Informe TP1:
Transferencia de archivos usando protocolo UDP

Integrantes:

Herrera Alexis	104639
La Torre Gabriel	87796
Lozano Ramiro	98263
Valmaggia Matias	105621

Grupo 10

Repo: https://github.com/mvalmaggia/fiuba_redes_tp1

Índice:

Introducción

El objetivo de este trabajo práctico es crear una aplicación de cliente/servidor con conexión UDP que cumpla con las bases de una transferencia de datos segura (RDT):

- Integridad de datos enviados (ya sea por corrupción de estos o por pérdidas en el camino)
- Datos recibidos en orden

Nuestra aplicación va a consistir de 3 ejecutables:

- Cliente:
  - *upload.py*: permite subir un archivo al servidor con el que se conecta
  - *download.py*: permite bajar un archivo del servidor y guardarlo en algún directorio local
- Servidor:
  - *start-server.py*: programa principal del servidor. Se encarga de escuchar y actuar acorde al pedido de un cliente

Hipótesis y suposiciones

- El MSS (Maximum Segment Size) por lo general suele ser de 1500 Bytes. Teniendo en cuenta que el encabezado en un segmento UDP es de 60 Bytes, nuestros paquetes manejados en software que contienen partes del archivo a transferir van a tener un tamaño de como mucho 1440 Bytes (Encabezado + Data)
- Los archivos a subir/bajar al servidor no tienen restricción de formato (pueden ser archivos de texto, PDFs, imágenes)
- El servidor debe ser apagado manualmente. De no ser así, seguirá esperando a que algún cliente desee conectarse con él. El apagado puede hacerse con una interrupción de consola (Ctrl + C)

Implementación

Para poder establecer una comunicación entre cliente y servidor se utilizan los sockets de tipo UDP del módulo socket de Python. Mediante estos podemos enviar bytes pero no podemos asegurar que su transmisión sea confiable.

Para lograr producir los servicios que promete RDT, decidimos desarrollar una clase llamada **Packet** que se encargue de encapsular la data a enviar, y de dar la suficiente información al cliente/servidor para continuar con la comunicación entre ambos. El formato de esta clase es el siguiente:

```
class Packet:
    seq_num: int
    checksum: int
    ack: bool
    fin: bool
    query_type: QueryType
    data: bytes
```

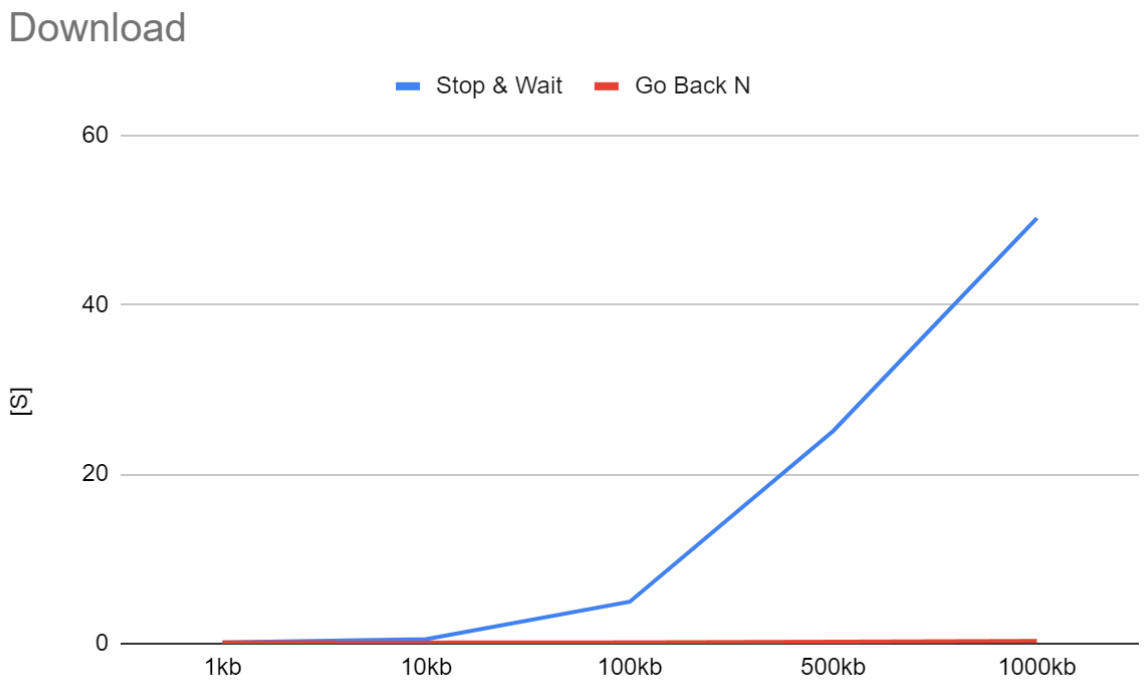
Sus campos son:

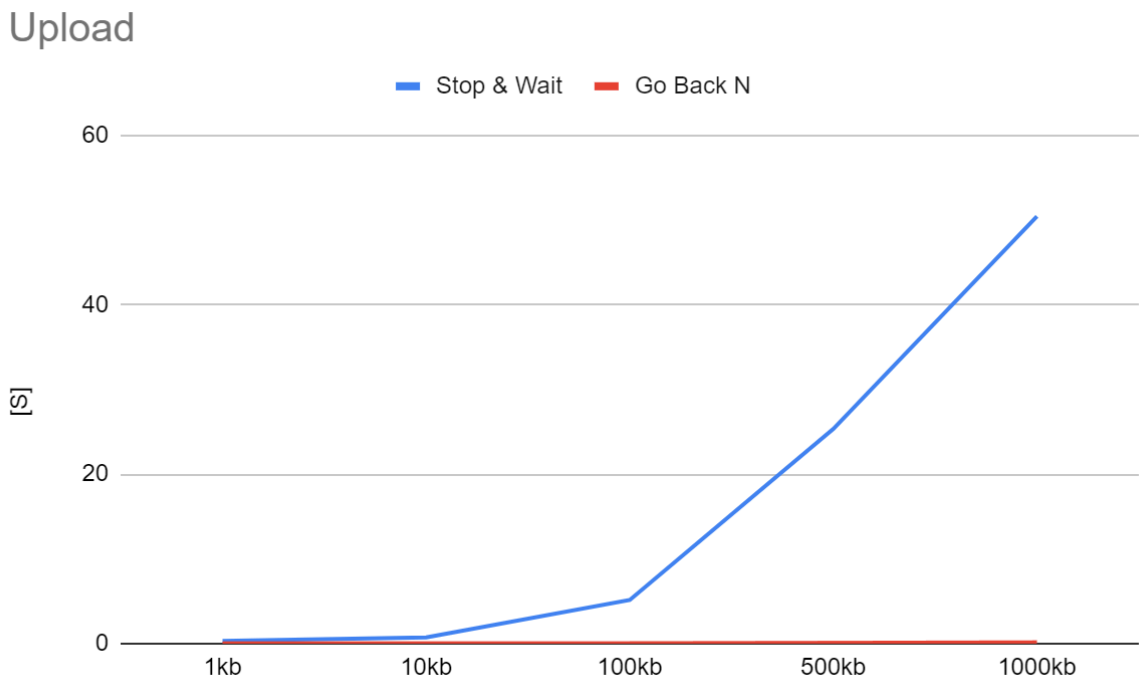
- *seq_num*: número de secuencia de la comunicación
- *checksum*: chequeo de integridad del Paquete
- *ack*: booleano de reconocimiento que el paquete con *seq_num* fue recibido
- *fin*: booleano que indica si se desea terminar la conexión
- *query_type*: tipo de paquete
    - None
    - UPLOAD
    - DOWNLOAD
- *data*: aquí es donde se guarda un bloque de data en bytes

Pruebas

Para comparar la eficiencia de los distintos algoritmos de retransmisión implementados realizamos pruebas variando los tamaños de los archivos a transferir y la pérdida de paquete tanto para el upload como para el download. A partir de esto planteamos una serie de gráficos que nos permitirán comparar los datos obtenidos, en los mismos se ve cómo varía el tiempo de ejecución en segundos con la variación de tamaño de archivo en kilobytes. Utilizamos para Go-Back-N una ventana de 10.

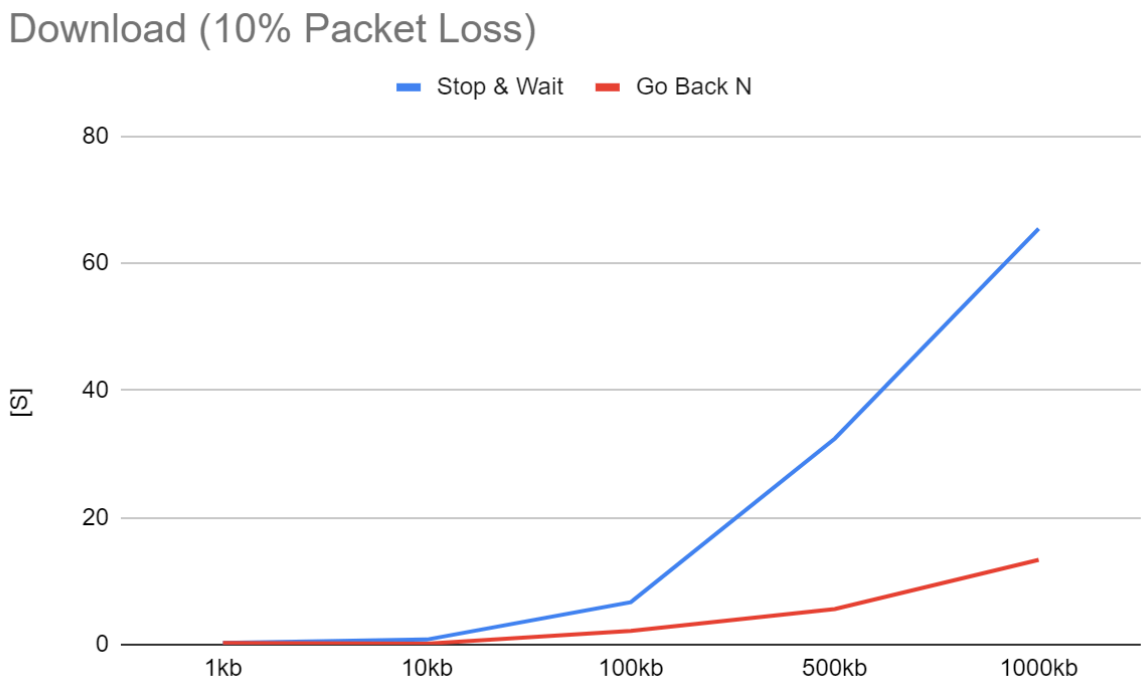
En primer lugar, tenemos la comparación cuando no hay packet loss:

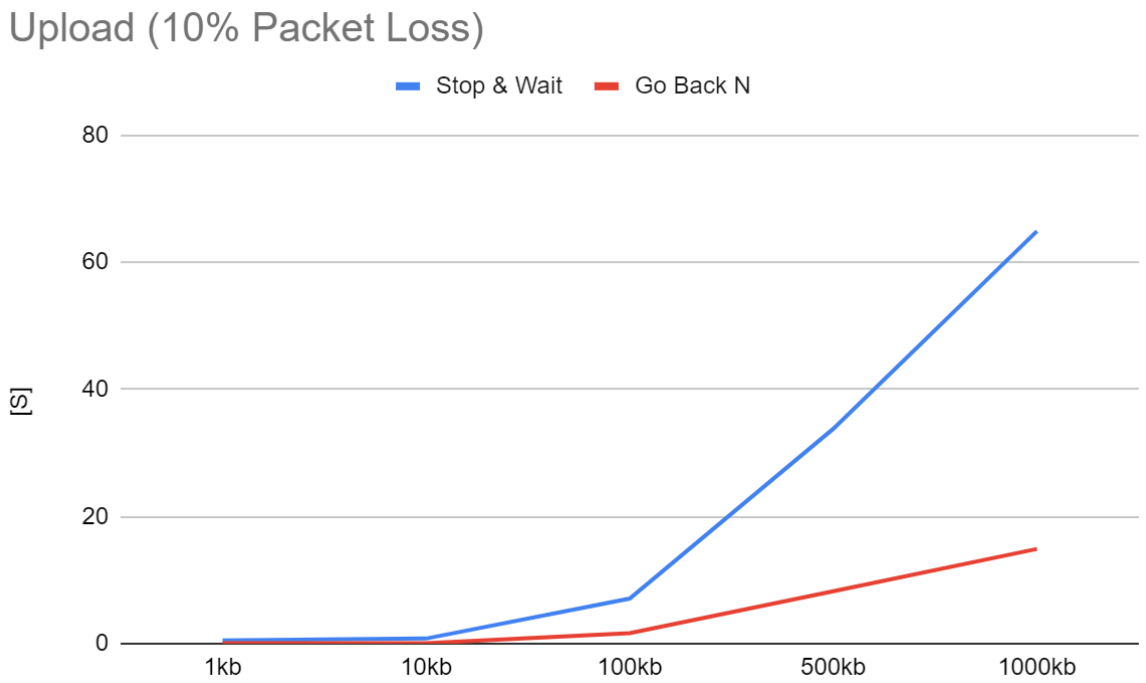




De estos gráficos podemos ver que la velocidad de transmisión con Go-Back-N se mantiene casi constante y el tiempo que tarda es incomparable con el de Stop & Wait.

Luego, lo correspondiente a una pérdida de paquete del 10%, la cual fue simulada utilizando comcast:





De estos gráficos podemos ver que si bien los tiempos de Stop & Wait siguen siendo casi lineales, para Go-Back-N tienen una diferencia notable de rendimiento.

Preguntas

- ### Describir la arquitectura Cliente/Servidor
  En la arquitectura de cliente/servidor tenemos un host server, que se comunica con muchos hosts llamados clientes. Cada cliente podrá realizar requests al servidor, quien deberá proveer los servicios solicitados. Además, un servidor siempre tendrá una dirección de IP fija, a la cual siempre se comunicaran los clientes para realizar sus solicitudes.

- ### ¿Cual es la función de un protocolo de capa de aplicación?
  La principal función de un protocolo de capa de aplicación es crear aplicaciones de red que puedan ser ejecutadas en distintos “end systems” sin diferencia. Al crear una nueva aplicación no deberá ser necesario considerar que pueda correr en cierto tipo de red o dispositivo, ya que la abstracción del protocolo permitirá que pueda ser ejecutada en todos por igual.

- ### Detallar el protocolo de aplicación usado en este trabajo
  El servidor recibe conexiones UDP, y para empezar a realizar un trabajo de UPLOAD o DOWNLOAD debe recibir un paquete incluyendo la solicitud.
  Al recibir uno, envía un ACK indicando que inicia la transmisión, ya sea esperando los paquetes en caso de un UPLOAD o empieza a enviar en caso de un DOWNLOAD.

Para confirmar un paquete recibido, se envía un paquete con el flag ACK activado y el seq_num del paquete recibido + 1 si era el paquete esperado. Luego dependiendo del protocolo de retransmisión realizará lo siguiente:

- Stop and Wait: En este caso el emisor envía un paquete y no envía el siguiente hasta que se haya recibido el ack correspondiente
- Go Back N: El emisor puede envíar hasta N paquetes sin esperar a que el receptor responda, después de eso, debe esperar a que el receptor testifique que recibió el primer paquete. En caso de que no se reciba un ACK del primer paquete en la ventana de N paquetes dentro de un tiempo determinado, o que se siga recibiendo un ACK del paquete anterior al principio de la ventana, se reenviará desde el último bien recibido (o sea, el primer paquete de la ventana actual).

- Los protocolos TCP y UDP, ¿Qué servicios ofrecen? ¿Cuáles son sus características? ¿Cuándo es apropiado usar uno o el otro?

  El protocolo UDP realiza la menor cantidad de tareas posibles sobre los datos que se desean enviar. Esto puede parecer algo malo, pero en determinados casos puede resultar más acorde a nuestras necesidades. Ya que apenas hay algún tipo de procesamiento sobre los datos a enviar, la aplicación tendrá más control sobre cuando esta misma es enviada, ya que el protocolo lo hará de manera directa.
  Además, no mantiene una conexión estable durante la comunicación entre hosts, lo cual permitirá que no se retrase el envío con el proceso de establecer la conexión inicial. Por último, se utiliza un header muy pequeño, de apenas 8 bytes, lo cual también agiliza el proceso.
  En cambio, el servicio principal que busca dar TCP es la transferencia confiable de datos en conexión que son llamadas point-to-point, es decir, entre un único host que envía y uno único que recibe. Para poder lograrlo utiliza los siguientes mecanismos:
    - Checksum: para detectar errores en los datos transmitidos.
    - Timer: para realizar un timeout y retransmitir los paquetes que se perdieron.
    - Número de secuencia: para poder enumerar en secuencia a los paquetes enviados.
    - ACK y NACK: utilizados por el host que está recibiendo paquetes para notificar al que está enviando si el paquete fue recibido o hubo error, respectivamente.
    - Window: se utiliza un rango de números de secuencia llamado “window” para determinar cuáles son los paquetes que pueden ser enviados.

  El servicio que provee es llamado “full-duplex” ya que los dos hosts que forman parte de la conexión pueden mandar y recibir paquetes de manera simultánea.

Cuando es apropiado utilizar cada uno de los protocolos es algo que dependerá puntualmente de las necesidades de cada aplicación. Si necesitamos que los datos sean transmitidos de la manera más rápida posible sin importar que a veces se pierdan, nos va a convenir utilizar UDP. Pero si en cambio, priorizamos que todos los paquetes lleguen correctamente a su destino, aunque esto implique un retraso en la transmisión, nos convendrá optar por el protocolo TCP.

Dificultades encontradas

- El primer problema que nos encontramos fue como enviar los datos en paquetes, ya que para asegurar que la transferencia sea confiable era necesario agregar más información a los datos a enviar.
  Para solucionar esto, planteamos una clase llamada Packet, la cual ya fue descripta anteriormente. Luego para poder enviar esto mediante el socket era necesario hacerlo en bytes. Para lograr esto utilizamos el módulo pickle, que mediante sus métodos loads() y dumps(), nos permite decodificar y codificar una instancia de la clase en bytes.

Conclusión

En condiciones óptimas, cuando no hay pérdida de paquetes, Go-Back-N resulta ser una mucho mejor opción como protocolo de retransmisión, manteniéndose en un orden casi constante, mientras que Stop & Wait crece de manera casi lineal.

Al introducir una pérdida de paquetes comienza a parecerse el rendimiento de ambos. Esto tiene sentido ya que un protocolo solo debe retransmitir el paquete que se pierde, en cambio el otro deberá transmitir todos los enviados luego del que fue perdido. Por lo tanto, para Go-Back-N el rendimiento resulta variante, ya que si siempre se pierde el primer paquete que es retransmitido la transferencia resultará mucho más lenta que Stop & Wait.

Por lo tanto, podemos decir que Go-Back-N será en casi todos los casos, incluso con pérdida de paquetes, más veloz que Stop and Wait, pero pueden darse los casos aislados donde la misma sea notablemente peor.