mostrará un análisis de mediciones utilizando Upload y Download para ambos protocolos con pérdida de paquetes y sin pérdida.~~

Preguntas a responder

> **Describa la arquitectura Cliente-Servidor.**

En la arquitectura cliente-servidor, un host permanece siempre activo como servidor para atender las solicitudes de otros hosts, denominados clientes. Los clientes no pueden comunicarse directamente entre sí. Para que un cliente se comunique con el servidor, este último posee una dirección fija y conocida llamada dirección IP. Sin

embargo, el servidor no tiene previamente conocimiento de las direcciones de los clientes.

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

Un protocolo de capa de aplicación establece cómo se comunican los procesos de aplicaciones que se ejecutan en diferentes sistemas finales. Esto implica definir: - Tipos de mensaje: Los mensajes pueden ser de solicitud o de respuesta. Las solicitudes son enviadas por el cliente al servidor para solicitar algún servicio o información, mientras que las respuestas son enviadas por el servidor al cliente en respuesta a una solicitud. - Campos de mensaje y su significado: Cada tipo de mensaje tiene campos específicos que contienen información relevante para la comunicación. El significado de cada campo se establece en la especificación del protocolo y puede variar según el contexto de la aplicación. - Reglas para enviar y responder mensajes: El protocolo define reglas para determinar cuándo y cómo un proceso envía y responde mensajes. Esto incluye aspectos como el establecimiento de conexiones, el formato de los mensajes, el manejo de errores y el cierre de la comunicación.

> **Detalle el protocolo de aplicación desarrollado en este trabajo._**

Se explicó en el item Implementación.

> **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?**

La capa de transporte en el stack TCP/IP tiene como objetivo principal proporcionar un servicio de entrega confiable de datos de la capa de red a la capa de aplicación. En este contexto, se encuentran dos protocolos principales: UDP (User Datagram Protocol) y TCP (Transmission Control Protocol), cada uno con sus características y servicios específicos.

TCP: Transmission Control Protocol

- Características
    - Proporciona una entrega de paquetes confiable sin pérdida ni duplicados.
    - Es un servicio orientado a la conexión para las aplicaciones que lo utilizan.
    - Tiene una estructura de encabezado más compleja que UDP.

- Servicios
    - EOfrece entrega de datos de proceso a proceso.
    - Implementa un protocolo de transferencia de datos confiable (RDT) que incluye chequeo de errores e integridad, garantía de entrega, y orden de entrega asegurado.
    - Realiza control de congestión para mejorar el rendimiento de la red.

UDP: User Datagram Protocol

- Características
    - Puede haber pérdida de paquetes y posibilidad de duplicados.
    - No requiere establecer una conexión previa.
    - Tiene una estructura de encabezado simple.
- Servicios:
    - Ofrece la entrega de datos de proceso a proceso.
    - Realiza un chequeo de errores e integridad utilizando un campo de detección de errores (checksum) en los encabezados._

UDP se prefiere en casos donde la velocidad de entrega es prioritaria sobre la confiabilidad de los datos. Esto se observa en aplicaciones como streaming multimedia, telefonía por internet y juegos en línea, donde la inmediatez es esencial y la pérdida ocasional de paquetes no afecta significativamente la experiencia del usuario.

En cambio, TCP se utiliza en escenarios donde la confiabilidad de la entrega es crucial. Aplicaciones como el correo electrónico, la web y la transferencia de archivos requieren una garantía de que los datos llegarán correctamente y en el orden adecuado, TCP es la elección preferida.

Dificultades encontradas

- Mantener consistencia en todos los protocolos para mantener separada la aplicación y la implementación de los protocolos.

Conclusión

~~En esta sección se explayará sobre la conclusión a lo largo del desarrollo del trabajo práctico.~~