

File Transfer

[75.43] Introducción a los sistemas distribuidos
1C2022

Apellido y Nombre	Padrón	Email
Mateo Capón Blanquer	104258	mcapon@fi.uba.ar
Ignacio Iragui	105110	iiragui@fi.uba.ar
Santiago Pablo Fernandez Caruso	105267	sfernandezc@fi.uba.ar
Gonzalo Sabatino	104609	gsabatino@fi.uba.ar
Federico Jose Pacheco	104541	fpacheco@fi.uba.ar

Índice

1. Introducción	2
2. Hipótesis y suposiciones realizadas	2
3. Implementación	2
3.1. Desacoplamiento entre la Transferencia de Datos Confiable y la lógica de la Trans-	
ferencia de Archivos	2
3.1.1. File transfer como parte del protocolo RDT	2
3.1.2. File transfer desacoplado del protocolo RDT	3
3.2. Arquitectura cliente - servidor	3
3.3. Modelo del File Transfer	4
3.4. Protocolo TLV	5
3.5. Protocolo RDT	5
3.5.1. Contención o múltiples sockets UDP abiertos	5
3.5.2. Modelado del protocolo	6
3.5.3. Servicios provistos	9
3.5.4. Especificaciones del protocolo	9
3.6. Posibles Mejoras	10
4. Pruebas	11
4.1. Análisis previo a las pruebas	11
4.2. SR - 20 clientes - 10 % perdida	11
4.3. SR vs SW - 1 cliente - 10 % perdida	12
4.3.1. Selective Repeat	12
4.3.2. Stop and Wait	12
4.4. SR vs SW - 1 cliente - sin perdida	12
4.4.1. Selective Repeat	13
4.4.2. Stop and Wait	14
4.5. SR vs SW - 1 cliente - 20 % perdida	14
4.5.1. Selective Repeat	15
4.5.2. Stop and Wait	16
4.6. SR vs SW - 3 clientes - 10 % perdida	16
4.6.1. Selective Repeat	16
4.6.2. Stop and Wait	17
4.7. Análisis de los resultados obtenidos	17
5. Preguntas a responder	17
6. Dificultades encontradas	18
6.1. Análisis de critical sections	18
6.2. Tests de casos bordes	19
6.3. Finalización de la comunicación	19
7. Conclusión	19
7.1. Conclusión sobre los protocolos	19
7.2. Performance y desacople entre las capas	20
7.3. Conclusión general del trabajo realizado	20

1. Introducción

El presente informe reúne la documentación del Trabajo Práctico 2 de la materia Introducción a los Sistemas Distribuidos - Facultad de Ingeniería Universidad de Buenos Aires.

El trabajo consiste de una aplicación de File Transfer entre un servidor y un cliente, y de un protocolo RDT para que se permita el intercambio de mensajes entre las aplicaciones.

2. Hipótesis y suposiciones realizadas

- **Archivos repetidos:** Cuando se da el caso de que un cliente quiere subir un archivo cuyo nombre ya está ocupado por otro archivo, el servidor lo resuelve agregando el primer número disponible entre paréntesis. Por ejemplo, si un cliente quiere subir un archivo llamado ejemplo.txt y en el servidor ya existe uno con ese nombre, lo guardará con el nombre ejemplo(1).txt. Al finalizar la carga dará aviso del cambio al cliente para que en caso de querer hacer un download sepa como solicitarlo.
Será equivalente el caso en que un cliente quiera descargar varias veces un archivo con el mismo nombre.
- **Eliminación de archivo antes fallas:** Si el archivo no se puede pasar por completo al otro extremo, este será eliminado.
Ésto puede tener que ver con fallas de alguno de los extremos, o porque alguno de ellos se cerró previo a que el traspaso de datos finalice correctamente.
- **Máximo tamaño posible del archivo:** Para el campo de número de secuencia utilizamos 3 bytes por lo que nos limita el máximo tamaño de archivo que puede ser transferido. Dado que son 3 bytes podemos enviar 2^{24} paquetes diferentes, como cada paquete es de 1280 bytes tenemos un límite aproximado de 20 Gigabytes.

3. Implementación

3.1. Desacoplamiento entre la Transferencia de Datos Confiable y la lógica de la Transferencia de Archivos

Antes de comenzar a desarrollar nuestra solución, nos encontramos ante dos posibles soluciones principales para el diseño general de nuestra aplicación. Una consiste en pensar a la transferencia de archivos como parte de la comunicación confiable de datos. Y la otra, en abstraernos del File Transfer, y construir dos capas de software: una, a menor nivel, que implemente la transferencia de datos confiable entre dos procesos, y otra que implemente la lógica de la transferencia de archivos. En esta sección detallamos las características de estas posibilidades y justificamos nuestra decisión.

3.1.1. File transfer como parte del protocolo RDT

La principal ventaja de esta solución es que implica una gran eficiencia tanto temporal, como espacial. Permite una disminución de la cantidad de bytes enviados a través de la red, dado que cada paquete de transferencia de un *chunk* del archivo necesita solamente de un campo que incluya la longitud del mismo, a diferencia de la segunda solución, donde se necesitan dos campos: uno para la longitud del paquete, y otro para la longitud del *chunk*.

Además, permite la llegada de datos en desorden. El número de secuencia de un paquete, se corresponde (más una constante) con el número del *chunk*. Por lo tanto, ante la llegada de un *chunk* desordenado, se podría mover el puntero de escritura del archivo, al lugar correspondiente, y luego volver a moverlo cuando llegue el anterior segmento. Poder escribir en el archivo *chunks* desordenados, a medida que se espera la llegada de otro paquete, implica una eficiencia temporal de la descarga. Sin extenderse tanto sobre esta propuesta, si se quisiera mejorar la eficiencia espacial, se podría evitar tener buffers en memoria de paquetes que fueron enviados, pero que no

fue confirmada su recepción. Esto es así, porque los datos ya están almacenados en el disco, en el archivo que se debe enviar.

3.1.2. File transfer desacoplado del protocolo RDT

A pesar de las enumeradas virtudes de la anterior solución, decidimos distinguir la aplicación en las dos capas mencionadas puesto que nos provee dos capas de software abstraídas, que se pueden utilizar independientemente: Una de File Transfer, y otra capa a más bajo nivel de protocolo RDT.

Esto nos permitió que, una vez diseñada la estructura general del sistema, podamos separarnos en dos grupos de desarrollo. Uno que se ocupe de la implementación de la RDT y otro, de la lógica de la transferencia de archivos. Al necesitar de un RDT, para la capa de más alto nivel, se utilizó provisoriamente el protocolo TCP.

Por ello la capa de más bajo nivel se resume en una API similar a la de TCP, que incluye tres clases principales: `ClientCommunicationSocket`, `ServerCommunicationSocket`, y `SocketListener`.

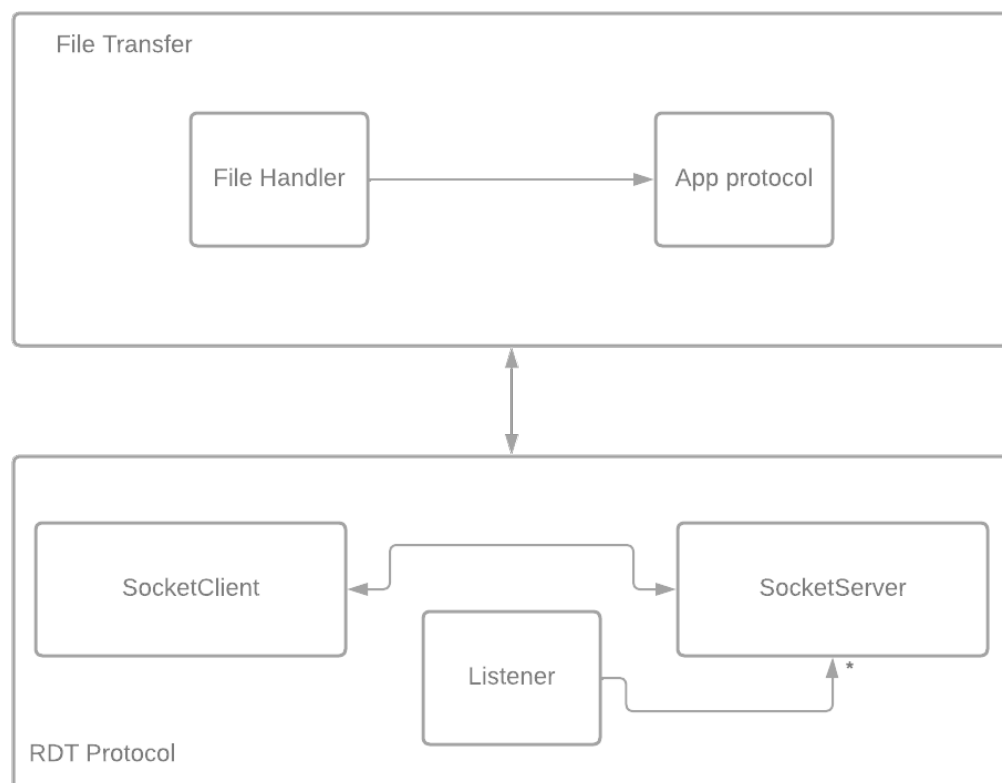


Figura 1: Separación de capas para el protocolo RDT y el de transferencia de archivos

3.2. Arquitectura cliente - servidor

Nuestro modelo la arquitectura cliente - servidor se realizó basándonos en TCP. El objeto instancia de la clase *Server* lanza un hilo que espera a aceptar conexiones a través de su *welcoming_socket*. Una vez que un cliente es aceptado, se crea un nuevo hilo, *ClientThread*, el cual se encarga de la comunicación con el cliente que se conectó. Luego, se limpia la lista de clientes que tiene el *Listener*, eliminando a aquellos clientes que ya finalizaron su conexión. De este modo, la lista de clientes no crece indefinidamente.

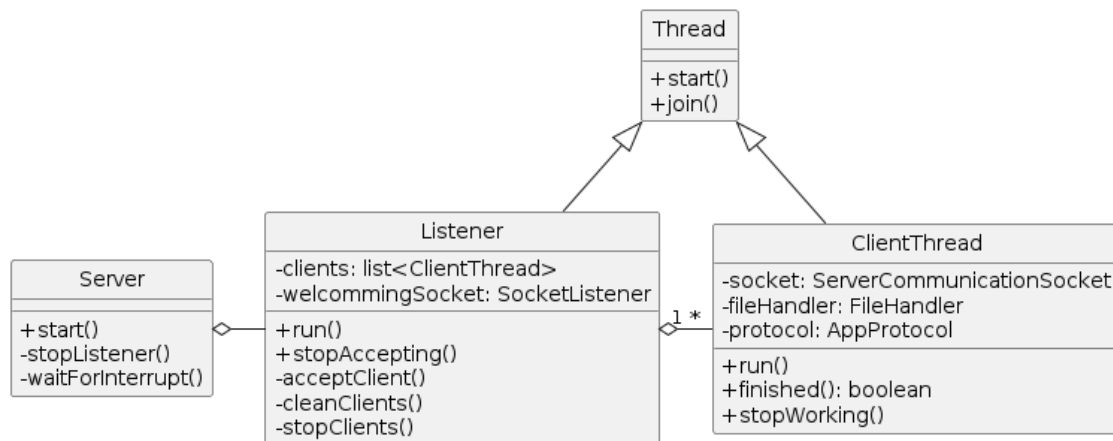


Figura 2: Diagrama de Clases del Servidor

3.3. Modelo del File Transfer

En la figura 3 presentamos la estructura del modelo del File Transfer. Tanto el objeto instancia de *Client* (del lado del cliente) como el de *ClientThread* (del lado del servidor), implementan una lógica mínima sobre la estructura general de la comunicación. Una vez que se decide la acción que se debe ejecutar, se delega el comportamiento de la misma en el objeto *fileHandler*, el cual llamará simétricamente desde el *cliente/servidor* al *upload/download*. A su vez, el *FileHandler* envía y recibe *chunks* utilizando al protocolo de comunicación que implementamos.

En concordancia a lo indicado en las hipótesis fue necesario agregar un monitor (*FilenamesMonitor*) para poder manejar de forma correcta los nombres de los archivos en caso de haber repetidos y así evitar race conditions.

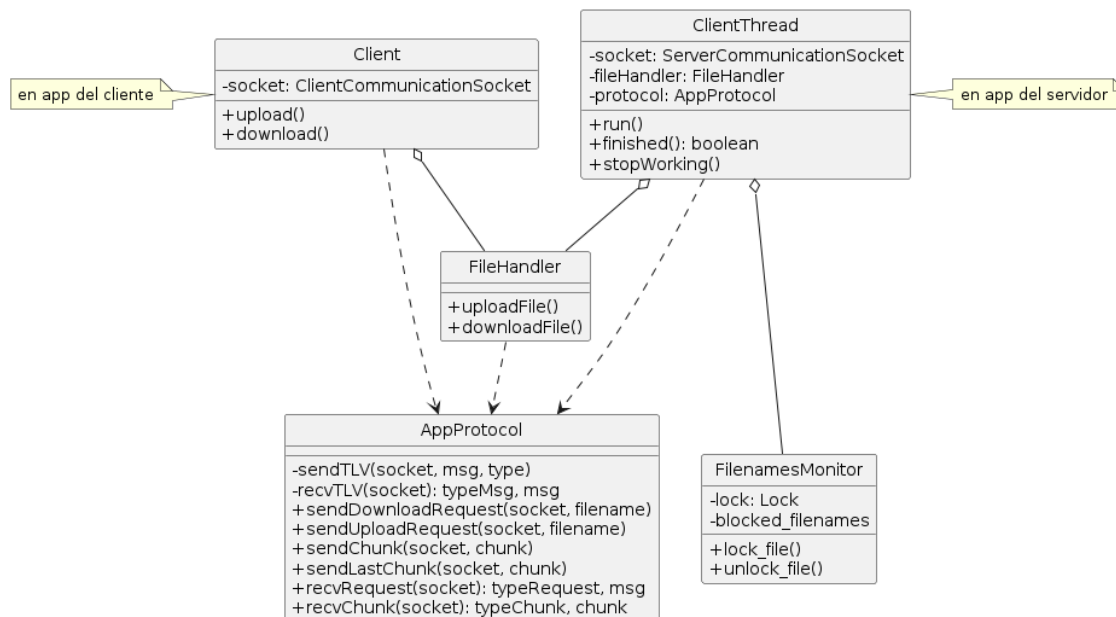


Figura 3: Diagrama de Clases del modelo de File Transfer

3.4. Protocolo TLV

Desde nuestra capa a más alto nivel, decidimos implementar un protocolo binario del tipo TLV (type-length-value).

Todos los métodos de la clase *AppProtocol*, diagramada en la figura 3, implementan este protocolo. Una vez que se genera la tira de bytes correspondiente al paquete, se transfiere el paquete a los *CommunicationSockets*, quienes se encargan de la transferencia confiable de los mismos.

Utilizando el campo de tipo, podemos diferenciar los siguientes mensajes:

- Download Request: Un extremo notifica al otro que quiere descargar un archivo (cuyo nombre estará en el payload del paquete, sino se descartará el request).
- Upload Request: Un extremos notifica al otro que quiere hacer un upload de un archivo (cuyo nombre estará en el payload del paquete, sino se descartará el request).
- Chunk: El extremo que está subiendo un archivo le manda una porción de dicho archivo en un chunk, para que el otro extremo pueda ir escribiendo a medida que se procesa el archivo.
- Last Chunk: El extremo que está subiendo el archivo notifica que es el último chunk que va a mandar, y luego de eso finalizará su ejecución.
- Error message y closed connection: En caso de error o cierre de conexión en un extremo, se notifica al otro de dicho evento.
- Success operation: En caso de que el servidor haya terminado de descargar o subir el archivo, notifica el evento al cliente, con el nombre de destino del archivo.

3.5. Protocolo RDT

En esta sección explicaremos la implementación de nuestra segunda capa de software, que proporciona un protocolo confiable de transferencia de datos. A la vista de la capa superior, la cual explicamos en los anteriores puntos, esta capa está resumida en un conjunto de tres clases: *SocketListener* (quien debe poder aceptar clientes), *CommunicationSocketServer* y *CommunicationSocketClient* (que se comunican utilizando *send* y *recv* por medio de una conexión establecida).

3.5.1. Contención o múltiples sockets UDP abiertos

Una de las primeras dudas que nos planteamos fue: ¿Cómo implementar la recepción y envío de mensajes desde el servidor? En respuesta a esta pregunta obtuvimos dos enfoques principales. Por un lado, que el envío y la recepción de mensajes sea siempre a través del mismo Socket UDP del servidor, y por el otro, que para cada cliente nuevo, se cree un nuevo Socket UDP, con el cual se establece la comunicación con el cliente. A continuación detallamos las ventajas y desventajas de estos diseños, y nuestra decisión en el presente trabajo.

La gran ventaja en *bindear* un nuevo Socket UDP para cada nueva conexión, se ve reflejada en la independencia que tienen los clientes entre si. En otras palabras, si se usa un solo Socket UDP para recibir y enviar los mensajes, entonces los objetos que realicen estas acciones, serán un cuello de botella en la conexión. En cambio, si se abren muchos Sockets UDP, cada uno envía y recibe mensajes por Sockets distintos, y no deben esperar a un *recv* o un *send* de otro cliente, para poder ejecutar su acción. Sin embargo, la desventaja de esta solución es que la cantidad de clientes que podrá recibir el servidor concurrentemente es a lo sumo la cantidad de puertos UDP disponibles en el sistema operativo.

A pesar de que esta última solución presenta ventajas en cuanto a eficiencia decidimos utilizar un único socket UDP para enviar y recibir todos los mensajes. Este enfoque nos brinda ciertas simplificaciones a la hora de implementar la comunicación. Si bien nuestra solución no nos limita en cuanto a la cantidad de clientes que se pueden conectar, a medida que esta cantidad crezca la diferencia en cuanto eficiencia se hará más notoria frente a la otra solución.

3.5.2. Modelado del protocolo

En las figuras 4 y 5 se observan diagramas de clases de ambos lados de nuestro protocolo RDT. Desde el lado del cliente, contamos con las siguientes clases:

- **CommunicationSocketClient**: Nuestro socket al cual el cliente le irá realizando distintas peticiones de intercambio de mensajes.
- **PayloadSender**: Clase encargada de generar un paquete (header + payload) a partir del payload que el cliente quiera mandar. Este paquete es pasado al **PacketSender**.
- **PacketOrchestrator**: Quien inicializa los timers de todos los paquetes que se mandan, notificando al **PayloadSender** que se tiene que realizar una retransmisión en caso de una expiración.
- **PacketSender**: Clase encargada de pasar el paquete a un formato de bytes y mandarlo directamente a través del socket UDP.
- **PacketReceiver**: Un hilo con el socket UDP que está constantemente tratando de recibir. En caso de recibir un paquete en bytes, lo pasa a un formato legible y lo inserta en el buffer del **PayloadReceiver**.
- **PayloadReceiver**: Encargado de obtener el siguiente paquete que el cliente está esperando recibir, y si no se tiene, se esperará a que llegue el mismo. Ésto quiere decir que si los paquetes llegan desordenados (utilizando *Selective Repeat*) y el usuario hace un *recv(size)* se tendrá que esperar hasta obtener el paquete ordenado, aunque hayan elementos en el buffer. De esta manera, si bien se pierde en performance por tener tiempos inactivos aunque hayan paquetes en el buffer para entregarle al usuario, se garantiza que se leerán todos los paquetes en orden.

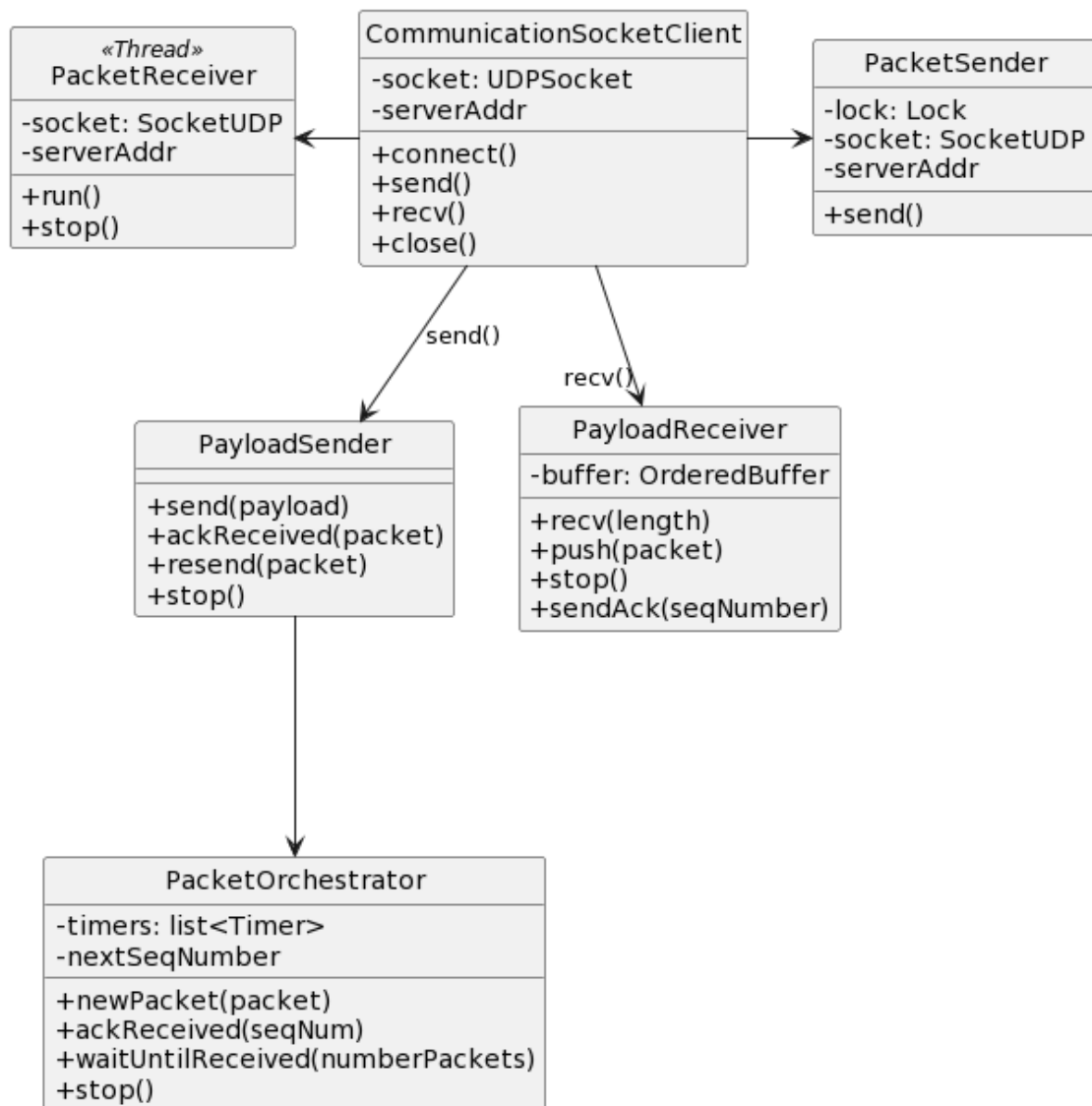


Figura 4: Diagrama de Clases del Protocolo RDT del lado del Cliente

Las clases del lado del servidor:

- **SocketListener**: El welcoming socket que está escuchando nuevos clientes y aceptando nuevas conexiones, creando nuevos **CommunicationSocketServers**.
- **CommunicationSocketServer**: Nuestro socket (ya establecida la conexión) al cual el servidor le irá realizando distintas peticiones de intercambio de mensajes.
- **ClientsMonitor**: Utilizada para agregar, eliminar y obtener clientes entrantes de forma concurrente.
- **PayloadSender**: Clase encargada de generar un paquete (header + payload) a partir del payload que el cliente quiera mandar. Este paquete es pasado al **PacketSender**.
- **PacketOrchestrator**: Quien inicializa los timers de todos los paquetes que se mandan, notificando al **PayloadSender** que se tiene que realizar una retransmisión en caso de una expiración.

- **PacketSender:** Clase encargada de pasar el paquete a un formato de bytes y mandarlo a través del **SenderOrchestrator**.
- **SenderOrchestrator:** Hilo compartido por todos los **CommunicationSocketServer**. Encargado de realizar el envío, por medio del socket UDP del servidor, a la dirección del cliente que el **PacketSender** especifique.
- **PacketReceiver:** Un hilo con una cola bloqueante, quien recibe los paquetes provenientes del **ReceiverOrchestrator**.
En caso de recibir un paquete en bytes en su cola bloqueante, lo pasa a un formato legible y lo inserta en el buffer del **PayloadReceiver**.
Esta clase podría no ser un hilo, pero lo modelamos así, para que no haya demasiada contención en el **ReceiverOrchestrator**.
- **ReceiverOrchestrator:** Hilo compartido por todos los **CommunicationSocketServer**. Encargado de realizar el recibo de datos, a través del socket UDP del servidor, distribuyendo las tiras de bytes a cada *CommunicationSocketServer*.
- **PayloadReceiver:** Encargado de obtener el siguiente paquete que se está esperando recibir, y si no se tiene, se esperará a que llegue el mismo.

Del lado del servidor, si bien difieren en algunas clases respecto del cliente, se puede apreciar una lógica similar y simétrica en lo que respecta al envío y recibo de paquetes con la conexión ya establecida.

Lo que se varía es qué clase es la que posee el socket UDP para enviar o recibir paquetes del otro extremo (Teniendo en cuenta que nuestro servidor maneja múltiples conexiones de distintos clientes, pero todas utilizando el mismo socket UDP).

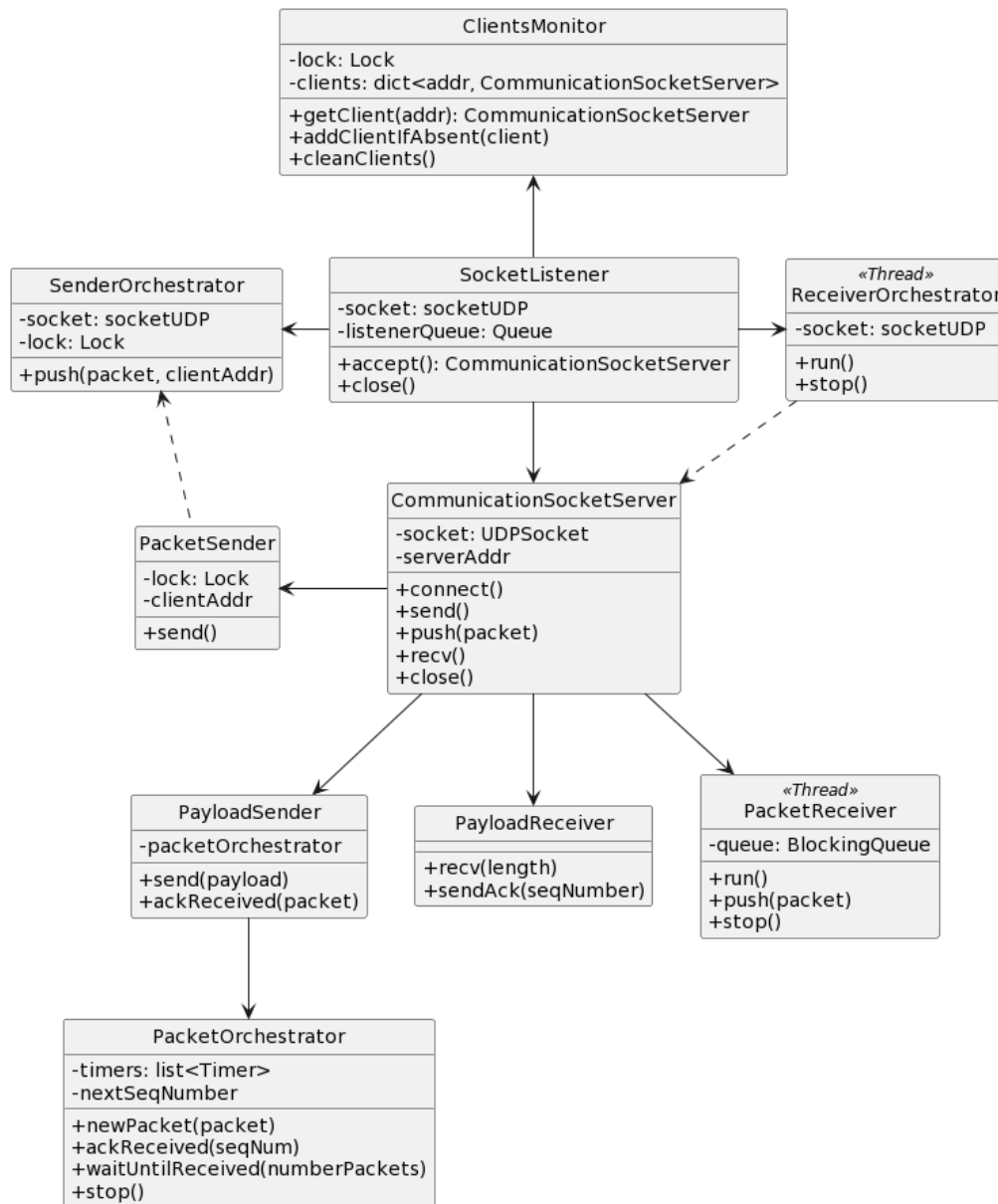


Figura 5: Diagrama de Clases del Protocolo RDT del lado del Servidor

3.5.3. Servicios provistos

El protocolo de la capa de más bajo nivel ofrece como servicios:

- Transferencia confiable de datos.
- Garantía de recibo de paquetes en orden.
- Una conexión entre ambos extremos.

Todo ello es construido sobre los servicios básicos que proporciona UDP.

3.5.4. Especificaciones del protocolo

El orden y formato de mensajes que los extremos se mandan es:

1. Para realizar la conexión: El cliente envía un paquete del tipo SYN y se queda esperando un SYN ACK. Luego de este período, el cliente le puede mandar y esperar recibir paquetes del servidor.
2. Para completar la conexión: El SocketListener recibe un SYN por parte de un cliente nuevo, le envía SYN ACK y crea el CommunicationSocketServer (es decir, la conexión fue establecida). Cabe notar que implementamos un pseudo *three-way-handshake*, puesto que el cliente no envía un ACK luego del SYN ACK. Esto es así porque, al no negociar variables en el establecimiento de la conexión, se tomó como equivalente que el cliente envíe un ACK a que envíe paquetes con carga útil.
3. Para enviar un payload: El extremo debe pasarlo a un formato de bytes, enviar el paquete por el socket UDP y esperar recibo del ACK correspondiente.
En caso de no recibir un ACK luego de cierto tiempo, tiene que volver a reenviar dicho paquete.
4. Para finalizar la conexión: El extremo simplemente se desconecta.
Desde el otro lado se está mandando cada cierto período de inactividad un paquete (sin carga útil) que sirve para chequear el estado de la conexión. Por ello, si luego de ciertos intentos no se pudo obtener ninguna respuesta, se da por finalizada la conexión y se cerrará el extremo faltante.

3.6. Posibles Mejoras

1. Implementación de una suma de comprobación más eficiente: UDP utiliza un checksum que, si no se comprueba, se descarta el paquete.
El problema de este checksum es la simpleza de tener un archivo corrupto y aún así poder comprobar la suma (simplemente haciendo un swap en los bits). Por ello, si se utiliza una herramienta que corrompa los archivos, se debería implementar una suma de comprobación en más dimensiones o un CRC (Cyclic Redundancy Check).
2. Utilización de hilos para la descarga del archivo: Un hilo escriba la primera parte del archivo, mientras otro hilo escriba la segunda.
3. Mejora en el problema de contención explicado en la sección 3.5.1.
4. Garantía de entrega: Hablando de forma estricta, nuestro protocolo RDT no garantiza 100 % la entrega de un mensaje. Puede pasar que la pérdida de paquetes sea muy grande, y por ello uno de los extremos simplemente se desconecte al no poder contactar su contraparte.
Es decir, se tiene para mandar un archivo, pero por un error en la conexión, se cierra la conexión, lo que no le da a un usuario nuestro protocolo RDT la certeza de su funcionamiento. La única desventaja de no implementarlo es no dar garantías absolutas, pero tiene como ventaja que en conexiones realmente inestables o un error que produjo un cierre del otro extremo, se cierre el socket que está comprobando el estado de la conexión.
5. Variar parámetros: El three-way-handshake de TCP le provee una negociación de variables (tamaño máximo de segmento, tamaño de buffers, etc). Al nosotros no implementar ese saludo inicial, nuestros parámetros son constantes, lo que produce un diseño muy estático. Lo más conveniente sería negociar los tamaños de los buffers, la duración de los timers de los paquetes (incluso variarlas según el estado de la red) y cuántas comprobaciones se hacen para chequear si el otro extremo sigue conectado.

4. Pruebas

4.1. Análisis previo a las pruebas

En las siguientes subsecciones, mostraremos diversas pruebas de performance sobre nuestra aplicación, utilizando *Selective Repeat* y *Stop & Wait*.

Esperamos que **Selective Repeat** sea mucho más performante que **Stop & Wait**, puesto que tiene un tamaño de ventana mucho mayor (si bien se espera que llegue el paquete ordenado para entregar, el buffer puede recibir muchos paquetes en lugar de estar inactivo).

Como comentario adicional, definimos de forma constante el tamaño de la ventana de *Selective Repeat* en 32 paquetes sin ACK.

4.2. SR - 20 clientes - 10 % perdida

Probamos 20 clientes concurrentes corriendo al mismo tiempo, solo con Selective Repeat puesto que los tiempos de Stop & Wait eran importantes.

Esto nos proporciona un primer indicio de cuánto performa un protocolo por sobre otro. Prueba con el archivo 'enunciado.pdf' de 255 kB.

```

~/Desktop/asd/TP1-intro-distribuidos/src main !3 14s 11:28:08
time python3 test.py
[INFO]: File 'files/copia_enunciado(30).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(26).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(28).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(26).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(33).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(25).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(24).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(34).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(32).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(31).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(27).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(35).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(29).pdf' has been downloaded
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: Filename uploaded ./server_storage/copia_enunciado(42).pdf
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: Filename uploaded ./server_storage/copia_enunciado(44).pdf
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: File 'files/enunciado.pdf' has been sent to 127.0.0.1:8080
[INFO]: Filename uploaded ./server_storage/copia_enunciado(39).pdf
[INFO]: Filename uploaded ./server_storage/copia_enunciado(40).pdf
[INFO]: Filename uploaded ./server_storage/copia_enunciado(38).pdf
[INFO]: Filename uploaded ./server_storage/copia_enunciado(41).pdf
[INFO]: Filename uploaded ./server_storage/copia_enunciado(43).pdf

-----
Executed in      6.13 secs    fish           external
   usr time     2.09 secs  784.00 micros    2.09 secs
   sys time     1.31 secs  361.00 micros    1.31 secs

```

Figura 6: SR - 20 clientes - 10 % perdida

4.3. SR vs SW - 1 cliente - 10 % perdida

Prueba con el archivo 'libro.pdf' de 18,3 MB.

4.3.1. Selective Repeat

```

~/Desktop/asd/TP1-intro-distribuidos/src main !3 11:30:03
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(1).pdf' has been downloaded

-----
Executed in 12.80 secs  fish      external
usr time   5.62 secs  0.00 millis  5.62 secs
sys time   2.65 secs  1.59 millis  2.65 secs

-----
~/Desktop/asd/TP1-intro-distribuidos/src main !3 12s 11:30:21
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(2).pdf' has been downloaded

-----
Executed in 12.29 secs  fish      external
usr time   5.62 secs  1.17 millis  5.62 secs
sys time   2.62 secs  0.52 millis  2.62 secs

-----
~/Desktop/asd/TP1-intro-distribuidos/src main !3 12s 11:30:37
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(3).pdf' has been downloaded

-----
Executed in 14.02 secs  fish      external
usr time   5.63 secs  663.00 micros  5.63 secs
sys time   2.74 secs  294.00 micros  2.74 secs

```

Figura 7: Selective Repeat - 1 cliente - 10 % perdida

4.3.2. Stop and Wait

```

~/Desktop/asd/TP1-intro-distribuidos/src main !3 3h 0m 9s 14:32:48
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(7).pdf' has been downloaded

-----
Executed in 242.18 secs  fish      external
usr time   11.68 secs  1.08 millis  11.68 secs
sys time   5.33 secs  0.48 millis  5.33 secs

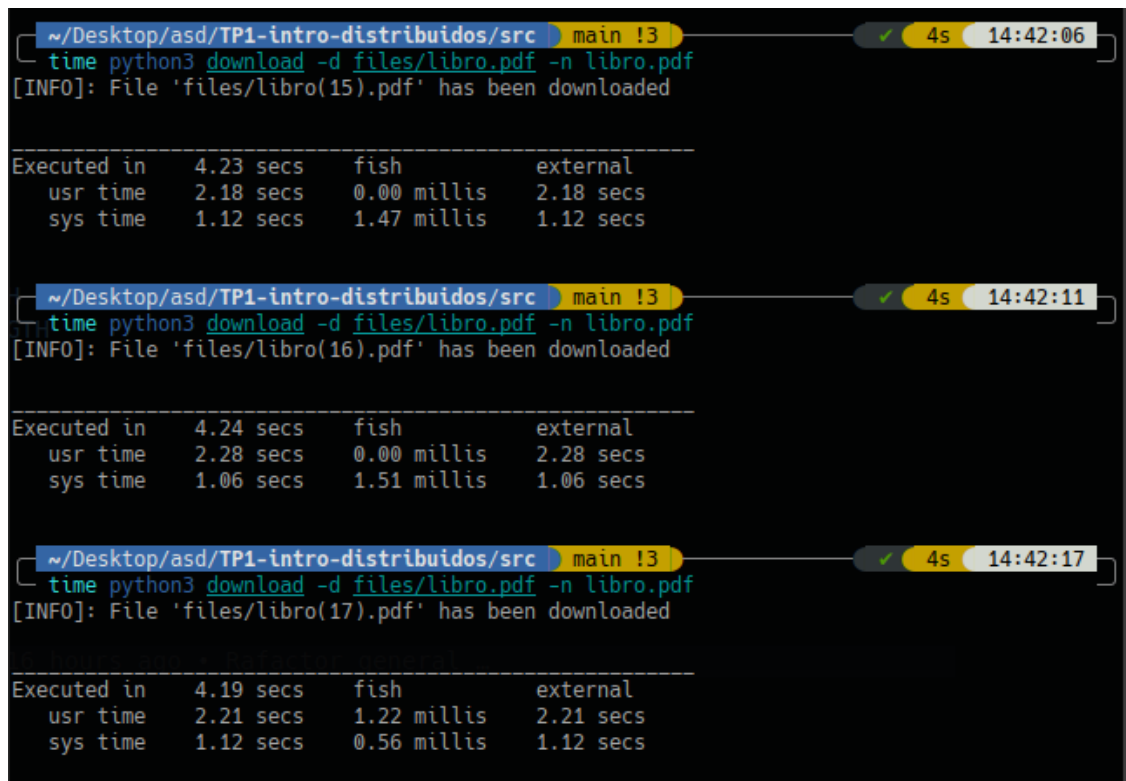
```

Figura 8: Stop and Wait - 1 cliente - 10 % perdida

4.4. SR vs SW - 1 cliente - sin perdida

Prueba con el archivo 'libro.pdf' de 18,3 MB.

4.4.1. Selective Repeat



```
~/Desktop/asd/TP1-intro-distribuidos/src main !3 4s 14:42:06
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(15).pdf' has been downloaded

-----
Executed in    4.23 secs    fish           external
   usr time    2.18 secs    0.00 millis    2.18 secs
   sys time    1.12 secs    1.47 millis    1.12 secs

~/Desktop/asd/TP1-intro-distribuidos/src main !3 4s 14:42:11
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(16).pdf' has been downloaded

-----
Executed in    4.24 secs    fish           external
   usr time    2.28 secs    0.00 millis    2.28 secs
   sys time    1.06 secs    1.51 millis    1.06 secs

~/Desktop/asd/TP1-intro-distribuidos/src main !3 4s 14:42:17
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(17).pdf' has been downloaded

-----
Executed in    4.19 secs    fish           external
   usr time    2.21 secs    1.22 millis    2.21 secs
   sys time    1.12 secs    0.56 millis    1.12 secs
```

Figura 9: Selective Repeat - 1 cliente - sin perdida

4.4.2. Stop and Wait

```

~/Desktop/asd/TP1-intro-distribuidos/src main !3 4m 2s 14:38:06
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(8).pdf' has been downloaded

-----
Executed in 4.79 secs fish external
usr time 2.27 secs 1.62 millis 2.27 secs
sys time 1.05 secs 0.00 millis 1.05 secs

STH
~/Desktop/asd/TP1-intro-distribuidos/src main !3 4s 14:40:16
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(9).pdf' has been downloaded

-----
Executed in 4.73 secs fish external
usr time 2.23 secs 1.68 millis 2.23 secs
sys time 1.10 secs 0.00 millis 1.10 secs

~/Desktop/asd/TP1-intro-distribuidos/src main !3 4s 14:40:28
time python3 download -d files/libro.pdf -n libro.pdf
[INFO]: File 'files/libro(10).pdf' has been downloaded

-----
Executed in 6.89 secs fish external
usr time 3.21 secs 549.00 micros 3.21 secs
sys time 1.39 secs 244.00 micros 1.39 secs

```

Figura 10: Stop and Wait - 1 cliente - sin perdida

4.5. SR vs SW - 1 cliente - 20 % perdida

Prueba con el archivo 'enunciado.pdf' de 255 kB.

4.5.1. Selective Repeat

```
~/Desktop/asd/TP1-intro-distribuidos/src main !3 ✓ 4s 14:44:35
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(1).pdf' has been downloaded

-----
Executed in 495.24 millis fish external
usr time 206.82 millis 485.00 micros 206.33 millis
sys time 50.02 millis 218.00 micros 49.80 millis

~/Desktop/asd/TP1-intro-distribuidos/src main !3 ✓ 14:44:36
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(2).pdf' has been downloaded

-----
Executed in 539.41 millis fish external
usr time 151.07 millis 1.19 millis 149.87 millis
sys time 38.00 millis 0.53 millis 37.47 millis

~/Desktop/asd/TP1-intro-distribuidos/src main !3 ✓ 14:44:40
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(3).pdf' has been downloaded

-----
Executed in 552.25 millis fish external
usr time 201.42 millis 1.17 millis 200.25 millis
sys time 94.51 millis 0.52 millis 94.00 millis
```

Figura 11: Selective Repeat - 1 cliente - 20% perdida

4.5.2. Stop and Wait

```

~/Desktop/asd/TP1-intro-distribuidos/src main !3 14:45:13
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(4).pdf' has been downloaded

-----
Executed in      8.96 secs      fish      external
  usr time    257.54 millis    1.67 millis    255.88 millis
  sys time    127.94 millis    0.00 millis    127.94 millis

~/Desktop/asd/TP1-intro-distribuidos/src main !3 8s 14:45:23
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(5).pdf' has been downloaded

-----
Executed in      7.84 secs      fish      external
  usr time    332.36 millis    1.08 millis    331.28 millis
  sys time     92.02 millis    0.48 millis    91.54 millis

~/Desktop/asd/TP1-intro-distribuidos/src main !3 7s 14:45:31
time python3 download -d files/enunciado.pdf -n enunciado.pdf
[INFO]: File 'files/enunciado(6).pdf' has been downloaded

-----
Executed in      6.94 secs      fish      external
  usr time    278.07 millis    1.11 millis    276.96 millis
  sys time     99.41 millis    0.49 millis    98.92 millis

```

Figura 12: Stop and Wait - 1 cliente - 20 % perdida

4.6. SR vs SW - 3 clientes - 10 % perdida

Prueba con el archivo 'enunciado.pdf' de 255 kB.

4.6.1. Selective Repeat

```

~/Desktop/asd/TP1-intro-distribuidos/src main !2 6s 14:52:08
time python3 test.py
[INFO]: File 'files/copia_enunciado(15).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(16).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(17).pdf' has been downloaded

-----
Executed in      1.47 secs      fish      external
  usr time    184.53 millis    0.00 millis    184.53 millis
  sys time     75.57 millis    1.76 millis    73.81 millis

```

Figura 13: Selective Repeat - 3 clientes - 10 % perdida

4.6.2. Stop and Wait

```

~/Desktop/asd/TP1-intro-distribuidos/src main !2
time python3 test.py
[INFO]: File 'files/copia_enunciado(20).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(22).pdf' has been downloaded
[INFO]: File 'files/copia_enunciado(21).pdf' has been downloaded

-----
Executed in    5.08 secs    fish           external
   usr time   578.50 millis   1.15 millis   577.35 millis
   sys time   283.04 millis   0.51 millis   282.53 millis

```

Figura 14: Stop and Wait - 3 clientes - 10 % perdida

4.7. Análisis de los resultados obtenidos

Podemos comprobar realmente que **Selective Repeat** **performa en tiempos mucho menores que Stop & Wait**, sobre todo en caso de pérdidas de paquetes.

Lo que corresponde exactamente con lo que esperábamos, puesto que nuestro buffer (aunque se esté esperando por el paquete ordenado) está activo recibiendo paquetes y devolviendo ACK de los paquetes que sí puede insertar.

5. Preguntas a responder

1. Describa la arquitectura Cliente-Servidor.

Esta arquitectura cuenta con dos hosts:

- Un cliente: Inicia contacto con el servidor. Normalmente, solicita un servicio del servidor.
- Un servidor: Soporta múltiples clientes y proporciona el servicio al cliente que lo solicite.

Ambas aplicaciones están conectadas lógicamente a partir de su capa de transporte.

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

Un protocolo de capa de aplicación está encargado de definir cómo es el procedimiento del envío y recibo de mensajes entre dos aplicaciones que quieren intercambiar información, además de definir la sintaxis y semántica de dichos mensajes.

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

Nuestro protocolo de aplicación consiste en las dos capas de software previamente descritas: Un protocolo TLV sobre el que corre nuestra aplicación de traspaso de archivo, que proporciona la forma en que se solicitan recursos del servidor, y cómo este le responde al cliente.

Luego, un protocolo RDT que proporciona la forma en que se intercambian mensajes directamente aplicación a aplicación, garantizando que si se manda un mensaje, llegará al otro extremo.

Esta es una descripción en forma breve, el cómo funciona el protocolo de forma específica esta completamente desarrollado a lo largo del informe, con hincapié en la sección 3.

4. 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?

UDP: Es un protocolo de capa de transporte que no proporciona ninguna garantía acerca de la entrega del paquete.

Lo único que provee es un servicio de multiplexación y demultiplexación (identificando el socket inequívocamente según la dirección IP y puerto de destino), y una suma de comprobación que, en caso de fallar, simplemente descarta el paquete.

Es apropiado utilizar UDP cuando:

- Se quiere tener un mejor control sobre qué datos se envían y cuándo: Cuando tiene un segmento disponible lo manda puesto que no utiliza métodos de control de flujo y de congestión.
- Menor memoria: Los headers UDP son mucho más pequeños en tamaño que los headers TCP (lo que provoca que los routers tengan un menor tiempo de procesamiento). A su vez, al no ser un protocolo RDT, no guarda estados de la conexión, lo que provoca menor consumo de memoria en el host.

TCP: Es un protocolo que provee una garantía de entrega de los paquetes (Reliable Data Transfer, RDT) y full-duplex (es decir, un canal bidireccional para mandar datos). Ofrece un servicio de multiplexación y demultiplexación (identificando el socket inequívocamente según la dirección de IP y puerto tanto de origen como de destino), la misma suma de comprobación que UDP y, lo que principalmente se agrega además del servicio RDT: un control de flujo (para no saturar al host receptor), un control de congestión de la red (para no saturar la red) y una orientación a la conexión.

Es apropiado utilizar TCP cuando:

- Interesa la equidad en la red: Es muy importante no saturar la red de paquetes puesto que se perjudica tanto la conexión entre las aplicaciones, como todo el envío y recibo de paquetes de la red. El mecanismo de control de congestión de TCP nos previene de estos casos.
- Se requiere que los paquetes lleguen al otro host: Por ejemplo, en aplicaciones de File Transfer, donde no importa el tiempo que se tarde tanto como que el mensaje llegue correctamente al destino.

Ninguno de los dos ofrecen servicios como ancho de banda mínimo, un tiempo mínimo para que lleguen los paquetes o seguridad.

6. Dificultades encontradas

6.1. Análisis de critical sections

Puesto que contamos con distintos hilos y recursos compartidos, tuvimos que hacer un análisis profundo de cuáles son las critical sections. Una vez que las identificamos, debimos separar los recursos compartidos en monitores, para generar un código con poca contención y sin la posibilidad de race conditions. Sin embargo, esta estructura fue difícil, principalmente en la implementación del protocolo RDT.

En un principio priorizamos la funcionalidad antes que un código limpio y con buen diseño, lo que condujo a la aparición de algunos bugs. Estos problemas nos llevaron a múltiples refactors y arreglo de errores, luego de realizar varios tests.

6.2. Tests de casos bordes

Al contar con una pérdida de archivos aleatoria, ciertos casos bordes aparecían en algunas ejecuciones y tuvimos que esperar varias ejecuciones más para comprobar si solucionamos ese problema o no.

Una solución fue hacer un test que abra múltiples hilos, haga un upload o download de un archivo de forma aleatoria, y luego nosotros comprobamos el resultado de todos los archivos.

6.3. Finalización de la comunicación

Quizás uno de nuestros mayores problemas: Cómo cerrar un extremo de forma correcta, teniendo en cuenta que implementamos una conexión similar a TCP.

Por dicha similitud, comenzamos implementando mensajes de FIN, lo que nos llevó a un diseño caótico y con problemas de concurrencia.

Encontramos que el núcleo del problema del fin de la comunicación viene dado cuando uno de los extremos está inactivo esperando recibir paquetes del otro lado. Por ello, eliminamos cualquier mensaje de FIN, y lo que hacemos es, si se está en un período de inactividad, comprobar el estado de la conexión: Si se obtiene respuesta de ese estado o simplemente se recibe un paquete, quiere decir que la conexión está estable; si no se obtiene respuesta luego de cierta cantidad de intentos (definida de forma constante) se da por pérdida la conexión y se desconecta ese extremo.

Así, cuando ocurre un cierre repentino (se hace un 'q' o un 'ctrl+c' en la terminal), ese extremo se cierra eliminando todos los timers de sus paquetes.

El otro lado puede estar esperando recibir paquetes de ese extremo, por ello luego de intentar ver el estado de la conexión, se dará por finalizada.

Si ese otro lado está intentando mandarle paquetes al extremo cerrado, al estar esperando el ACK de los paquetes, se dará por finalizada la conexión luego de cierta cantidad de intentos de reenviar los paquetes expirados.

7. Conclusión

7.1. Conclusión sobre los protocolos

Teóricamente (en el libro de la materia) encontramos una forma simple de implementar un protocolo de comunicación (sobre todo RDT). En el momento de pasar a la práctica, pudimos encontrar ciertas diferencias entre un modelo teórico y problemas de la vida real:

El producir un protocolo a nivel software implica más que definir cómo es el flujo de mensajes de nuestra aplicación. También involucra definir semántica de cómo se reciben los mensajes, cómo se parsean, cómo se reciben y procesan dichos mensajes en ambos extremos.

Otro desafío que pudimos notar en nuestro protocolo RDT es cómo sincronizar ambos extremos para que se establezca y luego se cierre la conexión. Un modelo teórico no realiza análisis de critical sections o de distintos hilos que un servidor puede soportar.

Para finalizar esta subsección, podemos decir que nuestro protocolo RDT no garantiza 100% la entrega de un mensaje a quien lo use: Si el estado de la conexión de la red no es buena u ocurrió un error en el otro extremo, se procederá a cerrar la conexión.

Esto es algo similar a lo que implementa TCP, es decir, podemos afirmar que TCP tampoco proporciona una garantía segura de entrega.

El realizar una garantía segura de entrega podrá ser posible en un modelo teórico, pero no en un modelo donde el otro extremo se puede desconectar, o el estado de la conexión de internet puede ser realmente muy pobre. Así, un modelo que, **mientras no haya errores** garantice el recibo de los mensajes en el otro extremo, está mejor preparado para el estado actual de internet.

7.2. Performance y desacople entre las capas

Al desacoplar la aplicación en dos capas de software bien diferenciadas, encontramos la desventaja de tener una gran contención del lado del servidor, es decir, tenemos un cuello de botella con el intercambio mensajes entre el servidor y múltiples clientes.

Podemos concluir que, si bien una mejora sería un modelo híbrido entre un desacople (que permita abstraernos de cualquier aplicación y ofrecer este servicio a cualquier otra capa de software que se quiera comunicar con otra aplicación) y un modelo con menor contención produce un coste muy alto en diseño, puesto que se pueden perder ambas características al unificar.

Preferimos un diseño separado, más simple de testear de forma separada, con responsabilidades bien marcadas para cada clase creada, con una performance peor para múltiples usuarios concurrentes (sobre todo cuando la cantidad de los mismos es realmente importante).

7.3. Conclusión general del trabajo realizado

Para finalizar, el trabajo realizado nos proporcionó un mejor entendimiento sobre los desafíos que implica hacer un protocolo: Se puede entender, luego de tener problemas de sincronización para iniciar y terminar una conversación, por qué TCP implementa el three-way-handshake y el 2-2 FIN; a su vez, qué beneficios le da tener una conexión activa (con tamaños de buffers negociados, seteo de parámetros variables, entre otras cuestiones); y, sobre todo, los diferentes casos bordes (como qué pasa cuando una conexión lleva mucho tiempo inactiva o el estado de la red es pobre) que nos sirvieron para **entender la interacción entre la capa de aplicación y la interfaz de sockets** desde un punto de vista de más bajo nivel a como lo veníamos viendo en la carrera.