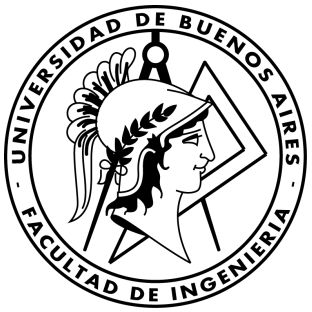


TP N°1: File Transfer

[TA048] Redes
Segundo cuatrimestre 2024
Grupo 3

Índice

1. Introducción

Para el desarrollo del siguiente trabajo práctico se pidió realizar una aplicación cliente servidor que permitiera realizar descargas y subidas de archivos, desarrollando un protocolo de aplicación para montar por sobre UDP de forma tal que se garantice el envío de datos confiable (reliable data transfer).

2. Hipótesis y suposiciones realizadas

Para que la aplicación pueda garantizar su correcto funcionamiento se realizan las siguientes suposiciones:

- El archivo a descargar/subir es lo suficientemente pequeño como para poder ser cargado en memoria por completo antes de enviarse. Esto es necesario ya que la lectura de cada archivo no se hace de forma secuencial, sino que se leen todos los bytes juntos para hacer el envío.
- Los host y los switches que conforman la red sobre la cual se utiliza la aplicación tienen un MTU (maximum transfer unit) de al menos 1028 bytes (tamaño máximo de los paquetes para el protocolo implementado). En caso contrario, los paquetes construidos por el protocolo serían fragmentados para ser enviados y eso podría ocasionar problemas.
- En ningún momento la cantidad de conexiones en simultaneo que recibirá el servidor será mayor a la cantidad de puertos disponibles para el mismo (alrededor de 64.000), ya que por cada conexión con un cliente que se haga el servidor reservará un puerto diferente.

3. Implementación

Los componentes fundamentales de la implementación de la aplicación solicitada son las siguientes:

- Protocolo de aplicación sobre UDP para garantizar el envío confiable.
- Definición y operaciones a realizar sobre los segmentos del protocolo definido.
- Servidor capaz de atender solicitudes en simultaneo.
- Operaciones Download y Upload.

3.1. Protocolo RDT

El enfoque utilizado para implementar un RDTP sobre UDP fue partir el protocolo en dos partes. La primera es un protocolo de intercambio de mensajes entre las aplicaciones basado en HTTP que cuenta con los siguientes mensajes:

```
"""
                    UPLOAD                      DOWNLOAD

               UP /file_name\ndata          DOWN /file_name\n


                     OK                          ERROR

               OK /file_name\ndata         ERR /file_name\ndata
"""
```

Los clientes y el servidor utilizan este protocolo para solicitar y enviar información a través de la interfaz de sockets provista por la segunda parte del protocolo, la cual provee un servicio de entrega confiable.
El detalle de la implementación de esta parte del protocolo se puede encontrar en el archivo message.py de la entrega.

La segunda parte del protocolo consta de dos clases, RdpStream y RdpListener.

RdpListener

```
class RdpListener:

    def bind(cls, ip: str, port: int, log: bool = False) → RdpListener:
        ...
    def accept(self) → RdpStream:
        ...
    def addr(self) → tuple[str, int]:
        ...
    def close(self):
        ...
```

Es utilizada en la aplicación por el servidor para escuchar conexiones provenientes de clientes en la dirección especificada al realizar el bind. Al recibir una conexión, luego de que se complete correctamente el handshake entre cliente y servidor, crea un RdpStream asociado a ese cliente en particular para poder continuar con la comunicación.

RdpStream

```python
class RdpStream:

    def connect(cls, ip: str, port: int, log: bool = False) → RdpStream:

    def send(self, data: bytes, winsize: int = 1):

    def recv(self, winsize: int = 1) → bytes:

    def addr(self) → tuple[str, int]:

    def peer_addr(self) → tuple[str, int]:

    def close(self):
```

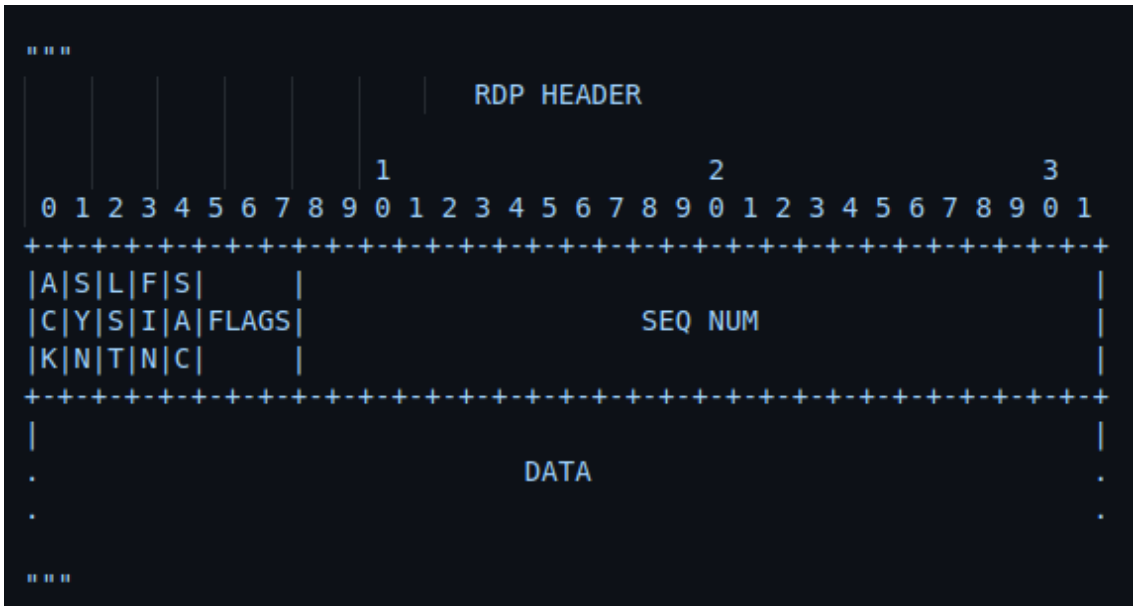
Es utilizado tanto por los clientes como el servidor. Permite establecer una conexión con un RdpListener, e intercambiar información con el mismo asegurando el envío confiable.

El parámetro winsize permite especificar el tamaño de la ventana de envío de paquetes. En caso de que no se especifique el tamaño de la ventana o se le asigne valor 1, el protocolo se comportará como un protocolo Stop&Wait, esperando el ACK de cada paquete antes de enviar el siguiente. En caso contrario, el protocolo realizará el envío de paquetes con una ventana del tabaño especificado apliacando Selective ACK.

El detalle completo de la implementación de esta sección del protocolo RDT puede encontrarse en el archivo socket.py de la entrega.

3.2. Segmento

La estructura del segmento implementado para el RDT es la siguiente:



Composición del paquete
El paquete consta de un header de 4 bytes, y el payload.

- El primer byte corresponde a los posibles flags del paquete. Estos son:
  - **ACK**: Bit de acknowledgement, utilizado para dara acuso de recibido de todos los paquetes hasta el número de secuencia del paquete actual.
  - **SYN**: Bit se sincronización utilizado en handshake al inicializar una nueva conexión.
  - **LST**: Bit para comunicar que este es el último paquete de la comunicación.
  - **FIN**: Bit utilizado para comunicar el fin de la conexión.
  - **SAC**: Bit utilizado para dar acuso de recibido del paquete del número de secuencia actual.
- Los próximos tres bytes corresponden al número de secuencia del paquete actual.
- Todos los demás bytes del segmento corresponden al payload o la información que transporta el paquete.

Funcionalidad de la clase Segment
La clase segment implementa métodos que permiten serializar y des-serializar el paquete, consultar por los flags encendidos, y construir paquetes con determinados flags encendidos.
El detalle completo de la implementación del segmento puede encontrarse en el archivo segment.py de la entrega.

3.3. Servidor

```python
if __name__ == "__main__":
    config = ServerConfig(argv)

    log = config.verbose()
    ip, port = config.addr()
    listener = RdpListener.bind(ip, port, log=log)
    ...
    for stream in listener:
        thread = Thread(target=handle_client, args=(stream, storage, winsize))
        threads.append(thread)
        thread.start()
```

El servidor se inicializa utilizando los parámetros provistos al ser ejecutado y escucha las conexiones que ingresen por el puerto configurado. Al recibir una conexión, crea un nuevo stream con un nuevo puerto exclusivo para esta conexión, y envía dicha conexión a un nuevo hilo, guardando su handle. Una vez que recibe una señal para detenerse, cierra el listener y hace el join de todos los threads creados.

3.4. Operaciones Download y Upload

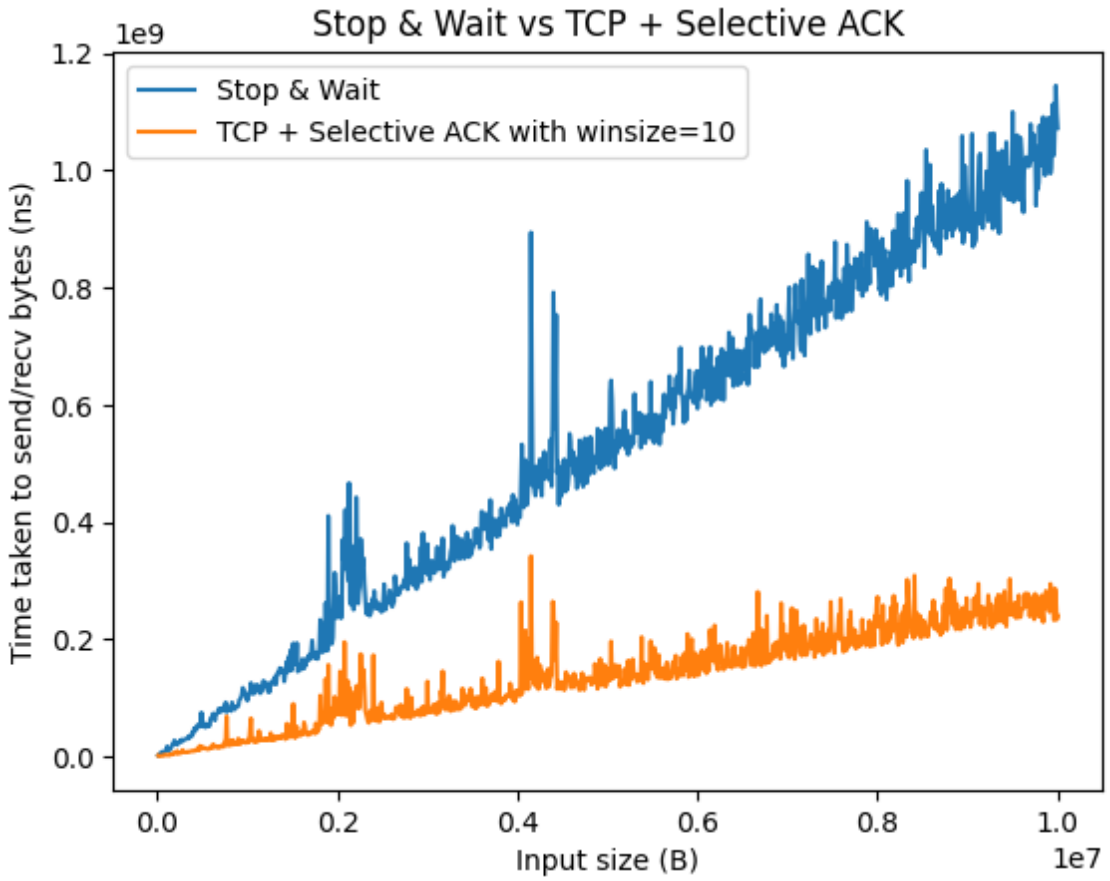
Las operaciones de Upload y Download tienen una implementación sencilla que puede consultarse en los archivos upload.py y download.py respectivamente.
Existe una pequeña diferencia entre la interfaz sugerida en el enunciado y la implementación actual, ya que se agrega en esta última el parámetro -w <window size>el cual permite especificar el tamaño de la ventana de paquetes a utilizar.
Las nuevas interfaces serían entonces:

```
"""
usage : upload.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-s FILEPATH] [-n FILENAME] [-w WSIZE ]

optional arguments:
    -h, --help     show this help message and exit
    -v, --verbose  increase output verbosity
    -q, --quiet    decrease output verbosity
    -H, --host     server IP address
    -p, --port     server port
    -w, --winsize  window size for pipeline streaming
    -s, --src      source file path
    -n, --name     file name
"""
```

```
"""
usage : download.py [-h] [-v | -q] [-H ADDR] [-p PORT] [-d FILEPATH] [-n FILENAME] [-w WSIZE]

optional arguments:
    -h, --help      show this help message and exit
    -v, --verbose   increase output verbosity
    -q, --quiet     decrease output verbosity
    -H, --host      server IP address
    -p, --port      server port
    -w, --winsize   window size for pipeline streaming
    -d, --dst       destination file path
    -n, --name      file name
"""
```

4. Pruebas

Se realizaron mediciones de tiempos para la ejecución del protocolo utilizando la metodología Stop&Wait y TCP + SACK. Se notó una gran mejoría en los rendimientos temporales al utilizar una ventana de envío de paquetes en lugar de esperar el ACK de cada paquete antes de enviar el próximo.



5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

En esta arquitectura los clientes asumen que existe una computadora (servidor) escuchando en una dirección IP + puerto específico lista para resolver pedidos de los clientes y responder siguiendo algún tipo de protocolo de comunicación previamente establecido.

A diferencia de la arquitectura P2P en la que no existe un servidor dedicado con este propósito sino que los clientes pueden o no tomar este rol en ciertos escenarios de comunicación entre ellos.

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

Permite definir las reglas de como se va a llevar a cabo la comunicacion entre 2 aplicaciones. Por ejemplo especifica el formato que llevan los datos.

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

Partimos el protocolo en dos partes, una simula ser un protocolo de capa de transporte el cual utiliza UDP e implementa la transferencia de datos confiable. La otra parte vive por encima de esta y utiliza su interfaz para implementar un servicio de transferencia de archivos confiable.

No queriamos juntar todo en un solo lugar porque pensamos que podría complejizar mucho las cosas tener todo mezclado. De esta forma las aplicaciones podrían cambiar la implementación del socket que utilizan por uno con TCP y no debería haber ningún problema.

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?

UDP brinda 2 servicios. - Multiplexación y Demultiplexación - Integridad de datagramas

TCP brinda 6 servicios. - Multiplexación y Demultiplexación - Integridad de segmentos - Fiabilidad - Orden de segmentos - Control de flujo - Control de conjestión

Multiplexación y Demultiplexación: Es el modo de diferenciar de que proceso es cada segmento, por ejemplo juntar los distintos segmentos de varios procesos que tienen que salir por el mismo cable y luego al momento de recibir segmentos poder diferenciar de quien es cada uno.

Integridad de segmentos: Ambos servicios implementan un sistema de checksum que se aplica sobre el header y los datos para garantizar (o por lo menos tener una baja probabilidad) de que los datos no hayan sido corrompidos durante el traslado del segmento por la red.

Fiabilidad: Los datos que se envian podemos garantizar que llegan al otro proceso.

Orden de segmentos: La información llega en el mismo orden en el que se envía.

Control de flujo: Es el control de la taza de envío de los segmentos para no desaprovechar ancho de banda pero a la vez no abrumar al otro proceso receptor de nuestros segmentos.

Control de conjestión: Es un sistema que permite regular la velocidad de envío teniendo un cuidado más panoramico donde suponemos que si estamos perdiendo segmentos entonces es porque la red está saturada. Esto funciona porque muchas aplicaciones deciden utilizar TCP como protocolo de capa de transporte y al auto regularse intenta proteger a la red de que no caiga en un estado inutilizable donde segmento que se envie se pierda.

5.4.1. ¿Cuando es apropiado utilizar cada uno?

UDP es un buen protocolo en situaciones donde la velocidad de transferencia es clave y no nos importa tanto que se pierdan datagramas, de ser así podriamos implementar un protocolo de aplicación que brinde este servicio y no los demás que brinda TCP (como en este trabajo práctico).

TCP brilla en ser completo. Brinda casi todas las comodidades que pueda tener un protocolo de transporte, no brinda por ejemplo encriptación de datos. Lo más destacable siendo la fiabilidad y orden de segmentos, la información va a llegar en el orden esperado, el control de flujo ayuda a mantener buen performance sin abrumar la cola de segmentos del otro lado y el control de conjestión es un servicio que se brinda a toda la red.

Siendo que ambos protocolos implementan integridad de segmentos, no es algo que sea una ventaja para ninguno. En cuanto a multiplexación y demultiplexación, es el mínimo indispensable para ser considerado un protocolo de transporte. De otra manera no podríamos determinar a que proceso entregar un datagrama/segmento recién recibido de la capa de red.

6. Dificultades encontradas

Al momento de realizar pruebas de conexiones en simultaneo con el servidor, sucedía que luego de enviarse cierta cantidad de paquetes en una red de mininet tanto el servidor como los clientes entraban en un deadlock, donde ambos quedaban esperando información del otro y la ejecución del programa se detenía. Estos problemas no sucedían al realizar pruebas locales en nuestros propios equipos. Finalmente el problema encontrado fue que el tamaño de paquete para los segmentos que creábamos (4096 bytes) estaba por encima del MTU (maximum transfer unit) de los hosts y los switches de la topología que utilizabamos en mininet, los cuales tienen por default un MTU de 1500 bytes. Esto provocaba que nuestros paquetes fueran seccionados nuevamente para poder ser transmitidos por los enlaces, lo cual generaba un comportamiento no determinado en la aplicación.

7. Conclusión

El trabajo presentado consistió en la implementación de una aplicación cliente-servidor para realizar transferencia de archivos utilizando un protocolo de aplicación sobre UDP que garantice un envío confiable de datos (RDT). La implementación se dividió en dos partes: un protocolo de intercambio de mensajes basado en HTTP y un protocolo de transporte confiable sobre UDP.

Durante la implementación y pruebas, se observaron mejoras significativas en la eficiencia del protocolo al utilizar una ventana de envío de paquetes en lugar de un esquema de espera y confirmación (Stop&Wait). Sin embargo, se presentaron dificultades cuando los paquetes excedían el MTU de la red, causando fragmentación y problemas en la transmisión.

En conclusión, mediante la resolución del trabajo se logró implementar un protocolo de transferencia confiable sobre UDP. Además, el uso de ventanas de envío optimizó el rendimiento del protocolo en términos de tiempo de transferencia, validando así las suposiciones y objetivos propuestos al inicio del proyecto.