FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

[75.43] Introducción a los sistemas distribuidos
Primer cuatrimestre de 2022

Fabbiano, Fernando	102464	ffabbiano@fi.uba.ar
Nocetti, Tomas	100853	tnocetti@fi.uba.ar
Alasino, Franco	102165	falasino@fi.uba.ar
Sportelli Castro, Luciano	99565	lsportelli@fi.uba.ar
Ganopolsky, Damian	101168	dganopolsky@fi.uba.ar

Índice

1. Introducción

El objetivo final del trabajo es hacer que el protocolo UDP sea confiable, implementando el servicio de Reliable Data Transfer (RDT). Por definición, UDP no provee dicho servicio, al contrario de TCP.

Para implementar la entrega confiable, se pide que implementemos los protocolos de Stop Wait, y de Selective Repeat.

El primero de ellos, consiste en enviar de a un paquete por vez. El remitente se queda esperando una señal de ACK para poder enviar al destinatario otro paquete. Esto hace que la transmisión sea más lenta.

Por otro lado, en Selective Repeat, se envían ráfagas de paquetes del tamaño máximo que indique la receive window del destinatario. Estos, son almacenados en un buffer en el remitente, que va a solicitar al destinatario el reenvio de aquellos paquetes de los cuáles el no recibió un ACK.

En ambos casos nos aseguramos que llegue la totalidad de los paquetes, lo cuál es el objetivo de este trabajo.

2. Hipótesis y soluciones realizadas

Para el trabajo se realizaron las siguientes hipótesis:

- Cuando un cliente decide subir un archivo y ya existe uno con el mismo nombre el archivo se sobre escribe.
- El cliente no tiene manera 'polite' de parar la descarga. Debe detener el programa.
- Un paquete se puede reenviar como máximo 3 veces, en caso de superar este valor, se considera la conexión como finalizada
- Debe pasar 1 segundo para que un paquete se considere perdido

3. Implementación

Para el desarrollo del trabajo se decidió partir lo que fue el diseño de la solución en dos partes fundamentales. Una capa de aplicación encargada de gestionar toda la funcionalidad del FileTransfer y una capa de transporte encargada de asegurar la comunicación bajo las premisas del enunciado. El objetivo del desarrollo a traves de esta modalidad fue abstraer a la capa de aplicación que manejaba el FileTransfer sobre que tipo de protocolo de transporte funcionaba por debajo. Así mismo permitió paralelizar el trabajo de manera tal que los integrantes enfocados en el desarrollo de la aplicación perse pudiesen avanzar sin tener una capa de transporte terminada.

3.1. Capa de Aplicación

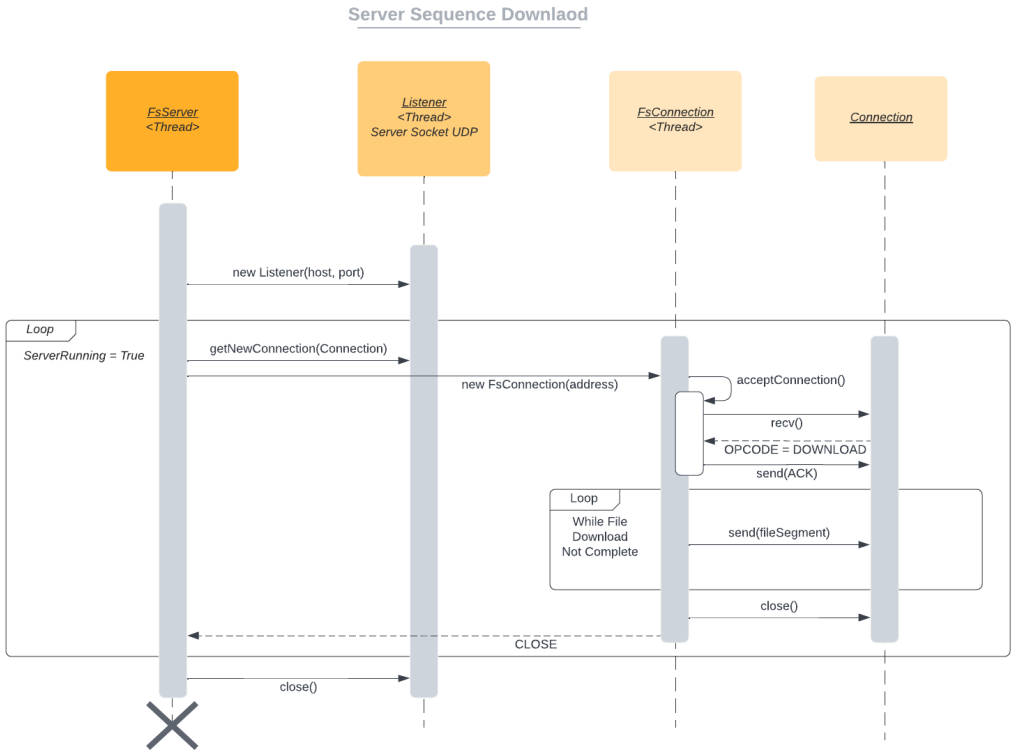
3.1.1. Servidor

El servidor maneja las siguientes entidades fundamentales:

- FsServer: mantiene el listado de conexiones activas y es el responsable de gestionar y lanzar nuevos hilos para conexiones de clientes entrantes.
- FsConnection: se genera una instancia por cliente dentro de la cual se resuelve la operación iniciada, ya sea 'UPLOAD' o 'DOWNLOAD'. Toda la comunicación de la capa de aplicación con el cliente queda encapsualada en esta clase.
- Listener: *desde un punto de vista de aplicación* es la entidad encargada de recibir nuevas conexiones entrantes y esperar a que el FsServer las proceses.

- Connection: es el wrapper a nivel aplicación que me brinda la api de comunicación con el cliente

A continuación se presenta un diagrama de secuencia de una operación de download del lado del servidor:



La operación de upload es equivalente.

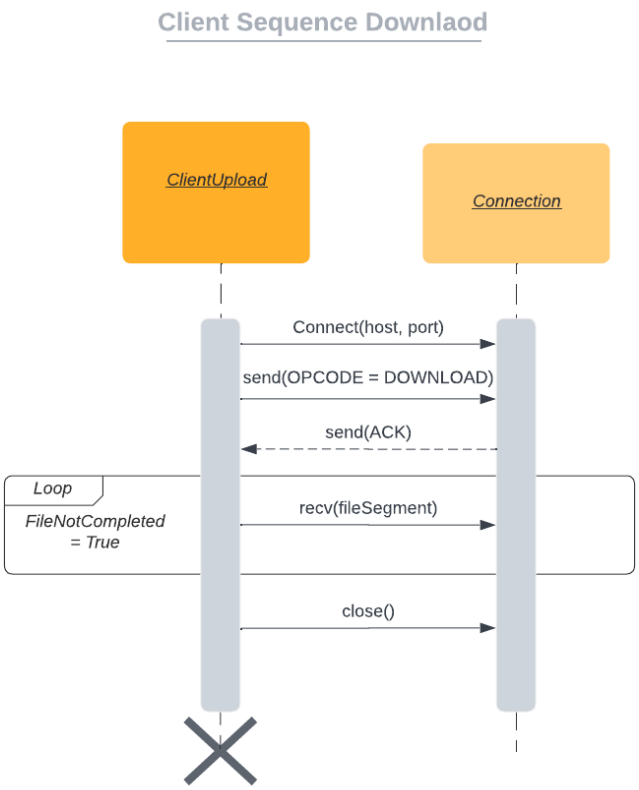
3.1.2. Cliente

El cliente maneja las siguientes entidades fundamentales:

- ClientUploadConnection/ClientDownloadConnection: se genera una instancia por cliente dentro de la cual se resuelve la operación iniciada, ya sea 'UPLOAD' o 'DOWNLOAD'. Toda la comunicación de la capa de aplicación con el servodpr queda encapsualada en esta clase.

- Connection: es el wrapper a nivel aplicación que me brinda la api de comunicación con el servidor.

A continuación se presenta un diagrama de secuencia de una operación de download del lado del cliente:



La operación de upload es equivalente.

3.1.3. Protocolo

Se decidió implementar un protocolo de comunicación a nivel aplicación sencillo. La secuencia de mensajes se estableció de la siguiente manera.

```
# CLI ---> SERVIDOR: Mensaje de Acción
# SERVIDOR ---> CLIENTE: Respuesta de Acción

# En caso de que la acción se confirme ---> Se realiza la acción.
# Caso contrario ---> Se termina la conexión.


# Opcodes:
#   - Upload = b'0; Payload
#   - Download = b'1; Payload
#   - Accepted = b'2; No payload
#   - FileNotFound b'3; No payload


#### ACCIÓN DE UPLOAD ####
# Mensaje de Acción:
#    0 --- 8 --- 16 --- 24 --- 31 --- 39 ----  47 -----------
```

```
#  0 | OP |         file_size         | fileNS |    fileName ...
#     ---------------------------------------------------------
#   OP = Upload
#
# Mensaje de Respuesta:
#    0 --- 8
#  0 | OP |
#     -----
#   OP = Accepted

#### ACCIÓN DE DOWNLOAD ####
# Mensaje de Acción:
#    0 --- 8 ---  15 -----------------
#  0 | OP | fileNS |    fileName ...
#     --------------------------------
#
# Mensaje de Respuesta:
#    0 --- 8
#  0 | OP |
#     -----
#   OP = [Accepted, FileNotFound]
#
```

Una vez aceptada la conexión y recibido el intento de acción el cliente/servidor procederá a enviar iterativamente los segmentos correspondientes al archivo y la conexión terminará cuando se hayan enviado todos los bytes correspondientes al tamaño del archivo. Los mensajes subsiguientes no tienen ningún formato en particular, solamente los bytes de transferencia.

3.2. Capa de Transporte

Para la estructura del disen

```
# Opcodes:
#   - NewConnection = 0x10; No payload
#   - Data = 0x21; Payload = data
#   - Ack = 0x32; No payload
#   - Close = 0x43; No payload ---- chequear!!! no tenemos Close implementado!
#
# Checksum: opcode XOR (payload_size & 0xFF) XOR ~(payload_size >> 8)
#
#    0 ------ 8 ------ 16 ------ 24 ------ 31
#  0 | opcode |   payload size   | checksum |
#     -------------------------------------
#  4 |           sequence number            |
#    0 ------ 8 ------ 16 ------ 24 ------ 31
#
# Handshake
# C> SEGMENTO(opcode=NewConnection, seq=1, payload_sz=0)
# <S SEGMENTO(opcode=Ack, seq=1, payload_sz=0)
# ** connection is established **
#
# Data
# C> SEGMENTO(opcode=Data, payload_sz=4, seq=1, payload="HOLA")
# <S SEGMENTO(opcode=Ack, seq=1, payload_sz=0)
```

```
# <S SEGMENTO(opcode=Data, payload_sz=4, payload="CHAU", seq=1)
# *loss* L> SEGMENTO(opcode=Ack, seq=1, payload_sz=0)
# *rto*
# <S SEGMENTO(opcode=Data, payload_sz=4, payload="CHAU", seq=1)
# C> SEGMENTO(opcode=Ack, seq=1, payload_sz=0)
```

4. Pruebas

A lo largo del desarrollo del trabajo se hicieron varias pruebas con ambos protocolos. Simulando perdidas de manera aleatoria con números random, RTO grandes y chicos, varios clientes en simultaneo realizando la misma operación.

Dejamos esta prueba final donde se utilizo WireShark para ver los paquetes Cliente Servidor en una operación de descarga.

Figura 1: Cliente-Servidor estableciendo la conexión, se puede ver como el cliente pierde 3 veces el paquete de inicio de conexión, finalmente recibe el ACK del servidor y envia la acción de UPLOAD

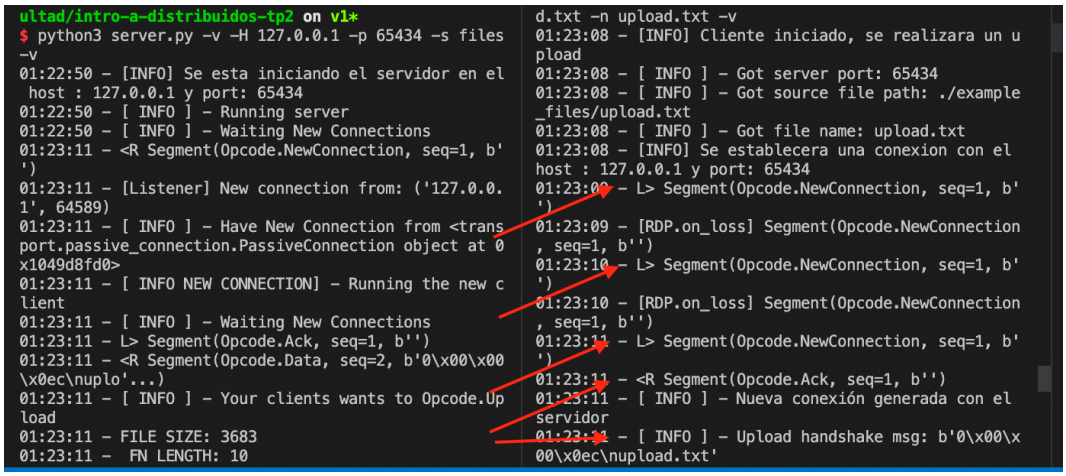


Figura 2: Imagen de Wireshark mostrando el primer intento de handshake del cliente al servidor. Se puede apreciar los otros dos intentos debajo. Se puede observar el opcode.

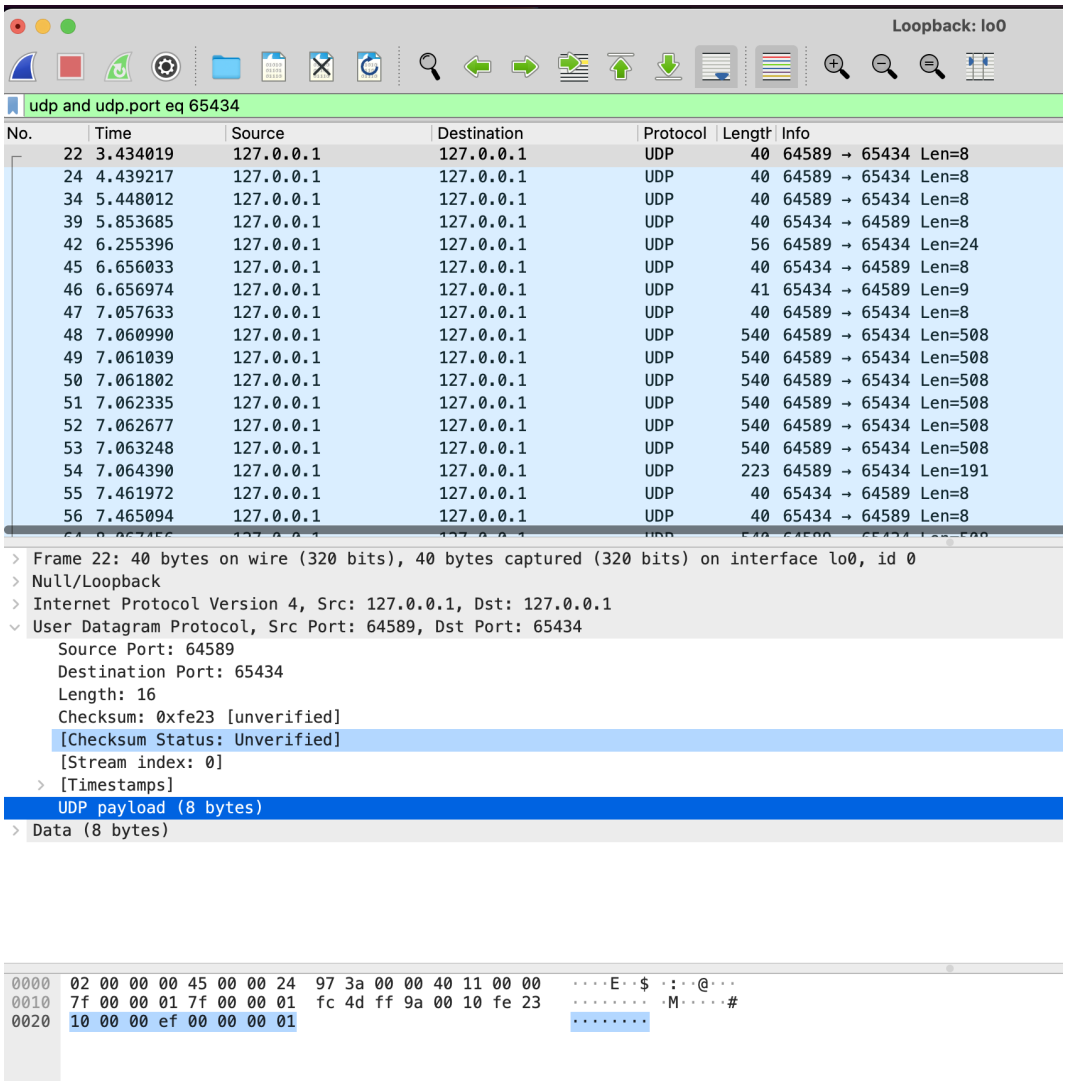


Figura 3: Imagen de Wireshark mostrando el ACK del servidor luego del 3er intento.

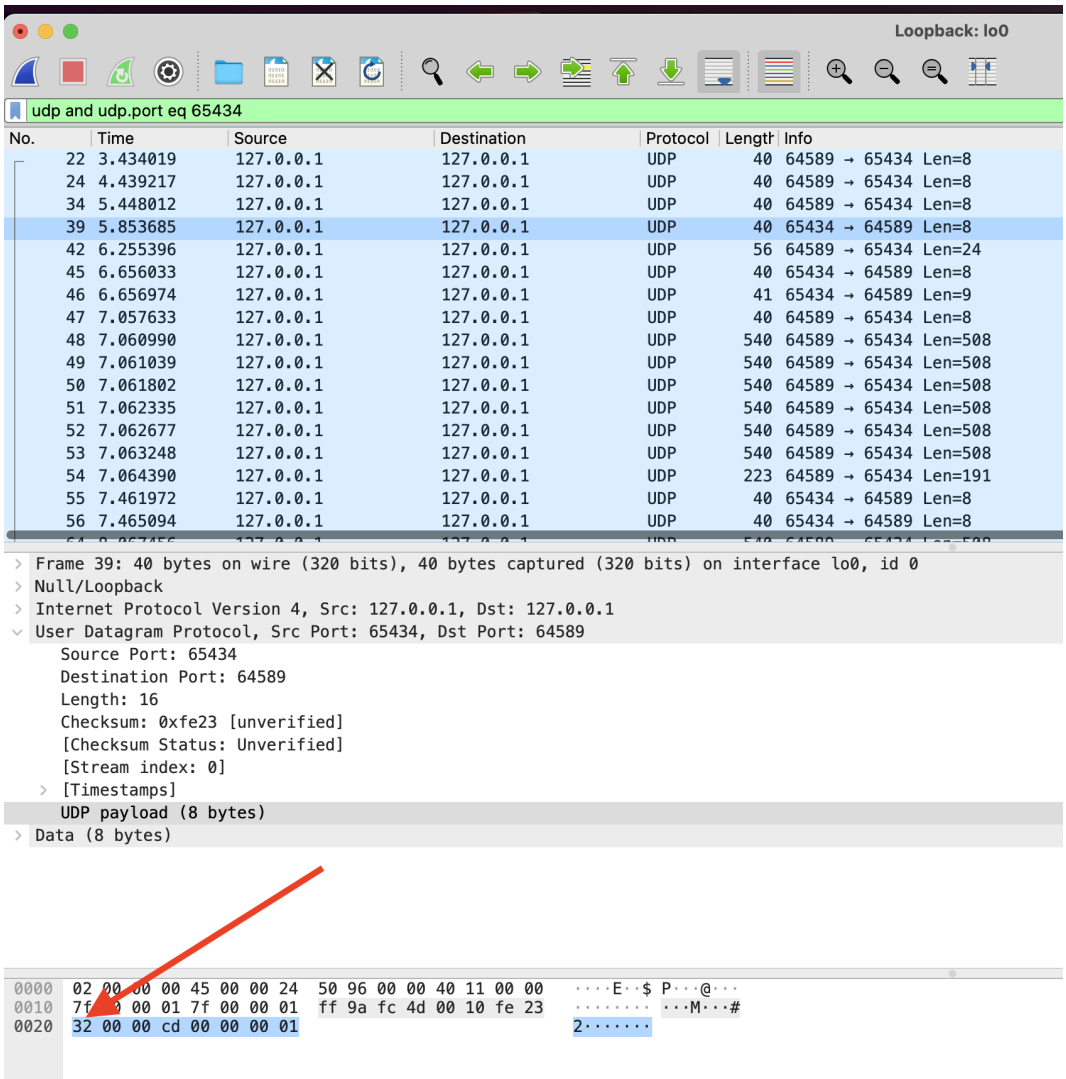
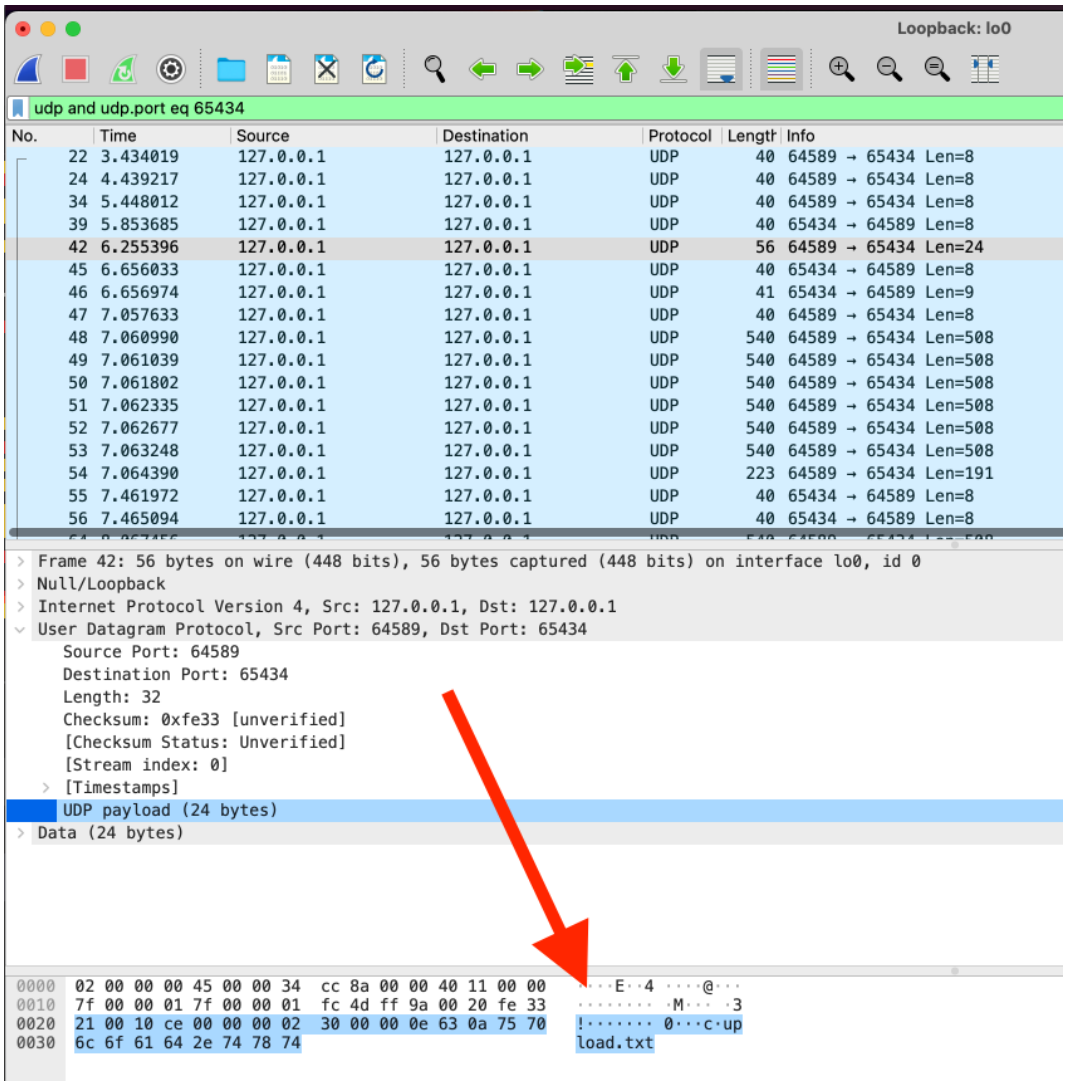


Figura 4: Imagen de Wireshark mostrando el payload de aplicación donde el cliente solicita la carga de la imagen. Se pueden observar el opcode y el nombre de la imagen.



5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

En la arquitectura cliente servidor tenemos un host que actúa de servidor y se encarga de recibir peticiones de otros hosts, llamados clientes.

El servidor tiene una dirección IP fija y conocida por los clientes. En general, suele estar disponible todo el tiempo, permitiendo que un cliente puede contactarlo en cualquier momento.

Algunas de las aplicaciones más conocidas que implementan esta arquitectura son la Web, FTP, Telnet y email.

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

Un protocolo de capa de aplicación define como dos procesos, que corren en diferentes end hosts, se pasan mensajes entre ellos. Los mismos viven en los end-hosts.

Estos protocolos definen:

- Los tipos de mensajes a intercambiar
- La sintáxis de esos mensajes: qué campos y de qué forma están alineados
- La semántica de los campos, es decir, cómo se interpretan los mismos
- Reglas para determinar de qué forma y en qué momentos se deben enviar mensajes y respuestas

f

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

Se detalla en la capa de aplicación del presente informe

5.4. 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?

5.4.1. TCP

TCP es un protocolo orientado a las conexiones que ofrece confiabilidad a la hora de entregar paquetes.

Ofrece los siguientes servicios:

- RDT: reliable data transfer. En otras palabras, TCP asegura que los paquetes que se envían utilizando este protocolo llegan a destino, y lo hacen de forma ordenada.
- Control de congestión: provee un mecanismo, utilizando una receive window y una congestion window, para evitar que la red se congestione de forma rápida.
- Mux/Demux
- Control de integridad: a través de un checksum, TCP puede detectar los paquetes que estan corrompidos, para asi descartarlos y solicitar al remitente que sean reenviados

5.4.2. UDP

UDP es un protocolo que, a diferencia de TCP, no está orientado a las conexiones, y no otorga garantía a la hora de transportar paquetes. Además, los datos se envian apenas están disponibles, sin importar la congestión de la reed.

Ofrece los siguientes servicios:

- Mux/Demux
- Control de integridad: mismo servicio que ofrece TCP, para asegurarse de no aceptar paquetes que estén corrompidos

5.4.3. Comparación

Podemos notar que hay marcadas diferencias entre ambos protocolos. En el caso de TCP, vemos que sería de utilidad usarlo cuando necesitamos confiabilidad en la entrega de los paquetes. Por el otro lado, UDP es más liviano y rápido, pero no nos asegura que los paquetes lleguen todos y en orden. Si bien en el segundo caso tenemos una mejora en la performance, solo sería útil en aquellos casos en los que se puede tolerar la pérdida de paquetes, como lo es por ejemplo una videollamada, o un stream.

6. Dificultades encontradas

6.1. Acceso concurrente en SR

Al implementar el Selective-Repeat(selective_repeat.py) tuvimos un problema de acceso concurrente que bloqueaba al cliente, evitando que este continue enviado sus mensajes y dejando bloqueado así también al servidor.

En la funcion send_segment primero se tomaba el lock de una condition variable (_send_window_cv) que se utilizaba para verificar si el sequence number del paquete a enviar estaba dentro de la send_window y en caso contrario esperaba. Luego ,con la condition variable tomada ,tomabamos otro lock para poder acceder y modificar los atributos de la clase.

Por otro lado en ack_recv, haciamos el proceso invertido, primero para modificar atributos tomabamos el lock y en caso de que debieramos mover la send_window se intentaba tomar la condition variable, para notificar a los que intentaban enviar que la ventana se había corrido.

Lo que ocurría era que el, al estar enviando y recibiendo los acks al mismo tiempo, al enviar se tomaba la condition variable y en paralelo se recibía un ack que tomaba el lock.Entonces, cuando el send iba a tomar el lock no podía hacerlo y se bloqueaba y el recv_ack intentaba tomar la condition variable pero tampoco podía entonces también esperaba. Cada hilo esperaba por la liberación de un recurso teniendo el otro bloqueado,generando un dead lock. De estaba forma el cliente quedaba bloqueado

6.2. Caída de conexión por perdida de últimos paquetes

En selective repeat nos pasaba que si los últimos paquetes enviados se perdían, el servidor cerraba la conexión. Para resolver esto tuvimos que implementar un método close que lo que hace es enviar todo lo que haya quedado en la send_window y luego enviar un mensaje de cierre de conexión.De esta forma si se perdieran los últimos paquetes y el receptor cerrara la conexión porque no recibió mas, se reenviarían los paquetes faltantes y recién ahí el emisor cerraría la conexión correctamente

7. Conclusión

Una vez terminado el trabajo concluimos que gran parte de la dificultad estuvo en armar los protocolos desde cero,entender como iban a funcionar,interactuar entre sí, que componentes

tendrían,etc.Es decir, obtener una primera de vista de la completa solución del problema para ya luego ir implementando las distintas partes.

Fue una decisión bastante acertada comenzar desarrollando por un lado el protocolo de aplicación usando sockets TCP y en paralelo empezar con el desarrollo del protocolo de transporte pedido.Esto nos ahorro tiempo ya que pudimos probar la capa de aplicación sin necesidad de esperar a tener el protocolo de transporte implementado.Esto redujo bastante la cantidad de problemas generados al integrar varias capas, ya que bastante del comportamiento de la capa de aplicación ya había sido probado.

Por otro lado , en la capa de transporte, crear la primer versión usando stop and wait fue lo mas complicado a la hora de plantear los componentes de la solución.Con el stop and wait ya implementado, implementar el selective repeat no fue tan distinto.La principal diferencia fue que en el selective repeat debiamos lidiar con problemas de concurrencia,lo que hizo que nos tomara mas tiempo llegar a la solución.