FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1 — RDTP

[75.43/75.33/95.60] Introducción a Sistemas Distribuidos
Curso 2
Segundo cuatrimestre de 2024

Alumno:	Escandar, Germán
Número de padrón:	105250
Email:	gescandar@fi.uba.ar
Alumno:	Grzegorczyk, Iván
Número de padrón:	104084
Email:	igrzegorczyk@fi.uba.ar

Índice

1. Introducción

El presente trabajo práctico tiene como objetivo la creación de una aplicación de red. Para lograr este objetivo, se deberá desarrollar una aplicación de arquitectura cliente-servidor que implemente la funcionalidad de transferencia de archivos mediante las siguientes operaciones:

- UPLOAD: Transferencia de un archivo del cliente hacia el servidor
- DOWNLOAD: Transferencia de un archivo del servidor hacia el cliente

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.

El protocolo TCP ofrece un servicio orientado a la conexión, garantiza que los mensajes lleguen a destino y provee un mecanismo de control de flujo. Por su parte, UDP es un servicio sin conexión, no ofrece fiabilidad, ni control de flujos, ni control de congestión, es por eso que se implementa una versión utilizando el protocolo Stop & Wait y otra versión utilizando el protocolo Selective ACK, con el objetivo de lograr una transferencia confiable al utilizar el protocolo.

2. Hipótesis y suposiciones realizadas

- Si un archivo ya existe en el servidor se remplaza
- Puede haber clientes usando Stop and Wait y otros con Selective ACK en simultaneo
- No se enviaran archivos lo suficientemente grandes como para que se acaben los números de secuencia y acknowledge resultando en que se reinicie su rango.
- El archivo a descargar siempre se encuentra presente en el servidor.
- El cliente tiene espacio en su disco para guardar los archivos que descarga.

3. Implementacion

3.1. Cliente

La funcionalidad del cliente se divide en dos aplicaciones de línea de comandos: upload y download.

El comando **upload** envía un archivo al servidor para ser guardado con el nombre asignado:

```
python upload [OPTIONS]
```

Donde las opciones son:

- -h/–help: Muestra este mensaje de ayuda
- -v/–verbose: El programa muestra niveles de logging ERROR, INFO, DEBUG
- -q/–quiet: El programa muestra solo el nivel de logging ERROR.
- -H/–host: Dirección IP del del servidor RDTP
- -p/–port: Puerto del servidor RDTP
- -s/–src: Path del archivo a subir.
- -n/–name: Nombre con el que se guardará el archivo a subir.
- –sack: Flag para utilizar Selective ACK en la comunicación, si no se especifica se usa Stop and Wait

En nuestra implementación, sin importar el protocolo, esta operación sigue los siguientes pasos:

1. Crea una instancia del protocolo correspondiente según los parametros ingresados, inicializando un socket en el host "localhost" y el puerto pasado por parámetro
2. Envía los metadatos de la operación al servidor
3. Se abre el archivo a enviar, si este no existe finaliza el programa
4. Mientras que no se haya enviado todo el contenido del archivo:
   a) Se leen UPLOAD_CHUNK_SIZE bytes del archivo
   b) Se envía lo leido al servidor
5. Se cierra el archivo y finaliza el programa

El comando **download** envía un archivo al cliente para ser guardado con el nombre asignado:

```
python upload [OPTIONS]
```

Donde las opciones son:

- -h/–help: Muestra este mensaje de ayuda
- -v/–verbose: El programa muestra niveles de logging ERROR, INFO, DEBUG
- -q/–quiet: El programa muestra solo el nivel de logging ERROR.
- -H/–host: Dirección IP del del servidor RDTP
- -p/–port: Puerto del servidor RDTP
- -d/–dst: Path donde se guardará el archivo a descargar
- -n/–name: Nombre del archivo a descargar
- –sack: Flag para utilizar Selective ACK en la comunicación, si no se especifica se usa Stop and Wait

En nuestra implementación, sin importar el protocolo, esta operación sigue los siguientes pasos:

1. Crea una instancia del protocolo correspondiente según los parametros ingresados, inicializando un socket en el host "localhost" y el puerto pasado por parámetro.
2. Envía los metadatos de la operación al servidor.
3. Se crea un archivo nuevo con el nombre especificado y se lo abre en modo de escritura
4. Mientras que no se haya recibido todo el contenido del archivo:
   a) Se leen DOWNLOAD_CHUNK_SIZE bytes del socket
   b) Se escribe lo leido al archivo
5. Se cierra el archivo y finaliza el programa

3.2. Servidor

El servidor aguardará una solicitud de un cliente y responderá acorde a la operación que este desee realizar.

```
python server [OPTIONS]
```

Donde las opciones son:

- -h/–help: Muestra este mensaje de ayuda
- -v/–verbose: El programa muestra niveles de logging ERROR, INFO, DEBUG
- -q/–quiet: El programa muestra solo el nivel de logging ERROR.
- -H/–host: Establece un host para el servidor. Por defecto localhost
- -p/–port: Puerto del servidor RDTP
- -s/–storage: Path donde se guardarán los archivos que los clientes carguen o descarguen
- –sack: Flag para utilizar Selective ACK en la comunicación, si no se especifica se usa Stop and Wait

El servidor va a proveer el servicio de almacenamiento y descarga de archivos. Para ello seguirá los siguientes pasos:

1. Cuando se ejecute el comando, creará un nuevo servidor con la implementación del protocolo determinado dependiendo de si se pasa el flag –sack.
2. Una vez que el mismo está creado, se quedará a la espera de una nueva conexión.
3. Cuando recibe una conexión nueva, abre un thread para resolverlo. Por lo tanto en simultáneo, está preparado para recibir nuevas conexiones. A la vez, cada thread inicializará un nuevo socket para comunicarse con el cliente, con el objetivo de no sobrecargar la cola de envío del socket del servidor.
4. Recibe la metadata de **UPLOAD** o **DOWNLOAD** y se prepara para ejecutar alguna de las dos operaciones permitidas

Para **UPLOAD**: Hay que encargarse de recibir un archivo

1. Crea la carpeta de destino que se pasa por parámetro en el caso de que no exista.
2. Recibirá el archivo de a segmentos de tamaño RECV_CHUNK_SIZE, iterando hasta que se reciba una cantidad de bytes igual al tamaño del archivo siendo transferido.
3. Una vez finalizada la transferencia, se cierra el socket de la conexión

Para **DOWNLOAD**: Hay que encargarse de recibir un archivo

1. Crea la carpeta de destino que se pasa por parámetro en el caso de que no exista.
2. Recibirá el archivo de a segmentos de tamaño RECV_CHUNK_SIZE, iterando hasta que se reciba una cantidad de bytes igual al tamaño del archivo siendo transferido.
3. Una vez finalizada la transferencia, se cierra el socket de la conexión

4. Preguntas a responder

1. **Describa la arquitectura Cliente-Servidor.**

   La arquitectura cliente-servidor es un modelo de diseño de software en el que el proveedor de un servicio, llamado servidor, responde a peticiones de quienes consumen el servicio, llamados clientes. La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un solo programa (por ej: los servidores web de aplicaciones masivas suelen derivar las solicitudes de sus clientes a balanceadores de carga, que a su vez vuelven a redirigir a un servicio que pueda responder).

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

   Un protocolo de capa de aplicación determina cómo se comunican entre sí los procesos de aplicaciones que corren en diferentes sistemas finales. Para ello define:

   - Los tipo de mensaje: solicitud o respuesta.
   - Los campos que tiene cada tipo de mensaje y el significado de cada uno.
   - Reglas para determinar cuándo y cómo un proceso envía y responde mensajes.

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

   Para este trabajo práctico se desarrolló un protocolo que cuenta con 2 modalidades de comunicación:

   - **Stop and Wait**: En esta modalidad, el emisor envía un paquete al receptor y solo cuenta el envío como finalizado una vez que recibe el ACK correspondiente. Ante un timeout al recibir dicha confirmación, se reenvía el paquete y se vuelve a esperar el ACK hasta un máximo de MAX_RETRIES veces. De esta forma nos aseguramos que los paquetes llegan en orden al receptor, por lo que este mismo podrá procesar cada paquete que reciba inmediatamente y descartar cualquier paquete duplicado que llegue como resultado de alguna perdida.
   - **Selective ACK**: En esta modalidad, el emisor mantiene una ventana de paquetes enviados, y solo permite enviar uno nuevo si hay lugar en dicha ventana para guardar el paquete a enviar. En comparación a la modalidad anterior, aquí el emisor envía un paquete al receptor y espera el ACK **una vez**. Si lo recibe, quita el paquete correspondiente de la ventana y ante timeout simplemente devuelve el control a la aplicación.

     Si al momento de enviar el paquete la ventana está llena, se reenvían todos los paquetes de la ventana y después de cada envío se espera el ACK. Esto se repite tantas veces como sea necesario hasta que se haga lugar en la ventana para enviar el paquete.

     Los paquetes pueden llegar desordenados al receptor, por lo que este mantiene un buffer de recepción en el que va guardando los paquetes que llegan por orden de número de secuencia. Además, si un paquete llega fuera de orden, se guarda su número de secuencia y el número de ACK esperado para el mismo (nsecuencia + tamaño) y se envía esa información al emisor para confirmar la recepción del paquete y para que el emisor pueda quitarlo de la ventana.

   Luego, independientemente de la modalidad a utilizar, el protocolo sigue los siguientes pasos:

   a) El cliente envía un paquete vacío al servidor para indicarle que desea realizar una operación

   b) El servidor crea un handler (un objeto de tipo ClientOperationHandler), que representa al hilo que administrará la conexión con el cliente

   c) El handler le envía un ACK al cliente, para que este sepa que ahora deberá comunicarse con el socket del handler.

d) Luego, el cliente sigue los pasos descriptos en la sección de implementación, dependiendo de la operación que desee realizar.

Ejemplos de uso

```
ivan@ivan-ThinkPad-E15-Gen-2:~/Documentos/personal/tp1-intro-distribuidos$ tail -f client1.log
2024-10-02 20:12:22,753 -  [ INFO ]              - Starting download for file small.txt (operations.py:3
6)
2024-10-02 20:12:22,753 -  [ DEBUG ]             - Sending handshake message to: sockaddr(host='10.0.0.
1', port=4000) (transport.py:239)
2024-10-02 20:12:22,753 -  [ DEBUG ]             - Sent 11 bytes to sockaddr(host='10.0.0.1', port=4000
), with data_len=0, seq=0, ack=0 (transport.py:225)
2024-10-02 20:12:22,863 -  [ DEBUG ]             - Sent 11 bytes to sockaddr(host='10.0.0.1', port=4000
), with data_len=0, seq=0, ack=0 (transport.py:225)
2024-10-02 20:12:22,865 -  [ DEBUG ]             - Reading fd 4: ([4], [], []) (transport.py:289)
2024-10-02 20:12:22,866 -  [ DEBUG ]             - Finished handshake, now talking to: sockaddr(host='1
0.0.0.1', port=37428) (transport.py:249)
2024-10-02 20:12:22,866 -  [ DEBUG ]             - Sent 22 bytes to sockaddr(host='10.0.0.1', port=3742
8), with data_len=11, seq=0, ack=0 (transport.py:225)
2024-10-02 20:12:22,866 -  [ DEBUG ]             - Waiting for ack... (nattempt=0) (transport.py:346)
2024-10-02 20:12:22,867 -  [ DEBUG ]             - Reading fd 4: ([4], [], []) (transport.py:289)
2024-10-02 20:12:22,867 -  [ DEBUG ]             - Received ack: 11, expected ack=11, nattempt=0 (trans
port.py:348)
2024-10-02 20:12:22,867 -  [ DEBUG ]             - Reading fd 4: ([4], [], []) (transport.py:289)
2024-10-02 20:12:22,868 -  [ DEBUG ]             - Received packet of length: 4 seq: 0, expected seq=0
(transport.py:260)
2024-10-02 20:12:22,868 -  [ DEBUG ]             - got package with seq=0, length=4. sending ACK to soc
kaddr(host='10.0.0.1', port=37428). pkt=[seq: 11, ack: 4, len(data): 0, op_metadata: False, sack_options
: []] (transport.py:273)
2024-10-02 20:12:22,868 -  [ DEBUG ]             - Sent 11 bytes to sockaddr(host='10.0.0.1', port=3742
8), with data_len=0, seq=11, ack=4 (transport.py:225)
2024-10-02 20:12:22,868 -  [ DEBUG ]             - Got file size of 17, fetching data (operations.py:43
)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - Reading fd 4: ([4], [], []) (transport.py:289)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - Received packet of length: 17 seq: 4, expected seq=4
 (transport.py:260)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - got package with seq=4, length=17. sending ACK to so
ckaddr(host='10.0.0.1', port=37428). pkt=[seq: 11, ack: 21, len(data): 0, op_metadata: False, sack_optio
ns: []] (transport.py:273)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - Sent 11 bytes to sockaddr(host='10.0.0.1', port=3742
8), with data_len=0, seq=11, ack=21 (transport.py:225)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - Writing packet seq: 4, ack: 11, len(data): 17, op_me
tadata: False, sack_options: [] to file (operations.py:48)
2024-10-02 20:12:22,869 -  [ INFO ]              - Finished downloading file small.txt (operations.py:50
)
2024-10-02 20:12:22,869 -  [ DEBUG ]             - Closing UDP socket (transport.py:325)
```

Figura 1: Cliente con Stop and Wait - Ejemplo de descarga

4. **La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características?¿Cuando es apropiado utilizar cada uno?**

   La capa de transporte tiene como principal objetivo extender el servicio de entrega de la capa de red a la capa de aplicación, entre procesos corriendo en diferentes sistemas finales. Dentro del stack TCP/IP se tienen los protocolos UDP y TCP, cada uno con sus respectivos servicios y características.

   UDP: User Datagram Protocol

   Características

   - Con pérdida de paquetes y posibilidad de paquetes duplicados
   - No orientado a la conexión
   - Pocos headers

   Servicios

   - Entrega de datos process-to-process, a traves de un canal de comunicación full duplex.

- Chequeo de errores/integridad. Lo hace poniendo el campo de detección de error (checksum) en los headers.

TCP: Transmission Control Protocol

Características

- Sin pérdida de paquetes ni duplicados (envío confiable)
- Orientado a la conexión
- Muchos headers

Servicios

TCP provee los mismos servicios que UDP, y se le suman los siguientes

- Reliable Data Transfer: Esto implica que a la capa de aplicación le van a llegar los paquetes en orden, sin perdidas, sin errores y sin duplicados.
- Control de flujo: Evita que el que envía desborde a la aplicación que hace la lectura del buffer. Iguala la tasa de envió a la de lectura.
- Control de congestión: Limita la tasa a la que el transmisor envía trafico a la conexión en función de la congestión que percibe en la red.

5. Dificultades encontradas

- Resultó compleja la implementación del algoritmo que actualiza los bloques de SACK del receptor, ya que tratamos de serle fiel a la RFC de SACK en la medida de lo posible.
- Tomó mucha prueba y error implementar correctamente el buffer de recepción para SACK
- Se presentaron dificultades para la instalación y correcta ejecución del programa mininet.

6. Conclusiones

- Se logro adquirir conocimiento sobre el funcionamiento de sockets UDP y arquitectura Cliente-Servidor expandiendo lo aprendido en materias previas, ganando una nueva apreciación por la complejidad de TCP.
- Se pudo observar la diferencia en performance entre Stop and Wait y SACK, siendo esta última la mejor opción ante la perdida de paquetes.
- Se podría optimizar la performance del protocolo, disminuyendo el tamaño maximo del header o mejorando los algoritmos de SACK (de actualizacion de ventana y de bloques de paquetes cuya recepcion fue confirmada).