FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Trabajo Practico 1: File Transfer

(TA045) Redes
Curso 02 - Hamelin
Segundo cuatrimestre de 2025
Fecha: 02/10/2025


| Alumno | Padrón | Email |
|---|---|---|
| Alejandro, Pablo Martín | 98021 | palejandro@fi.uba.ar |
| Gutiérrez, Matías | 92172 | mgutierrez@fi.uba.ar |
| Escobar Barreto, Noel Alejandro | 111151 | nescobar@fi.uba.ar |
| Pinargote, Tom | 111689 | tpinargote@fi.uba.ar |
| Olalla, Nervo | 111731 | nolalla@fi.uba.ar |

# 1. Introducción

El objetivo de este trabajo practico es el diseño de *File Transfer* usando la arquitectura de *cliente-servidor*, en donde la aplicación va a tener su propio protocolo para la comunicación entre los servicios. El trabajo implementa las siguientes aplicaciones:

- UPLOAD: el cliente manda un archivo al servidor.
- DOWNLOAD: el cliente le pide al servidor un archivo.
- SERVER: servidor que maneja los archivos.

Además, cada una debe cuplir las siguientes condiciones:

- Se debe poder cargar/descargar archivos binarios
- Las aplicaciones deben demorar menos de 2 minutos para archivos de 5 MB.
- Las aplicaciones debe contemplar por lo menos 2 condiciones de error.
- El protocolo debe garantizar la entrega de paquetes para una perdida del 10 % en la red.
- El servidor debe procesar clientes en forma concurrente, es decir, permitir la transferencia de archivos con múltiples clientes.
- El protocolo debe contemplar flujo de errores.
- El protocolo debe implementar utilizando UDP como protocolo de transporte.
- El protocolo debe implementar una version *Stop & Wait* y otra versión utilizando el mecanismo de *error-recovery* de *Selective Repeat*.

Por ultimo, cada aplicación debe estar programada en el lenguaje *Python* y en archivos diferentes.

## 2. Hipotesís y suposiciones realizadas

Para el desarrollo del trabajo, consideramos algunas condiciones importante. Como se menciono antes, cada aplicación, va a tener su archivo propio ejecutable. Dichos programas tiene la opción de *versobe* o *quiet*, en el caso de que el usuario setee las dos opciones, se devolverá un error del programa.

El protocolo tiene una base de UDP que nos permite tener las cosas básicas de un protocolo de comunicación, luego para la conexión con varios clientes, tomamos el *Three-way Handshake* del TCP modificado para poder enviar al servidor si el cliente va a descargar o cargar un archivo y aparte le pasamos el nombre del archivo. Además, agregamos los flags típicos ACK, PSH, SYN y FIN de TCP, junto con dos flags nuevos FNAME y OP para mandar el nombre del archivo y la operación. La figura 1 muestra el header del protocolo, mientras la figura 2 muestra la forma que un cliente se conecta al servidor.

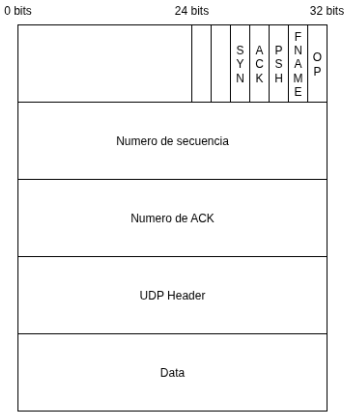


Figura 1: Header del protocolo

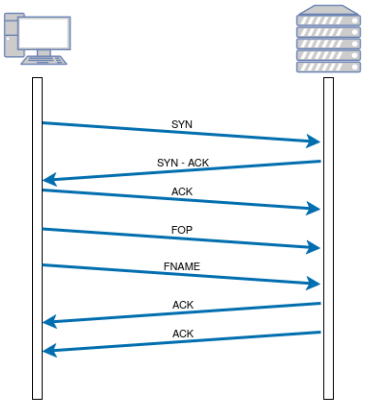


Figura 2: Three-way Handshake

Al momento de terminar el envió de datos, la aplicación que esta mandando el archivo inicia una secuencia activando el flag FIN, luego se responde con un FIN-ACK, y se finaliza la conexión con un ACK de respuesta. Se puede observar la secuencia en la siguiente imagen

Otra hipótesis tomada, es el uso del método por defecto. En el caso de que el usuario no seleccione un protocolo se tomara por defecto el protocolo *Stop & Wait*.

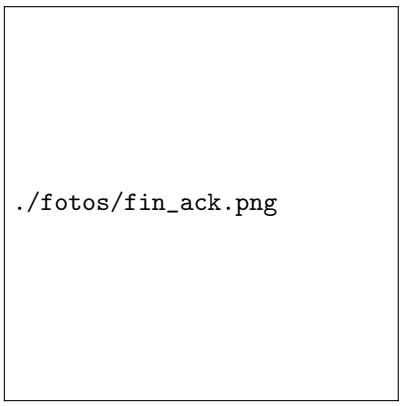


Figura 3: Secuencia FIN

# 3. Implementación

La implementan de los programas, se utilizaron diferentes clases para la comunicación de las etapas. La imagen 4 muestra un diagrama de clases simplificado para mostrar como se comunican los objetos en los diferentes programas.

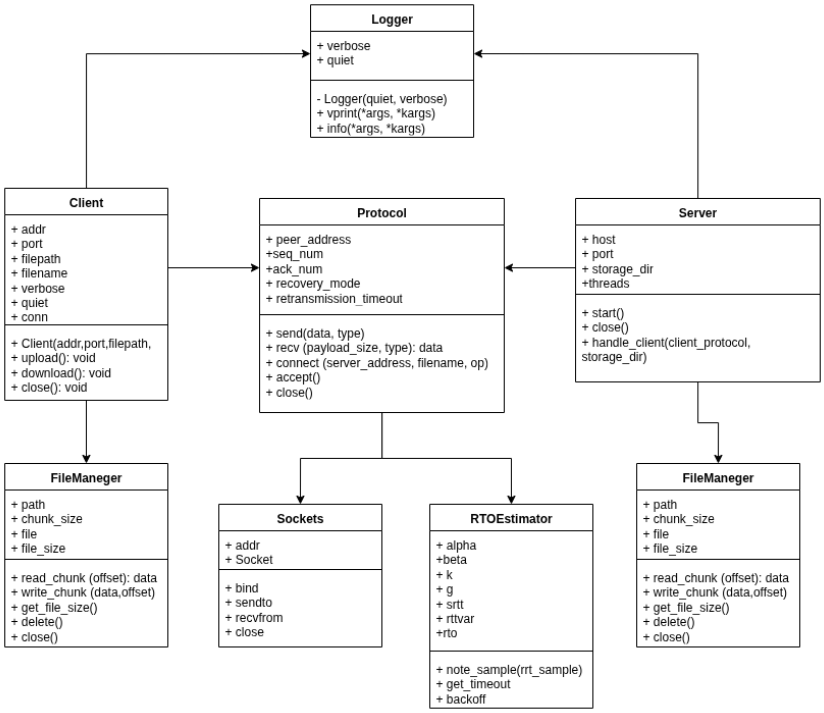


Figura 4: Diagrama de clases simplificado

Las clases principales son *Client*, *Server* y *Protocol*. Estas clases tienen los métodos que se utilizan para la comunicación entre el cliente y el servidor. Si el cliente quiere cargar un archivo, en el código principal, se ejecutara el metodo de *upload*, en cambio, si quiere descargar un archivo, en el código principal, se ejecutara el método de *download*. Como el server es el mismo, se mantiene una única método de *start*. El servidor tiene una lista de *threads* que van a ir guardando cada uno de los clientes para poder tener múltiples conexiones. Una vez hecha la conexión, se envían los datos del archivo por *chunk* que tienen un valor por defecto de 400, cuando el cliente/servidor termine de recorrer el archivo, entonces se envía un paquete con le flag FIN pendido que comienza a terminar la conexión con el cliente/servidor.

Tanto como el servidor como los clientes, usan la misma clase de protocolo. Esta clase tiene el socket para el envió de los datos. Dentro de esta clase, se implementa los protocolos de *recovery*. Los métodos de *recv* y *send* tiene una condición que ejecutan métodos privados, los cuales implementan los algoritmos de *Stop & Wait* y *Selective Repeat*.

El protocolo de *Stop & Wait* es el más simple que envía un paquete y espera su respectivo ACK para poder enviar el siguiente paquete. En el código, se implemento los *recovery* en ambos lados (tanto como el servidor como del cliente). Si hay una perdidas de paquete, es decir, el receptor no recibió la respuesta, se definió un timeout que realiza un re intento del ultimo segmento enviado. A continuación se muestra una imagen del *Stop & Wait*

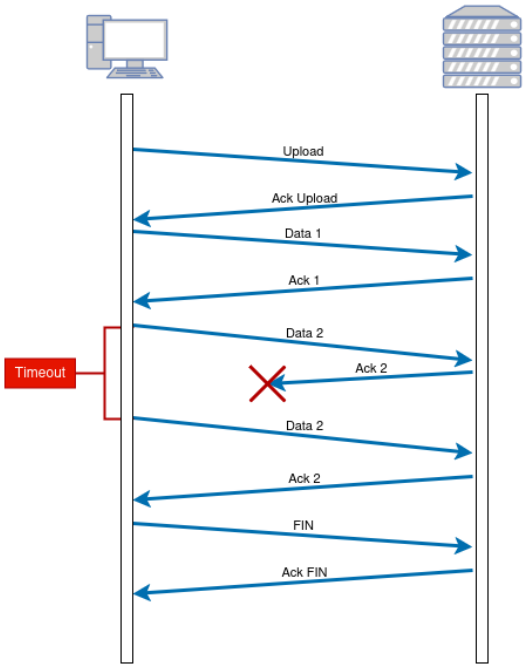


Figura 5: Stop & Wait

Mientras el *Selective Repeat* tiene la diferencia que no necesita esperar el ACK para poder enviar el siguiente paquete, esto lo realiza utilizando una venta que determina la cantidad de paquetes que se pueden guardar antes de esperar el primer ACK enviado, esto quiere decir, si el tamaño de la ventana es de 5, entonces el cliente puede mandar 5 paquetes sin esperar el ACK y luego el servidor va a tener una ventana de 5 para la recepción de los paquetes. A medida que se van recibiendo los ACK, se continua enviando los siguientes mensajes. En el caso que se pierde un paquete, el servidor le responderá al cliente siempre con el mismo ACK para que el cliente envié ese paquete faltante, a su vez, el servidor se va guardando los mensajes que no sean dicho paquete para evitar repreguntar. La imagen 6 muestra este comportamiento.

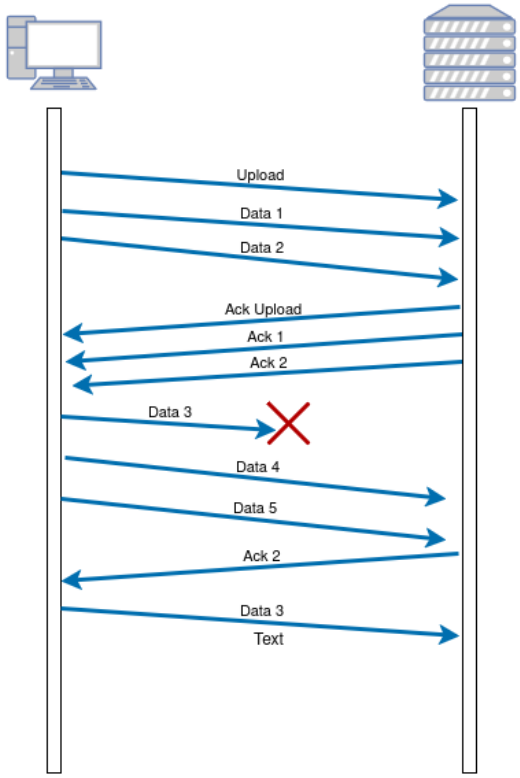


Figura 6: Stop & Wait

# 4. Pruebas

En esta sección, vamos a estar analizando algunas capturas del *Wireshark*. La idea es mostrar el funcionamiento del protocolo y si cumple los requerimientos mencionados anteriormente, para eso, se hicieron capturas y se estará mostrando a continuación.

# 5. Preguntas

## 5.1. 1. Describa la arquitectura Cliente-Servidor

La arquitectura de Cliente-Servidor es un modelo informático donde las tareas se dividen entre clientes y servidores. Los clientes se encargar en consumir servicios que el servidor ofrece de forma centralizada. En este trabajo, nuestro clientes podían acceder a la base de datos del servidor para poder descargar o guardar archivos en su base de datos, mientras que el servidor se encargaba de realizar las consultas que los clientes realizaban.

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

El protocolo en la capa de aplicación sirve para poder determinar un conjunto de reglas y estándares para la comunicación entre los servidores y clientes, es decir, establecen como se intercambia la información, además garantiza que los datos se transfieran de manera fluida y se interpreten correctamente entre diferentes sistemas.

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

## 5.4. 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?

La capa de transporte ofrece los protocolos de TCP y UDP, estos tienen características únicas que determinan que servicios pueden otorgar. En el caso de UDP, siendo este un protocolo de comunicación básico, tiene las características principales de un protocolo de comunicación, esto implica que el header del mismo sea el más simple de todos. UDP se encarga del envió de los datos pero no tiene un control elaborado para saber si el dato llego correctamente, dentro del header se encuentran los puerto de destino y origin, junto con el length de los datos y por ultimo el checksum que chequea muy básicamente si el paquete no fue alterado. Al ser un protocolo simplificado, esto nos permite enviar paquetes a gran velocidad pero dejamos a manos del cliente/servidor de chequear el correcto envió de los paquetes, tanto como el orden como la seguridad del paquete. Un caso frecuente de uso de UDP es el envió de audios y vídeo, debido a que se necesita una gran velocidad de envíos.

Por el otro lado, tenemos el protocolo de TCP que a diferencia de UDP, el cual no necesita una conexión previa, necesita establecer una conexión con el cliente/servidor usando un *Three-Way Handshake* debido a que el protocolo tiene envíos de flags que no permite determinar la recepción de un paquete. Otra ventaja que tiene, permite a los usuarios evitar el control del orden de los paquetes, debido a que presenta un numero de secuencia junto con un numero de ACK. Al igual que UDP, tiene los bytes de puerto de origen, puerto de destino y checksum. Como el header puede recibir opciones, es necesario agregar bytes de offset permitiendo así saber cuando comienza los datos. La característica mas importante es el uso de la ventana que nos permite pushear mensajes sin esperar el ACK para acelerar la red, a su ves, si se ralentiza la red, se puede modificar la ventana y así evitar la perdida de paquete por congestionamiento. El caso mas común de ver el protocolo TCP es cuando se navega por la web junto con el protocolo HTTP de la capa de aplicación

# 6. Dificultades encontradas

Dentro del trabajo practico, se presentaron diferentes dificultades pero la mas grande es el envió de un archivo de 5 MB dentro del tiempo establecido. A su vez, al ser un trabajo en equipo, siempre esta la dificultad de la comunicación haciendo que varias personas trabajaron sobre lo mismo a pesar de tener reuniones constantemente. El diseño del protocolo y de las clases casi no tuvieron dificultad debido a los braind storming del equipo.

En el desarrollo, los algoritmos de recovery fueron un problema constante, principalmente el de *selective repeat* y el tamaño de la ventana, desde archivos enviados pero no se recibieron bytes, hasta problema de comunicación entre el servidor y el cliente. Utilizando *Wireshark* pudimos encontrar los problemas en nuestro código para poder desarrollar una aplicación que funcione correctamente.

# 7. Conclusión

El diseño y el desarrollo de un protocolo que corre en UDP es un trabajo que si no se hace con cuidado puede demorar un tiempo largo. Es importante la definición de un protocolo para la aplicación previa a realizar una implementación, además es importante generar clases para poder dividir las responsabilidades del programa.

La arquitectura cliente-servidor permite el desarrollo de aplicación que usuarios pueden usar sin la necesidad de saber como se comunican los cliente, pero es importante un buen desarrollo de un protocolo de comunicación. Para concluir, es importante tener un método de recuperación de paquetes para evitar la perdida de los mismo, porque sino se podría estar enviando información incompleta.