FACULTAD DE INGENIERIA
Universidad de Buenos Aires

FIUBA - 75.43

Introducción a los Sistemas Distribuidos

Trábajo Práctico N∘1: TCP File Transfer
1er cuatrimestre, 2021

Integrantes:

Nombre	Padrón	Mail
Levi Fernández, Matías	99119	milevif@fi.uba.ar
Pérez Machado, Axel	101127	axelmpm@gmail.com
Franco, Tomás	91013	tomasnfranco@gmail.com

Fecha de Vencimiento: 14/05/2021

Índice

1. Introducción

El objetivo de este Trabajo Práctico es que los estudiantes puedan lograr una mayor comprensión cómo se comunican distintos procesos a través de la red. Para afianzar estos conceptos se deberá crear una aplicación de arquitectura cliente-servidor que implemente la funcionalidad de transferencia de archivos.

2. Hipótesis y suposiciones realizadas

Si se hace upload-file

```
el -d es el path de donde el cliente saca el file a mandar
el -n es el nombre con el que quiere que el server guarde el
archio
el archivo el server lo guarda en su directorio marcado por el
argumento -s
```

Si se hace download-file

```
el -d es el path de donde el server saca el file que el cliente
le pide
el -n es el nombre con el que el cliente se guarda el archivo
el cliente se guarda el archivo en la carpeta de donde se
ejecuta el comando
```

Asumimos que el cliente conoce los archivos que hay disponibles.

Si un archivo de mismo nombre que otro existente en ese directorio es creado entonces se le cae encima al anterior.

Si el archivo pedido para descarga o del cual se usa para la carga no se encuentra en el file system se tira un error.

3. Implementación

Las aplicaciones a crear deben implementar 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

El protocolo de capa de transporte mediante el cual se comunicarán debe ser TCP.
A su vez, ambas aplicaciones deben ser desarrolladas en el lenguaje Python respetando la guía de estilos PEP8, utilizando la librería estándar de sockets y deben poder ser desplegadas en localhost.

3.1. Aplicacion

Para encarar este problema, analizamos y dividimos las responsabilidades del cliente y servidor en sus respectivas files (server-tcp, client-tcp). Ademas, se encontro funcionalidad en comun que se consolido en un archivo 'common-tcp'

- server-tcp: posee un unico metodo 'serve' que se encarga de inicializar el servidor y colocarlo a la escucha de nuevas conexiones y, en el momento que se empieza una de ellas, se inicializa en una clase propia llamada 'connection-instance' que representa una conexion activa directa con un cliente y encapsula el protocolo de comunicacion.

- cliente-tcp: el cliente esta encapsulado en una clase que se encarga de encapsular el protocolo de comunicacion con el servidor por medio de un socket-tcp.
- common-tcp: en este archivo se encuentra la implementacion del wrapper del manejador de archivos (FilesManager) y el wrapper del socket (socket-tcp) ambos comunes a tanto a el server como el client.

3.2. Protocolo

Durante el desarrollo, nos encontramos con la necesidad de desarrollar un orden o metodologia de comunicacion, por eso en el caso del socket-tcp, se encapsula las acciones comunes utilizadas por el protocolo de comunicacion: Debido a que en el problema a resolver se presentan situaciones donde el recv-er necesita recibir informacion especifica (por ejemplo, el tamaño o nombre del archivo) se opto por encapsular este funcionamiento en el wrapper del socket. En las situaciones donde uno de los lados de la conexion necesita esperar por la confirmacion del otro, se decidio por realizar un sistema de $ACKs$ en donde se señaliza con un mensaje predefinido $OK - ACK$

Finalmente, el socket es utilizado por las clases de client-tcp y connection-instance (en el server), las cuales poseen la implementacion real del protocolo.

A modo de ejemplo, imaginemos que queremos enviar desde el cliente un archivo 'archivo.file' asumiendo ya iniciada la conexion, desde el client la secuencia del protocolo seria:

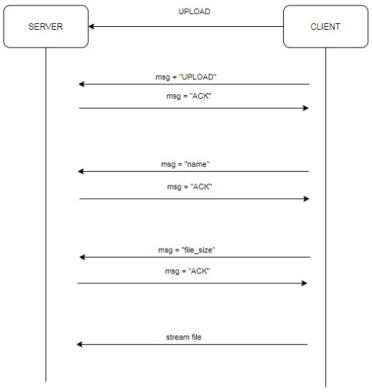
Cliente

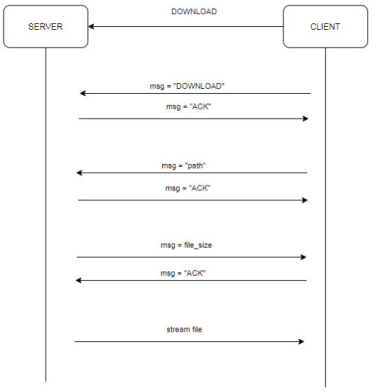
```
# pseudocodigo
socket.send(UPLOAD)
socket.wait_ack()
socket.send_file_name(filename)
socket.wait_ack()
socket.send_file_size(filesize)
socket.wait_ack()

socket.send_file(file)

socket.close()
```

Servidor:

```
# pseudocodigo
action = socket.recv_action()
dispatch_action(action) # action sera igual a UPLOAD en este caso

# dentro de dispatch_action
socket.send_ack()
socket.recv_fname()
socket.send_ack()
socket.recv_fsize()
socket.send_ack()

socket.recv_file()

socket.close()
```

En el caso donde se quiera realizar una DOWNLOAD en vez de UPLOAD, el procedimiento del protocolo es similar pero con los roles invertidos.





3.3. Testing

En la carpeta lib tambien adjuntamos una carpeta tests compuesta por la especificacion de los test corridos (especificacion per se, comandos asociados, numero de test) y sus respectivos resultados por consola en forma de captura de pantalla tanto del server como del client.

4. Preguntas a responder

1) Describa la arquitectura Cliente-Servidor

En una arquitectura Cliente-Servidor existen una aplicación cliente y una servidor que se comunican entre sí. Esta última puede atender a varios clientes de forma simultánea.

Es el cliente quien inicia la conexión realizando una petición al servidor. Este debe estar siempre disponible, ya que no conoce cuándo un cliente va a necesitar comunicarse.

El servidor debe tener una dirección física conocida a la cual los clientes puedan enviar sus peticiones.

La principal limitación de esta arquitectura es que los clientes pueden comunicarse solamente con el servidor pero no entre sí.

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

La función de un protocolo de capa de aplicación es proveer una estandarización para los mensajes que las aplicaciones se envían y reciben.

Esta estandarización permite que las mismas logren entenderse, logrando una comunicación efectiva.

3) Detalle el protocolo de aplicación desarrollado en este trabajo Ya previamente explicado en la seccion de Implementacion: Protocolo

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?

Los servicios que son provistos por el protocolo TCP (Transmission Control Protocol) son:

- Conexión: El cliente y el servidor deben iniciar una comunicación mediante un handshake antes de poder comenzar a intercambiar mensajes. La conexión entre estos es full-duplex, lo cual significa que pueden enviarse datos mutuamente de forma simultánea.

- Garantía de entrega, integridad y orden: Este protocolo implementa mecanismos para garantizar que los mensajes lleguen de forma completa y en el orden en que fueron enviados.
- Control de congestión: Este protocolo regula el throughput en función del estado de carga de la red, lo cual permite preservar el bienestar de la red y el uso equitativo de la misma por parte de los usuarios.

Por otro lado, el protocolo UDP (User Datagram Protocol) no ofrece ninguno de los servicios descriptos para TCP: Es sin conexión, no proporciona control de congestión ni garantías de entrega y orden.

La principal ventaja de UDP radica en que permite una velocidad de transferencia superior ya que implica un overhead un impacto en CPU menor.

Debido a los motivos expuestos, TCP es utilizado por aquellas aplicaciones que necesitan mayor robustez en la transferencia de sus paquetes. Un ejemplo de esto es la transferencia de archivos por SSH o FTP, donde deben existir las garantías que son provistas de forma automática por este protocolo.

En cambio, UDP suele ser utilizado por aplicaciones cuyo funcionamiento puede tolerar la pérdida de algunos paquetes al mismo tiempo que la mayor velocidad representa una ventaja importante. Un ejemplo de esto son las aplicaciones de streaming o de videollamadas.

5. Dificultades encontradas

Al ser un programa sencillo, no se encontraron dificultades mayores. Aun asi, hubo instancias donde se plantearon las siguientes cuestiones:

1.

- ¿Como coordinamos las interacciones entre el servidor y el cliente?
  Esta necesidad de una interaccion coordinada la resolvimos definiendo un protocolo de comunicacion comun entre el server y el client.
- ¿Cual es la mejor estrategia para implementar los distintos niveles de verbosidad?
  Terminamos optando por implementar un wrapper sobre la impresora, el cual tiene distintos niveles de herencia para representar los distintos niveles de verbosidad, esta impresora se crea en el momento de parsear los argumentos y se pasa como parametro al server/client
- ¿Como encaramos el hecho de que se debe soportar archivos de cualquier tipo?
  Al comienzo, nuestros tests se basaron unicamente en archivos de texto, y al momento de intentar de pasar otros formatos, nos encontramos con una necesidad de refactorizar. Afortunadamente, al haber encapsulado los metodos del protocolo en comun dentro del socket, pudimos (aumentando el nivel de indireccion) redirigir las llamadas internas a socket.recv/send sin la necesidad de hacer cambios en el codigo.

  Antes

```
# dentro del socket.send_file()
# pseudocodigo

data = file.read(chunk)
socket.send(data.encode())

#dentro de socket.send_ack
socket.send(OK_ACK.encode())
```

  Despues

```
# dentro del socket.send_file()
# pseudocodigo

data = file.read(chunk)
socket.send_raw_data(data)

#dentro de socket.send_ack
socket.send_encoded(OK_ACK)
```

6. Conclusión

Si bien este fue un trabajo introductorio, como conclusion podemos decir que aunque utilizar sockets TCP soluciona muchos de las dificultades que se presentarian con UDP, aun asi es necesario desarrollar un protocolo de alto nivel para realizar operaciones complejas: al ser un socket un stream de datos, es necesario establecer un cierto orden para poder interpretar la informacion recibida.