

Trabajo Práctico 2

UDP File Transfer

Introducción a los Sistemas Distribuidos (75.43)
Facultad de Ingeniería, Universidad de Buenos Aires

1^{er} Cuatrimestre 2021

Integrantes

Mauro Parafati - 102749 - mparafati@fi.uba.ar
Taiel Colavecchia - 102510 - tcolavecchia@fi.uba.ar
Yuhong Huang - 102146 - yhuang@fi.uba.ar

Índice

1. Introducción	2
2. Hipótesis y suposiciones realizadas	2
2.1. Cliente	2
2.2. Servidor	2
3. Implementación	3
3.1. Lineamientos Generales	3
3.2. Utilización de Sockets UDP	3
3.3. Flujo de ejecución	3
3.3.1. Servidor	3
3.3.2. Cliente	4
3.4. Protocolo de aplicación	4
3.5. Protocolos RDT	5
3.5.1. Stop & Wait	5
3.5.2. Go-Back-N	5
4. Caso de prueba	5
5. Comparación de protocolos RDT	10
5.1. Sin pérdida de paquetes	10
5.1.1. Stop & Wait	10
5.1.2. Go-Back-N sin buffering	10
5.1.3. Go-Back-N con buffering	10
5.1.4. Conclusiones	10
5.2. Con 10% de pérdida de paquetes	11
5.2.1. Stop & Wait	11
5.2.2. Go-Back-N sin buffering	11
5.2.3. Go-Back-N con buffering	11
5.2.4. Conclusiones	11
5.3. Con 20% de pérdida de paquetes	12
5.3.1. Stop & Wait	12
5.3.2. Go-Back-N sin buffering	12
5.3.3. Go-Back-N con buffering	12
5.3.4. Conclusiones	12
6. Dificultades encontradas	14
7. Conclusión	15

1. Introducción

El presente trabajo práctico tiene como objetivo la creación de una **aplicación de red**.

Para tal finalidad, será necesario comprender cómo se comunican los procesos a través de la red, y cuál es el modelo de servicio que la **capa de transporte** le ofrece a la **capa de aplicación**.

Además, para poder lograr el objetivo planteado, se aprenderá el uso de la interfaz de **sockets UDP** y los principios básicos de la **transferencia de datos confiable** (del inglés Reliable Data Transfer, **RDT**).

2. Hipótesis y suposiciones realizadas

Se listan a continuación las distintas hipótesis y suposiciones realizadas en la etapa de análisis del problema, para luego ser tenidas en cuenta a la hora de implementar la solución propuesta.

2.1. Cliente

- Puede especificarse cualquier ruta fuente para subir los archivos, incluyendo sub-directorios.
- Puede especificarse cualquier ruta destino para descargar los archivos, incluyendo sub-directorios.
- Si el *host* dado para la conexión con el servidor no es válido, o si el mismo no está aceptando conexiones en el puerto especificado, el cliente finalizará su ejecución (luego de un período dado de re-intentos – ver *README.md* –).
- En el modo *quiet* (*-q* o *--quiet*) se debe loggear lo mínimo e indispensable, en nuestro caso consideramos sólo los errores críticos. Cualquier otra cosa no será incluida en este modo.

2.2. Servidor

- El nombre de los archivos que se ofrecen para descargar se considera como el nombre del archivo y su extensión.
- No es necesario que se puedan crear sub-directorios para almacenar archivos. Todos los archivos comparten directorio raíz de la ruta especificada para la ejecución del servidor.
- No hay parámetros obligatorios para correr el servicio, es decir que si no se especifica *host*, *puerto* o *directorio de almacenamiento* se utilizarán valores por defecto. (ver *constants.py*)
- En el modo *quiet* (*-q* o *--quiet*) se debe loggear lo mínimo e indispensable, en nuestro caso consideramos sólo los errores críticos. Cualquier otra cosa no será incluida en este modo.
- Si se recibe un archivo con un nombre que ya existe en el directorio de archivos del servidor, el mismo se reemplaza por el recibido.

3. Implementación

3.1. Lineamientos Generales

Se utilizaron *hashbangs* al comienzo de los archivos para lograr la ejecución de los scripts en `python3`:

```
#!/usr/bin/python3
```

Para cada script a ejecutar, se utilizó la biblioteca nativa de `python argparse` para lograr los requerimientos de opciones para cada uno.

3.2. Utilización de Sockets UDP

Se optó por implementar una clase `Socket` que encapsule la clase nativa del lenguaje para facilitar su uso, y con el objetivo también de poder reutilizar gran parte del código escrito en la capa de aplicación diseñada para el trabajo práctico anterior (TCP File Transfer).

3.3. Flujo de ejecución

3.3.1. Servidor

El Servidor debe mantener su naturaleza concurrente, es decir, debe seguir siendo capaz de manejar múltiples consultas de forma paralela. En TCP, cada vez que un cliente realiza una *request* se establece una conexión y se obtiene un *peer* socket a través del cual ambos extremos se pueden comunicar de forma directa. Sin embargo, en UDP no se tiene tal conexión y por consiguiente, tampoco se tiene tal *peer* socket. Se debe notar además, que al no contar con el mismo se deberá realizar el trabajo de demultiplexar manualmente.

En este caso, lo que se tiene es un sólo socket UDP del lado del Servidor, al que los clientes deberán enviar mensajes para que su solicitud sea atendida. El desafío es entonces, teniendo que recibir en un punto centralizado, distribuir los mensajes para mantener el comportamiento concurrente.

Se diseñó una arquitectura que permite al Servidor esto último por medio de una abstracción: en cada **ClientHandler** asociado a cada Cliente que se comunica con el servidor, se tiene una cola que funciona como *buffer* de recepción. Cuando el servidor quiera obtener paquetes de una determinada dirección, basta entonces con que los saque de la cola correcta. Pero, ¿cómo se logra que estas colas se llenen de forma correcta?

Así como en el primer trabajo práctico se tenía un hilo independiente que se encargaba de *aceptar* conexiones, ahora se tiene uno que se encarga sólo de recibir todo lo que puede del socket **compartido**. Cuando recibe un datagrama UDP, lo deriva al buffer de recepción que corresponda según su dirección y puerto de origen (*demux*), permitiendo de esta forma que las colas puedan ser accedidas concurrentemente de forma segura.

Al igual que en el primer trabajo práctico realizado sobre TCP, en este también se comenzará un hilo independiente por fuera del principal cuando se inicia el servidor. El principal, esperará en un *prompt* con dos opciones:

- `s`: Para imprimir *stats* (estadísticas) del servidor.
- `q`: Para terminar ordenadamente la ejecución del programa.

Cada **ClientHandler** mantiene también su comportamiento a grandes rasgos respecto del anterior trabajo práctico: comienza su ejecución con la request de un cliente, realiza las acciones para resolver la misma y luego finaliza su ejecución de forma ordenada.

3.3.2. Cliente

La ejecución del cliente (independientemente del script a ejecutar) funciona únicamente sobre el hilo principal, finalizando la misma ante cualquier error de lectura de archivos o de conexión con el servidor. La comunicación con este último se realiza mediante el protocolo de aplicación descrito a continuación.

3.4. Protocolo de aplicación

Se utilizó el mismo protocolo de aplicación que para el trabajo práctico anterior. Teniendo en cuenta las consideraciones de las correcciones recibidas del mismo, en este caso: resolver las ineficiencias de mandar varias “idas y vueltas” paquetes antes de que se comience a transferir la información.

Para resolver el problema planteado, se decidió que el primer mensaje que envía un cliente al servidor será un paquete de tamaño fijo: 256 bytes.

Este mensaje se ve de las siguientes maneras para cada tipo de solicitud de un cliente:

- *download-file:*

```
+-----+-----+-----+
| op_code |          filename          | padding |
+-----+-----+-----+
```

- *upload-file:*

```
+-----+-----+-----+
| op_code | filesize (int) |          filename          |
+-----+-----+-----+
```

- *list-files:*

```
+-----+-----+-----+
| op_code |          padding          |
+-----+-----+-----+
```

Teniendo que los tamaños disponibles para cada uno de estos es:

- **op_code:** 1 byte
- **int:** 8 bytes
- **filesize:** 247 bytes (*max*)
- **padding** (será el valor sobrante de los campos para completar 256 bytes)

Puede ser notado, que el tamaño máximo está dado por el necesario para la subida de un archivo, ya que se debe transferir el tamaño del archivo, así como el nombre del mismo.

De esta forma, el flujo de envíos de mensajes entre un cliente y servidor es ahora simplificado y en un solo mensaje de respuesta se puede indicar si es válida o no la consulta, y así que se comience a transferir la información directamente. Se debe mencionar, que para este protocolo de aplicación es transparente qué tipo de protocolo de transferencia confiable se utiliza, y no debe mantener *headers* en cada mensaje para verificar su llegada confiable ni similar.

3.5. Protocolos RDT

¿Qué es un protocolo RDT?

Un protocolo de transferencia de datos confiable, es uno que nos provee la garantía de que un dado mensaje llegará, completo y sin errores, al destino.

Para poder utilizar sockets UDP es necesario definir un protocolo de transferencia de datos confiable, ya que UDP no provee las garantías que sí provee TCP.

En el presente trabajo práctico se desarrollaron dos protocolos RDT a fin de poder compararlos: **Stop & Wait**, y **Go-Back-N**. A continuación se realiza una breve introducción a la idea general, para luego pasar a los casos de prueba.

3.5.1. Stop & Wait

Este tipo de protocolos tienen como filosofía principal lograr una transferencia confiable de forma sencilla manteniendo en todo momento **un único paquete en vuelo**, es decir, que aquel proceso que envíe los datos no va a enviar nada nuevo hasta estar seguro de que el otro extremo recibió correctamente el paquete en cuestión.

Esto permite una mayor simplicidad de código y de comprensión del mismo, pero trae como desventajas una importante lentitud en la transferencia propia de su naturaleza sincrónica y secuencial.

3.5.2. Go-Back-N

Go-Back-N es uno de los protocolos RDT que, a diferencia de Stop & Wait, permiten tener más de un paquete en vuelo en un momento dado, para de esta forma aumentar la velocidad y eficiencia de la transferencia.

Sus ventajas son contrarrestadas por un código más complejo, ya que deben manejarse múltiples números de secuencia, así como un posible aumento en los paquetes totales enviados (dependiendo de varios factores: qué valor de N se utilice, qué tanta pérdida de paquetes exista, y el mecanismo usado en el receptor para bufferear paquetes futuros).

A fines del presente trabajo, se desarrollaron dos versiones de este protocolo que permiten comparar su rendimiento:

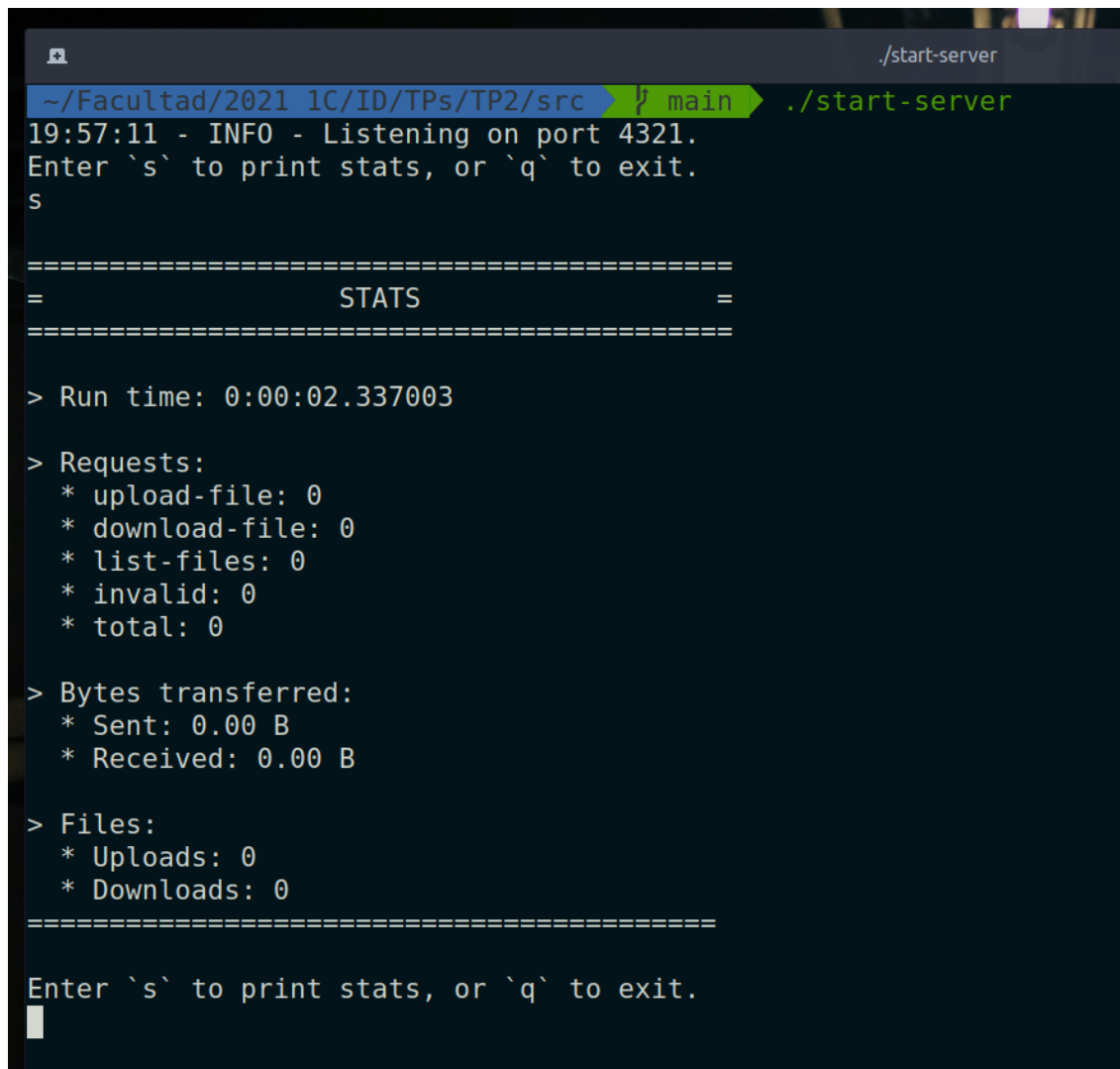
1. Una primer versión simple en la cual el receptor no implementa *buffering*, por lo que cuando un paquete llega de forma adelantada se lo descarta.
2. Una segunda versión en la cual el receptor implementa buffering, para así intentar contrarrestar las posibles pérdidas de paquetes.

En las comparaciones, como se verá más adelante, se observó que dependiendo de los parámetros del entorno, puede haber mucha diferencia entre ambas versiones.

4. Caso de prueba

Terminada la implementación de la solución propuesta, se procedió a llevar a cabo el detallado seguimiento de un caso de prueba para poder ver los distintos comandos en acción.

Primero, se levanta el servidor mediante el comando `start server`:

A terminal window titled './start-server' showing the execution of a server program. The prompt is '~/.Facultad/2021 1C/ID/TPs/TP2/src >'. The user enters 'main' and then './start-server'. The output shows the server listening on port 4321. The user enters 's' to print stats, which are displayed in a formatted table. The stats show zero requests, bytes transferred, and files. The user enters 'q' to exit.

```
./start-server
~/Facultad/2021 1C/ID/TPs/TP2/src > main > ./start-server
19:57:11 - INFO - Listening on port 4321.
Enter `s` to print stats, or `q` to exit.
s
=====
=                               =
=                   STATS                   =
=====

> Run time: 0:00:02.337003

> Requests:
* upload-file: 0
* download-file: 0
* list-files: 0
* invalid: 0
* total: 0

> Bytes transferred:
* Sent: 0.00 B
* Received: 0.00 B

> Files:
* Uploads: 0
* Downloads: 0
=====

Enter `s` to print stats, or `q` to exit.
q
```

Figura 1: Inicio del servidor

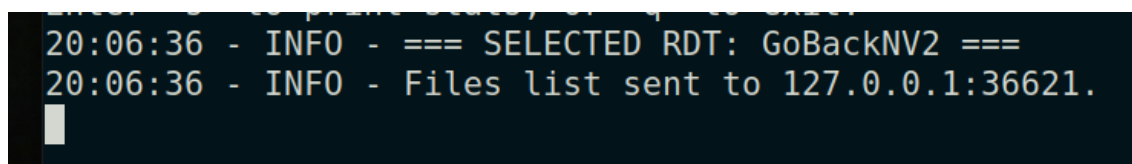
Podemos ver, imprimiendo estadísticas, que el servidor no ha recibido aun ningún tipo de consultas.

Ahora, un cliente decide solicitar información sobre la lista de archivos que el servidor posee, mediante el comando `list-files`:



```
mp@seven:~/Facultad/2021 1C/ID/
~/Facultad/2021 1C/ID/TPs/TP2/src > main > ./list-files
20:06:36 - INFO - === SELECTED RDT: GoBackNV2 ===
No hay archivos disponibles en el servidor.
~/Facultad/2021 1C/ID/TPs/TP2/src > main > 
```

Figura 2: list-files, vista del Cliente

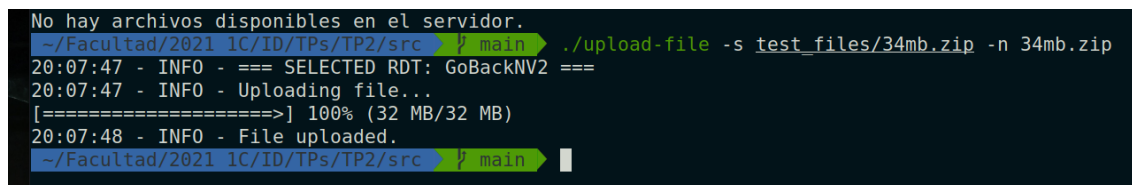


```
20:06:36 - INFO - === SELECTED RDT: GoBackNV2 ===
20:06:36 - INFO - Files list sent to 127.0.0.1:36621.

```

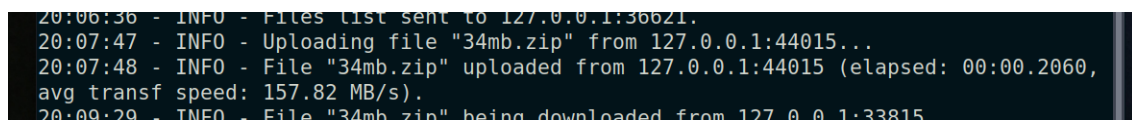
Figura 3: list-files, vista del Servidor

El cliente decide entonces, al ver que no hay archivos, subir un primer archivo llamado 34mb.zip mediante el comando `upload-file`:



```
No hay archivos disponibles en el servidor.
~/Facultad/2021 1C/ID/TPs/TP2/src > main > ./upload-file -s test_files/34mb.zip -n 34mb.zip
20:07:47 - INFO - === SELECTED RDT: GoBackNV2 ===
20:07:47 - INFO - Uploading file...
[=====>] 100% (32 MB/32 MB)
20:07:48 - INFO - File uploaded.
~/Facultad/2021 1C/ID/TPs/TP2/src > main > 
```

Figura 4: upload-file, vista del Cliente



```
20:06:36 - INFO - Files list sent to 127.0.0.1:36621.
20:07:47 - INFO - Uploading file "34mb.zip" from 127.0.0.1:44015...
20:07:48 - INFO - File "34mb.zip" uploaded from 127.0.0.1:44015 (elapsed: 00:00.2060,
avg transf speed: 157.82 MB/s).
20:09:29 - INFO - File "34mb.zip" being downloaded from 127.0.0.1:33815
```

Figura 5: upload-file, vista del Servidor

Finalizada la carga, el cliente es desconfiado y quiere verificar que el archivo se haya subido correctamente, para lo que lo descarga utilizando el comando `download-file`:


```
20:07:48 - INFO - File uploaded.
~/Facultad/2021 1C/ID/TPs/TP2/src > main > ./download-file -n 34mb.zip -d test_files/34mb_from_server.zip
20:09:29 - INFO - === SELECTED RDT: GoBackNV2 ===
20:09:29 - INFO - Downloading file...
[<=====] 100% (32 MB/32 MB)
20:09:29 - INFO - File downloaded.
~/Facultad/2021 1C/ID/TPs/TP2/src > main >
```

Figura 6: download-file, vista del Cliente

```
avg transf speed: 197.02 MB/s).
20:09:29 - INFO - File "34mb.zip" being downloaded from 127.0.0.1:33815...
20:09:29 - INFO - File "34mb.zip" downloaded from 127.0.0.1:33815 (elapsed: 00:00.1279
, avg transf speed: 254.21 MB/s).
```

Figura 7: download-file, vista del Servidor

Finalmente, en su sistema operativo, el cliente aplica el hash `sha256sum` a ambos archivos (el que descargó del servidor vs. el que subió desde su ordenador) con el fin de verificar que, efectivamente, son iguales:

```
~/Facultad/2021 1C/ID/TPs/TP2/src > cd test_files
~/Facultad/2021 1C/ID/TPs/TP2/src/test_files > main > sha256sum 34mb.zip 34mb_from_server.zip
f1f0ac0e43d087e131ab896a8d60cb5c1a964650339f054c61dad0177327b24d 34mb.zip
f1f0ac0e43d087e131ab896a8d60cb5c1a964650339f054c61dad0177327b24d 34mb_from_server.zip
~/Facultad/2021 1C/ID/TPs/TP2/src/test_files > main >
```

Figura 8: Chequeo de correctitud

Por último, se cierra el servidor, viendo que imprimió estadísticas finales:

```
20:09:29 - INFO - File "34mb.zip" being downloaded from
20:09:29 - INFO - File "34mb.zip" downloaded from 127.0.
, avg transf speed: 254.21 MB/s).
q

=====
=                               STATS                               =
=====

> Run time: 0:13:35.948936

> Requests:
* upload-file: 1
* download-file: 1
* list-files: 1
* invalid: 0
* total: 3

> Bytes transferred:
* Sent: 32.59 MB
* Received: 33.59 MB

> Files:
* Uploads: 1
* Downloads: 1

=====

~/Facultad/2021 1C/ID/TPs/TP2/src > ↵ main > |
```

Figura 9: Cierre del Servidor con estadísticas finales

5. Comparación de protocolos RDT

Se corrieron pruebas sobre la red local utilizando [comcast](#) para simular la pérdida de paquetes y poder así plantear distintos escenarios.

Bajo cada escenario planteado, se transfirieron los mismos archivos de distintos tamaños para analizar los tiempos, las velocidades promedio de carga y descarga, así como la tasa de retransmisión.

5.1. Sin pérdida de paquetes

5.1.1. Stop & Wait

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	1.0 ms	103.80 MB/s	100.1 KB	0 %
1 MB	6.5 ms	153.16 MB/s	1.00 MB	0 %
10 MB	60.0 ms	166.54 MB/s	10.00 MB	0 %
100 MB	536 ms	186.41 MB/s	100.1 MB	0 %

Cuadro 1: Stop & Wait sin pérdida de paquetes

5.1.2. Go-Back-N sin buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	0.9 ms	113.17 MB/s	100.3 KB	0 %
1 MB	5.7 ms	175.04 MB/s	1 MB	0 %
10 MB	67.7 ms	147.61 MB/s	10 MB	0 %
100 MB	661.2 ms	151.23 MB/s	100 MB	0 %

Cuadro 2: Go-Back-N v1 sin pérdida de paquetes

5.1.3. Go-Back-N con buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	1.3 ms	76.07 MB/s	100.3 KB	0 %
1 MB	6.2 ms	160.51 MB/s	1 MB	0 %
10 MB	67.5 ms	148.24 MB/s	10 MB	0 %
100 MB	771.2 ms	129.67 MB/s	100 MB	0 %

Cuadro 3: Go-Back-N v2 sin pérdida de paquetes

5.1.4. Conclusiones

En este primer escenario en el que no hay pérdida de paquetes, vemos que los tres protocolos se comportan de forma similar, lo que tiene mucho sentido, puesto que al no haber pérdida de paquetes deberían estar pasando por flujos similares. Una primer conclusión que podemos observar

de estas tablas es que la transferencia del archivo de $100kB$ se ve significativamente afectada por los mensajes iniciales del protocolo, mientras que esto se vuelve insignificante para los archivos de tamaño mayor.

5.2. Con 10 % de pérdida de paquetes

5.2.1. Stop & Wait

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	1.01 s	99.51 KB/s	116.48 kB	aprox. 16 %
1 MB	2.16 s	474.97 KB/s	1.14 MB	aprox. 14 %
10 MB	2.33 s	4.29 MB/s	10.91 MB	aprox. 9 %
100 MB	12 s	7.95 MB/s	111.45 MB	aprox. 11 %

Cuadro 4: Stop & Wait con 10 % pérdida de paquetes

5.2.2. Go-Back-N sin buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	908 ms	110.11 KB/s	104.28 kB	aprox. 4 %
1 MB	1.19 s	855.30 KB/s	1.34 MB	aprox. 34 %
10 MB	517 ms	19.34 MB/s	13 MB	aprox. 30 %
100 MB	4.8 s	20.78 MB/s	125 MB	aprox. 25 %

Cuadro 5: Go-Back-N v1 con 10 % pérdida de paquetes

5.2.3. Go-Back-N con buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	907 ms	110.14 KB/s	120.28 kB	aprox. 20 %
1 MB	552 ms	1.81 MB/s	1.16 MB	aprox. 16 %
10 MB	877 ms	11.39 MB/s	12.13 MB	aprox. 21 %
100 MB	4.45 s	22.43 MB/s	120 MB	aprox. 20 %

Cuadro 6: Go-Back-N v2 con 10 % pérdida de paquetes

5.2.4. Conclusiones

Con pérdida de paquetes se puede observar claramente las diferencias entre los protocolos. Podemos sacar algunas conclusiones:

- Las velocidades de **Stop & Wait** son sustancialmente más bajas.
- En los archivos de tamaño pequeño, el inicio de la comunicación es muy significativo en la velocidad, mientras que observamos valores más estables en los archivos más pesados.

- **Stop & Wait** presenta las mejores tasas de retransmisión, manteniéndolas por debajo del 15 %.
- **GBN** presenta velocidades similares con y sin buffering, pero vemos que hay una diferencia en las tasas de retransmisión (lo que es esperable, ya que al usar buffering estamos contrarrestando ciertas pérdidas de paquetes).

5.3. Con 20 % de pérdida de paquetes

5.3.1. Stop & Wait

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	2.64 s	37.95 KB/s	149.25 kB	aprox. 49 %
1 MB	10 s	97.76 KB/s	1.57 MB	aprox. 57 %
10 MB	12 s	831.1 KB/s	15.82 MB	aprox. 58 %
100 MB	35 s	2.65 MB/s	155.87 MB	aprox. 56 %

Cuadro 7: Stop & Wait con 20 % pérdida de paquetes

5.3.2. Go-Back-N sin buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	1.05 s	99.60 KB/s	151.0 kB	aprox. 51 %
1 MB	4.18 s	239.17 KB/s	1.60 MB	aprox. 60 %
10 MB	3.45 ms	2.90 MB/s	17.4 MB	aprox. 74 %
100 MB	17 s	5.83 MB/s	188.3 MB	aprox. 88 %

Cuadro 8: Go-Back-N v1 con 20 % pérdida de paquetes

5.3.3. Go-Back-N con buffering

Tamaño de archivo	Tiempo de transferencia	Velocidad promedio	Bytes transferidos	Tasa de retransmisión
100 kB	2.91 s	34.4 KB/s	154.6 kB	aprox. 55 %
1 MB	3.12 s	321.1 KB/s	1.56 MB	aprox. 56 %
10 MB	6.19 s	1.62 MB/s	16.9 MB	aprox. 70 %
100 MB	8.70 s	11.5 MB/s	166.8 MB	aprox. 67 %

Cuadro 9: Go-Back-N v2 con 20 % pérdida de paquetes

5.3.4. Conclusiones

Al aumentar la pérdida de paquetes podemos notar los síntomas de los pros y contras de cada protocolo:

- **Stop & Wait** es el que más sufre en su velocidad, estando totalmente limitado por la mayor pérdida de paquetes.

- También aquí se nota cómo la capacidad de buffering beneficia con menores tasas de retransmisión alcanzando mayores velocidades de transferencia.

6. Dificultades encontradas

En este trabajo práctico se presentaron numerosas dificultades propias de la implementación de un protocolo de transferencia confiable (RDT). Para resolverlas, fue necesaria la discusión grupal, la investigación, y el refuerzo por medio de bibliografía.

Una dificultad puntual que aparece para cualquier implementación de un protocolo RDT que se construya por encima de UDP, es: ¿Qué hacer si se pierde el último *acknowledge* del flujo?

En nuestro caso, no fue simple darnos cuenta de que ante un escenario de pérdida de paquetes, nunca se puede contar con una certidumbre del 100 % de que este último paquete llega, y hay que terminar la ejecución por acumulación de timeouts, asumiendo que la información llegó.

Otra dificultad particular que radica en la implementación de **Go-Back-N** surgió cuando quisimos determinar los correspondientes *índices* de los paquetes enumerados, dado un *sequence number*. Inicialmente, propusimos una fórmula que permitía determinarlos, pero después nos dimos cuenta de que la misma no funcionaba cuando comenzaba a existir un desfase (dado que la enumeración de los paquetes siempre inicia en 0, pero los *sequence number* no siempre). Para solucionar este problema, se optó por la utilización de un diccionario que permite mapear los *sequence number* recibidos a los índices originales.

7. Conclusión

A partir de este trabajo práctico, se logró tomar consciencia de las dificultades que se presentan a la hora de construir un protocolo de transferencia confiable por encima de UDP, permitiendo de esta forma entender el valor de TCP para este tipo de aplicaciones.

Por otra parte, se pudo comparar **Go-Back-N** y **Stop & Wait** mediante distintas mediciones (con distintas configuraciones y frente a distintos escenarios de pérdida de paquetes), llegando a la conclusión de que **Go-Back-N** presenta una mayor *performance*, pero que **Stop & Wait** minimiza la retransmisión de paquetes.

Para finalizar, se considera que la experiencia de diseñar y utilizar una aplicación compuesta por protocolos de distintas capas (de aplicación, de transporte, etc) fue muy formativa y permitió tomar mayor contacto con la interfaz que provee en este caso Python para el manejo de sockets UDP.