.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo Práctico 2 — File Transfer

[75.43/75.33/95.60] Introducción a los Sistemas Distribuidos
Primer cuatrimestre de 2022

Alumno	**Padrón**	**e-mail**
Mauricio Buzzone	103783	mbuzzone@fi.uba.ar
Lukas De Angelis Riva	103784	ldeangelis@fi.uba.ar
Kailásh Aquista	104505	kaquista@fi.uba.ar
Matías Merlo González	104093	mmerlog@fi.uba.ar
Ramiro Javier Sánchez	104095	rsanchez@fi.uba.ar

Índice

1. Introducción **2**

2. Hipótesis y supuestos **2**
2.1. El protocolo Selective Repeat, realiza como máximo 18 retransmisiones por paquete 2
2.2. Ambos protocolos esperan como máximo 18 timeouts 2
2.3. Si se hace Upload de un archivo que ya existe en el servidor, se sobrescribe 2
2.4. Si algún cliente intenta hacer un Upload/Download de algún archivo entonces ningún otro cliente podrá hacer un Upload/Download del mismo archivo hasta que termine el anterior. 3

3. Implementación **3**
3.1. FileTransfer 3
3.2. Socket RDT con Stop and wait 3
3.2.1. Send() 4
3.2.2. Recv() 4
3.3. Selective Repeat 4
3.3.1. Send() 4
3.3.2. Recv() 4
3.4. RDTPacket 4

4. Pruebas **5**
4.1. Comparativa 5

5. Preguntas a responder **6**
5.1. Describa la arquitectura Cliente-Servidor 6
5.2. ¿Cuál es la función de un protocolo de capa de aplicación? 6
5.3. Detalle el protocolo de aplicación desarrollado en este trabajo 6
5.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP 6
5.4.1. ¿Qué servicios proveen dichos protocolos? 6
5.4.2. ¿Cuáles son sus características? 6
5.4.3. ¿Cuándo es apropiado utilizar cada uno? 7

6. Dificultades encontradas **7**

7. Conclusión **7**

8. Referencias **8**

1. Introducción

El presente informe reúne la documentación del segundo trabajo práctico de la materia Introducción a los Sistemas Distribuidos. El objetivo es implementar un protocolo RDT (Reliable Data Transfer) sobre el protocolo de entrega no confiable UDP. Vamos a desarrollar dos diferentes implementaciones del protocolo RDT, la primera será con un mecanismo stop-and-wait y la segunda con Selective Repeat.

Para realizar un protocolo RDT debemos tener en cuenta los siguientes 3 aspectos:

- Garantizar la entrega de todos los paquetes.
- Mantener el orden de los paquetes.
- Asegurar la integridad de los datos.

2. Hipótesis y supuestos

2.1. El protocolo Selective Repeat, realiza como máximo 18 retransmisiones por paquete

Tuvimos la necesidad de agregar una cantidad máxima de retransmisiones para un paquete debido a que se puede dar el caso donde el receptor de la conexión se cierra inesperadamente y el emisor se queda retransmitiendo indefinidamente.

Decidimos 18 retransmisiones como máximo, ya que es la cantidad mínima de intentos necesarios para que la probabilidad de que el paquete no se pierda definitivamente sea mayor a 0.95, en una red con 60 % de packet loss.

Primero calculamos la probabilidad de que se pierda un paquete o su ACK en una red con 60 % de packet loss con un arból de probabilidades:

$$P(Retransmision) = 0{,}6 + 0{,}4 \cdot 0{,}6 = 0{,}84$$

Notar que lo calculamos como la probabilidad de que el paquete se pierda a la ida sumado a la probabilidad de que el ACK se pierda a la vuelta.

Luego calculamos el cuantil 0.95 de una distribución geométrica (que modela la cantidad de intentos hasta el primer éxito), con un parámetro de $p = 0{,}16$, ya que es la probabilidad de que el paquete se envíe correctamente. El resultado es el número 18.

En el caso extremo de tener un packet loss de 60 % en la red, podemos asegurar con un 0.95 de probabilidad que cuando se pierden 18 retransmisiones se debe a que se perdió la conexión.

2.2. Ambos protocolos esperan como máximo 18 timeouts

Debido al análisis realizado en el supuesto anterior, el emisor esperará el ACK de un paquete a lo sumo 18 timeouts. En caso de no haber recibido la confirmación dentro de ese lapso, asumimos que el receptor tuvo un cierre inesperado.

2.3. Si se hace Upload de un archivo que ya existe en el servidor, se sobrescribe

En caso de que algún cliente quiera subir un archivo que ya existe en el servidor, dado que ira a parar al mismo directorio dentro del servidor decidimos que se sobrescriba el archivo.

2.4. Si algún cliente intenta hacer un Upload/Download de algún archivo entonces ningún otro cliente podrá hacer un Upload/Download del mismo archivo hasta que termine el anterior.

Si un cliente intenta descargar/cargar un archivo mientras otro lo está leyendo o modificando, se le informará que no es posible acceder al contenido. Esto no ataca al espíritu del trabajo práctico, y facilita la implementación del file transfer.

3. Implementación

3.1. FileTransfer

Implementamos una clase FileTransfer que se encarga de hacer el envío y la recepción de un archivo. Esta clase se encarga de leer los archivos de memoria y empezar a enviarlos a través del socket generado en la conexión. También se encarga de recibir los paquetes a través del socket y guardarlos en un archivo local. En caso de algún error al abrir archivos, el FileTransfer mandara el error correspondiente. Notar que esta clase no realiza lógica RDT, sino que delega dicha responsabilidad sobre el socket que está utilizando.

3.2. Socket RDT con Stop and wait

Para la implementación de Stop and Wait, utilizamos una clase llamada **RDTSocketSW**. El objetivo fue implementar una interfaz similar a la que utiliza un socket **TCP** de la biblioteca estándar. Es decir, que implementamos nuestro propio *listen()*, *accept()*, *send()* y *recv()*.

Internamente, RDTSocketSW cuenta con un socket UDP que se utilizará para enviar y recibir paquetes.

Cuando creamos un RDTSocketSW y ejecutamos el método *listen()*, el socket creará un hilo donde estaremos esperando peticiones de conexiones de clientes. Al recibir un paquete SYN de un cliente nuevo, crearemos un RDTSocketSW específico para la conexión con dicho cliente, al igual que lo hace TCP. Ver en la figura 1.

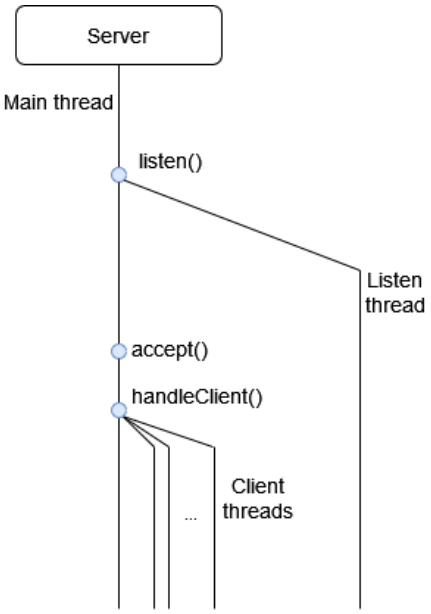

Figura 1: Diagrama de ejecución del servidor

3.2.1. Send()

El método send() de stop and wait se encarga de enviar un paquete, y esperar el ACK del mismo. Es por esto que la función es bloqueante y no nos permitirá enviar otro paquete hasta haber recibido la confirmación del receptor.

3.2.2. Recv()

El método recv() de stop and wait también es una función bloqueante. Dentro de esta función vamos a estar esperando un paquete cuyo **sequence number** coincida con nuestro **acknowledge number**. Si no se cumple la condición anterior, descartamos el paquete ya que correspondería a uno que llego en orden incorrecto o que se debe a una retransmisión.

3.3. Selective Repeat

Para la implementación del protocolo Selective Repeat creamos la clase RDTSocketSR, donde al igual que RDTSocketSW, implementa una interfaz similar a la de un socket TCP, pero sin implementar la abstracción de STREAM de datos. Tendremos dos estructuras de datos principales:

- InputBuffer: se encargará de guardar todos los paquetes recibidos.
- OutPutWindow: se encargará de guardar todos los paquetes que fueron enviados pero aún no confirmados.

Dentro de un RDTSocketSR vamos a tener un hilo receptor, que va a estar esperando por paquetes y los insertará en un input buffer. Además, cuando recibe un ACK, actualiza la output window. Por último se encarga de cerrar el socket cuando recibe un paquete FIN.

Cabe aclarar que decidimos utilizar una ventana de 10 paquetes en los sockets de selective repeat de modo que la transferencia de los archivos de prueba no sea instantánea.

3.3.1. Send()

El método send() de Selective Repeat, a diferencia del send() de Stop and Wait, **NO es bloqueante**. Esta función va a enviar un paquete (en caso de que la output window no esté llena) y comenzará un *Timer* en un hilo aparte, cuando este termina, verificará de que se haya confirmado el paquete enviado, y si no es el caso, lo reenvía (comenzando otro Timer, recursivamente)

3.3.2. Recv()

Dado que tenemos un hilo recibidor, que despacha los paquetes recibidos a un input buffer, el recv() se quedará esperando hasta que encuentre el paquete con el Sequence Number esperado en el Input Buffer. De esta forma nos aseguramos de que los paquetes lleguen en el orden correspondiente. Cabe aclarar que una vez pasada cierta cantidad de tiempo, se levantará la excepción de LostConnection, asumiendo que la otra parte de la conexión se ha cerrado inesperadamente. Este tiempo fue calculado en base al tiempo máximo que tardaría un paquete en ser retransmitido la máxima cantidad de veces desde el cliente.

3.4. RDTPacket

Para la transmisión de información entre sockets, creamos un RDTPacket, que contiene información necesaria para llevar a cabo las responsabilidades del protocolo confiable, y el payload a transferir.

Un RDTPacket contiene la siguiente información:

- Las siguientes flags: **SYN**, **FIN** y **ACK**
- Sequence number

- Acknowledge number
- Checksum
- Payload (en formato de bytes)

Como ya sabemos, para enviar estos RDTPackets, necesitamos hacerlo a través de un socket UDP. Estos sockets reciben un arreglo de bytes, por lo tanto, implementamos un método en la clase RDTPacket para serializar los campos descritos anteriormente. Esta función convierte todo el paquete a un arreglo de byte, de modo que podamos mandarlo con el socket UDP.

Para la serialización de RDTPacket utilizamos al biblioteca **struct** ya que nos permite enviar integers con tamaño fijo, como lo hacen C y C++.

Cabe mencionar que sobre esta interfaz de RDTPacket, se construye otra clase llamada Packet, que define el formato de mensajes que intercambian los clientes con el servidor. Se puede encontrar más información sobre el formato de los mensajes en la sección 5.3

4. Pruebas

Realizamos una serie de pruebas donde se describe el procedimiento junto al resultado esperado en el archivo *test.md*, listamos los títulos aquí:

```
## TESTS:
### Test 1: Subir archivo chico
### Test 2: Subir un archivo grande
### Test 3: Sobrescribir un archivo que ya existe
### Test 4: descargar archivo
### Test 5: descargar un archivo inexistente
### Test 6: Descargar simultáneamente diferentes archivos
### Test 7: Descargar simultáneamente el mismo archivo
### Test 8: Cargar simultáneamente hacia diferentes paths
### Test 9: Cargar simultáneamente hacia mismo path
### Test10: Cargar y subir diferentes archivos en simultaneo
### Test11: Cargar y subir el mismo archivo
```

4.1. Comparativa

Para comparar los protocolos de **Stop and Wait** y **Selective Repeat** utilizamos Hamachi para poder transferir un archivo a través de internet. En un host tenemos corriendo el servidor, mientras que en otro ejecutamos los scripts upload.py/download.py.

El primer caso que probamos fue el de subir un archivo de 5Mb al servidor con **Selective Repeat** (Modificamos el tamaño del output window a 1000 ya que por default usamos una ventana de 10 para realizar pruebas locales). A continuación podemos observar el tiempo de transferencia:

```
2022/06/03 12:32:51 PM [DEBUG]: Finished uploading the file in 2717ms
```

A continuación subimos el mismo archivo al servidor, pero en esta ocasión con el protocolo **Stop and Wait**:

```
2022/06/03 12:27:51 PM [DEBUG]: Finished uploading the file in 166722ms
```

Como podemos observar, la diferencia entre los tiempos de transferencia es inmensa, el protocolo Selective Repeat es mucho más rápido que el de Stop and Wait.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

La arquitectura Cliente-Servidor se caracteriza por tener un *host* llamado **servidor** que recibe peticiones y otros *host* llamados **clientes** que son los que mandan las peticiones y las responden. En esta arquitectura los clientes **no** se comunican entre sí (como ocurre en la arquitectura *peer to peer*) y el servidor suele tener una dirección IP fija.

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

Un protocolo de la capa de aplicación tiene que definir los **tipos de mensajes** que se mandan en la misma (petición y respuesta) al igual que la **sintaxis** de los mensajes (los campos del mensaje y como se delimitan) y su **semántica** (el significado de la información en cada campo). De la misma forma el protocolo tiene que definir **cuándo y cómo se envían y responden** esos mensajes.

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

Para iniciar la conexión entre cliente y servidor se manda un mensaje query en cuyos campos se incluye la siguiente información: *Tipo de mensaje*, *tamaño del archivo* y *nombre del archivo*.

Los tipos de mensaje pueden ser:

- **Download**: Se está solicitando una descarga del archivo especificado en file_name.
- **Upload**: Se está solicitando una carga de un archivo de tamaño file_size.
- **Error**: Cuando no se pudo abrir el archivo solicitado en el servidor.
- **Busy**: El archivo solicitado no está disponible.
- **OK**: El archivo solicitado está disponible para la descarga.

El cliente envía el query aclarando si quiere iniciar la descarga o carga. Luego el servidor contesta diciendo si hubo un error, si el archivo no está disponible o si el archivo esta disponible para descarga. En caso de estar disponible se inicia la transferencia del archivo utilizando los sockets RDT.

5.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP

5.4.1. ¿Qué servicios proveen dichos protocolos?

Tanto TCP como UDP proveen servicios de control de integridad (es decir asegurarse de que el mensaje no fue corrompido) y de multiplexación/demultiplexación que permiten la comunicacion entre distintos procesos en *hosts* distintos. Adicionalmente TCP incluye los servicios de entrega confiable (asegurarse que todos los paquetes lleguen y sea en orden) y control de congestión (acomodar el envío de mensajes para no saturar la red)

5.4.2. ¿Cuáles son sus características?

UDP es un protocolo no orientado a la conexión. No genera una conexion con los clientes para mandarles mensajes, sino que simplemente manda los mensajes y no se preocupa por si llegan o no. Dado que no se genera una conexión con los clientes, UDP nos sirve para hacer broadcast. Además un socket (la API para usar estos protocolos de transporte) UDP queda definido solamente por la dirección IP destino y puerto destino. Por otra parte el protocolo TCP está orientado a la conexión, esta se genera mediante un procesos conocido como **three way handshake**. Las

conexiones se generan entre 2 *hosts* y nada mas, lo que quiere decir que este protocolo **no permite hacer broadcast.** Si un cliente quisiera mandar mensajes a muchos clientes, deberá mandarlo a través de cada conexión individual. Además un socket TCP queda definido por la tupla compuesta por 4 componentes: las direcciones IP de origen/destino y los puertos origen/destino. Estos 4 componentes nos permiten identificar unívocamente esa conexión entre procesos.

5.4.3. ¿Cuándo es apropiado utilizar cada uno?

El uso del protocolo TCP es apropiado en aquellas ocasiones donde es necesario asegurar la entrega confiable de la información, como puede ser conectarse a una página web (el protocolo HTTP utiliza TCP) ya que no nos gustaría que una pagina web cargue de forma incompleta.

En cambio el protocolo UDP es más ligero (su header tiene longitud menor al de TCP), se puede usar en aplicaciones de tiempo real o que sean tolerantes a perdidas como lo puede ser una aplicación de streaming. UDP es más minimalista que TCP, lo que da espacio para que se implementen más servicios en la capa de aplicación como por ejemplo el protocolo QUIC que implementa reliable data transfer sobre UDP sin las restricciones del congestion control que tiene TCP. UDP no tiene el three way handshake, por lo que no tiene ese *delay* asociado, una de las razones por las que se usa en el protocolo DNS (ademas de la necesidad de hacer broadcast). Tambien se tiene mas control sobre la información que se envía. Por último, al no ser orientado a la conexión como TCP, gasta menos recursos asociados a eso como pueden ser los receive y send buffers, parametros de control de congestión y parámetros de números de secuencia y ACK. Al ahorrarse estos recursos, un servidor podría llegar a ser capaz de encargarse de mas clientes cuando corre en UDP que en TCP.

6. Dificultades encontradas

La primera dificultad encontrada fue el desarrollo de la función *listen()* del socket RDT ya que, si bien sabíamos de que el servidor debía crear un socket nuevo cuando un cliente contactaba al servidor, no estábamos seguros si los sockets nuevos tendrían todos el mismo puerto de origen. Ante la imposibilidad de distinguir sockets UDP por 4 atributos (ip/puerto origen, ip/puerto destino), decidimos que a cada cliente se le asigne un puerto distinto. De esta forma un servidor que utilice un socket RDT soportará como máximo 65535 conexiones.

Otro inconveniente que tuvimos fue durante el desarrollo de la función close() de los socket RDT. Nuestro objetivo era simular el cierre de conexión de TCP donde ambos lados de la comunicación mandan un paquete FIN y se espera al ACK, sin embargo no pudimos hacerlo ya que habían muchos casos borde que no supimos resolver. Terminamos implementando un "two way handshake", con algunos casos borde donde se culmina la conexión frente a un timeout.

Por último, uno de los mayores inconvenientes encontrados a lo largo de todo el trabajo práctico fue la correcta utilización de la biblioteca threading, mayormente debido a las race-conditions.

7. Conclusión

Podemos concluir, como era de esperarse, que el protocolo implementado con **Selective Repeat** es considerablemente mas rápido que la versión implementada con **Stop and Wait**. Esto es debido a que el segundo debe esperar al menos un RTT por cada paquete enviado y recién ahí enviar el siguiente, mientras que **Selective Repeat** (con una ventana lo suficientemente grande) elimina ese delay.

A favor del protocolo **Stop and Wait**, podemos decir que su implementación es más simple que el **Selective Repeat** debido a que hay muchos menos problemas derivados de su comportamiento concurrente.

Por último, el trabajo práctico nos sirvió para afianzar y poner en práctica los conocimientos teóricos que aprendimos en la materia.

8. Referencias

- https://docs.python.org/3/library/socket.html
- Computer Networking: A top down approach
- Repositorio de GitHub