Trabajo Práctico 1
TCP File Transfer

Introducción a los Sistemas Distribuidos (75.43)
Facultad de Ingeniería, Universidad de Buenos Aires

1^{er} Cuatrimestre 2021

Integrantes
Mauro Parafati - 102749 - mparafati@fi.uba.ar
Taiel Colavecchia - 102510 - tcolavecchia@fi.uba.ar
Yuhong Huang - 102146 - yhuang@fi.uba.ar

Índice

1. Introducción

El presente trabajo práctico tiene como objetivo la **creación de una aplicación de red**.

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.

Además, para poder lograr el objetivo planteado, se aprenderá el uso de la interfaz de sockets TCP y la arquitectura básica de una aplicación cliente-servidor.

2. Hipótesis y suposiciones realizadas

Se listan a continuación las distintas hipótesis y suposiciones realizadas en la etapa de análisis del problema, para luego ser tenidas en cuenta a la hora de implementar la solución propuesta.

2.1. Cliente

- Puede especificarse cualquier ruta fuente para subir los archivos, incluyendo sub-directorios.
- Puede especificarse cualquier ruta destino para descargar los archivos, incluyendo sub-directorios.
- Si el *host* dado para la conexión con el servidor no es válido, o si el mismo no está aceptando conexiones en el puerto especificado, el cliente finalizará su ejecución.
- En el modo *quiet* ($-q$ o $--quiet$) se debe loggear lo mínimo e indispensable, en nuestro caso consideramos sólo los errores críticos. Cualquier otra cosa no será incluida en este modo.

2.2. Servidor

- El nombre de los archivos que se ofrecen para descargar se considera como el nombre del archivo y su extensión.
- No es necesario que se puedan crear sub-directorios para almacenar archivos. Todos los archivos comparten directorio raíz de la ruta especificada para la ejecución del servidor.
- No hay parámetros obligatorios para correr el servicio, es decir que si no se especifica *host*, *puerto* o *directorio de almacenamiento* se utilizarán valores por defecto. (ver `constants.py`)
- En el modo *quiet* ($-q$ o $--quiet$) se debe loggear lo mínimo e indispensable, en nuestro caso consideramos sólo los errores críticos. Cualquier otra cosa no será incluida en este modo.
- Si se recibe un archivo con un nombre que ya existe en el directorio de archivos del servidor, el mismo se reemplaza por el recibido.

3. Implementación

3.1. Lineamientos Generales

Se utilizaron *hashbangs* al comienzo de los archivos para lograr la ejecución de los scripts en `python3`:

```
#!/usr/bin/python3
```

Para cada script a ejecutar, se utilizó la biblioteca nativa de `python` `argparse` para lograr los requerimientos de opciones para cada uno.

3.2. Utilización de Sockets TCP/IP

Se optó por implementar una clase Socket que encapsule la clase nativa del lenguaje para facilitar su uso, y con el objetivo también de que si se quisiera utilizar otro protocolo de transporte como podría ser UDP, debería simplificar la transición. La misma provee como interfaz la capacidad de:

- Para un cliente:
  - Conectarse a un servidor.
- Para un servidor:
  - Asociar el socket a un puerto local.
  - Encender el socket para esperar nuevas conexiones.
  - Aceptar nuevas conexiones.
- Para la comunicación (ambos extremos de una conexión):
  - Enviar datos en forma de bytes.
  - Recibir datos en forma de bytes.

3.3. Flujo de ejecución

3.3.1. Servidor

El servidor tiene un funcionamiento multi-hilo para poder aceptar nuevos clientes y responder sus consultas de forma concurrente. Así, un servidor en ejecución consta de al menos dos hilos: el principal y el aceptador de nuevos clientes, además por cada nuevo cliente conectado comenzará su ejecución un nuevo hilo *handler* (`ClientHandler`) para el mismo. Los hilos secundarios fueron implementados utilizando la clase `Thread` nativa de la biblioteca `threading` de `python`.

Al comenzar su ejecución el hilo principal inicializará el socket (eso es, si los parámetros, lanzará la ejecución del hilo aceptador (`Accepter`) y esperará con a un *prompt* con dos opciones:

- `s`: Para imprimir *stats* del servidor.
- `q`: Para terminar ordenadamente la ejecución del programa. Cuando se preciona esta tecla, el hilo principal cerrará su ejecución finalizando la espera del `Accepter`.

El hilo aceptador, tiene como ciclo principal de ejecución una espera de nuevos clientes. A medida que las conexiones nuevas son iniciadas, el mismo lanza un hilo para la resolución de la consulta del nuevo cliente y une a cada hilo lanzado anteriormente que ya haya finalizado su ejecución.

Cuando este debe finalizar su ejecución, espera a la finalización de los flujos de los hilos hijos (todos los `ClientHandler` lanzados).

Cada `ClienHandler` comienza su ejecución con la conexión nueva de un cliente, realiza las acciones para resolver la consulta del mismo enviando las respuestas correspondientes para luego finalizar su ejecución.

3.3.2. Cliente

La ejecución del cliente, independientemente del script a ejecutar funciona sobre únicamente el hilo principal finalizando su ejecución ante cualquier error de lectura de archivos o conexión con el servidor. La comunicación con el servidor se realiza mediante el protocolo descripto a continuación.

3.4. Protocolo de aplicación

Para poder establecer cualquier tipo de comunicación, es necesario un protocolo que permita a ambos extremos de la misma "entenderse" y "hablar en el mismo idioma".

Para esto, una vez establecida la conexión TCP mediante el sistema de *3-way handshake* (de esto se encarga la abstracción de socket provista por el sistema operativo), se define una estructura que deben seguir todos los mensajes que se envíen entre el Cliente y el Servidor.

Todo mensaje que el cliente envía al servidor, comienza por un byte de *opcode*. Existen tres opcodes representando las tres posibles operaciones, y el resto del mensaje dependerá de la opción elegida:

- **UPLOAD_FILE_OP:**
  1. El cliente informa que está enviando un archivo.
  2. Una vez que el servidor valida el opcode, le devuelve el estado exitoso en una respuesta de un byte, denominado *estado*.
  3. El cliente envía el nombre del archivo, para lo cual primero envía la longitud del mismo utilizando 8 bytes fijos, y luego procede a enviar los bytes que conforman al nombre del archivo en sí.
  4. El cliente envía el archivo, siguiendo el mismo procedimiento: primero la longitud en 8 bytes fijos, y luego los bytes que conforman al archivo en sí. Una vez que envía el archivo cierra la conexión y termina ordenadamente.
  5. Una vez que el servidor recibe el archivo, lo guarda, y cierra la conexión con el cliente ordenadamente.
- **DOWNLOAD_FILE_OP:**
  1. El cliente informa que está descargando un archivo.
  2. Una vez que el servidor valida el opcode, le devuelve el estado exitoso en una respuesta de un byte, denominado *estado*.
  3. El cliente envía el nombre del archivo que quiere descargar, para lo cual se utiliza el mismo proceso descripto en **UPLOAD_FILE_OP**.
  4. El servidor le envía una confirmación de un byte al cliente si el archivo existe. Caso contrario, le envía en este mismo byte un código de error.
  5. Si el cliente recibe código de error, se termina la ejecución. Caso contrario, se espera

recibir el archivo del servidor.

6. El servidor envía el archivo, siguiendo el mismo procedimiento: primero la longitud en 8 bytes fijos, y luego los bytes que conforman al archivo en sí. El servidor entonces cierra la conexión con el cliente ordenadamente.

7. Una vez que el cliente recibe el archivo, cierra la conexión con el servidor ordenadamente.

- **LIST_FILES_OP:**

1. El cliente informa que quiere recibir la lista de archivos del servidor.

2. Una vez que el servidor valida el opcode, le devuelve el estado exitoso en una respuesta de un byte, denominado *estado*.

3. El servidor envía la lista de archivos al cliente, para lo cual primero envía la longitud en 8 bytes fijos y luego envía la lista en formato binario. Luego cierra la conexión con el cliente de forma ordenada.

4. El cliente recibe la longitud y luego con esto la lista en formato binario, para luego parsearla y poder mostrarsela al usuario. Finalmente cierra la conexión con el server.

4. Casos de prueba

Terminada la implementación de la solución propuesta, se procedió a llevar a cabo distintos casos de prueba que permiten ver a la arquitectura en funcionamiento.

4.1. Caso 1: Subir un archivo

- **Objetivo:** Permite al cliente solicitante subir un archivo al servidor, para que otros luego puedan verlo en la lista de archivos y/o descargarlo.
- **Descripción del flujo:**
  1. El cliente establece la conexión con el servidor.
  2. El cliente envía el archivo siguiendo el protocolo descripto.
  3. El servidor recibe el archivo.
  4. El servidor guarda el archivo en su directorio, disponibilizandolo para el resto de sus clientes (si el nombre es repetido, se lo reemplaza).
- **Capturas de pantalla:**

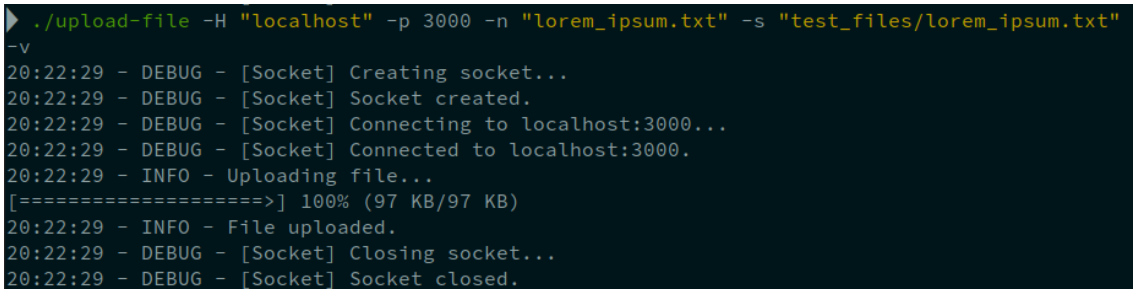

Figura 1: Vista del cliente

```
./start-server -H "localhost" -p 3000 -v
20:22:26 - DEBUG - [Socket] Creating socket...
20:22:26 - DEBUG - [Socket] Socket created.
20:22:26 - DEBUG - [Socket] Binding to localhost:3000...
20:22:26 - DEBUG - [Socket] Bound to localhost:3000.
20:22:26 - DEBUG - [Accepter] Waiting for client...
20:22:26 - DEBUG - [Socket] Accepting client...
20:22:26 - INFO - Listening on port 3000.
Enter `s` to print stats, or `q` to exit.
20:22:29 - DEBUG - [Socket] Client accepted from 127.0.0.1:47498.
20:22:29 - DEBUG - [Socket] Creating socket...
20:22:29 - DEBUG - [Socket] Socket created.
20:22:29 - DEBUG - [ClientHandler:0] Started.
20:22:29 - DEBUG - [Accepter] Waiting for client...
20:22:29 - DEBUG - [Socket] Accepting client...
20:22:29 - INFO - File lorem_ipsum.txt uploaded from 127.0.0.1:47498.
20:22:29 - DEBUG - [ClientHandler:0] Finished.
20:22:33 - DEBUG - [Socket] Closing socket...
20:22:33 - DEBUG - [Socket] Socket closed.
20:22:33 - DEBUG - [ClientHandler:0] Forcing join.
20:22:33 - DEBUG - [Socket] Closing socket...
20:22:33 - DEBUG - [Socket] Socket closed.
20:22:33 - DEBUG - [ClientHandler:0] Joined.
20:22:33 - DEBUG - [Accepter] Client joined.
20:22:33 - DEBUG - [Socket] Closing socket...
```

Figura 2: Vista del servidor

4.2. Caso 2: Descargar un archivo

- **Objetivo:** Permite al cliente cargar un archivo desde el servidor.
- **Descripción del flujo:**
  1. El cliente establece la conexión con el servidor.
  2. El cliente le solicita el archivo al servidor.
  3. El servidor le envía el archivo solicitado al cliente, de existir.
- **Capturas de pantalla:**

```
./download-file -H "localhost" -p 3000 -n "audio.mp3" -d "test_files/audio.mp3" -v
20:25:54 - DEBUG - [Socket] Creating socket...
20:25:54 - DEBUG - [Socket] Socket created.
20:25:54 - DEBUG - [Socket] Connecting to localhost:3000...
20:25:54 - DEBUG - [Socket] Connected to localhost:3000.
20:25:54 - INFO - Downloading file...
[<====================] 100% (15 MB/15 MB)
20:25:54 - INFO - File downloaded.
20:25:54 - DEBUG - [Socket] Closing socket...
20:25:54 - DEBUG - [Socket] Socket closed.
```

Figura 3: Vista del cliente

```
./start-server -H "localhost" -p 3000 -v
20:24:33 - DEBUG - [Socket] Creating socket...
20:24:33 - DEBUG - [Socket] Socket created.
20:24:33 - DEBUG - [Socket] Binding to localhost:3000...
20:24:33 - DEBUG - [Socket] Bound to localhost:3000.
20:24:33 - DEBUG - [Accepter] Waiting for client...
20:24:33 - INFO - Listening on port 3000.
Enter `s` to print stats, or `q` to exit.
20:24:33 - DEBUG - [Socket] Accepting client...
20:25:54 - DEBUG - [Socket] Client accepted from 127.0.0.1:47506.
20:25:54 - DEBUG - [Socket] Creating socket...
20:25:54 - DEBUG - [Socket] Socket created.
20:25:54 - DEBUG - [ClientHandler:0] Started.
20:25:54 - DEBUG - [Accepter] Waiting for client...
20:25:54 - DEBUG - [Socket] Accepting client...
20:25:54 - INFO - File audio.mp3 downloaded from 127.0.0.1:47506.
20:25:54 - DEBUG - [ClientHandler:0] Finished.
```

Figura 4: Vista del servidor

4.3. Caso 3: Listar archivos

- **Objetivo:** Permite al cliente ver información de los archivos disponibles para descargar.
- **Descripción del flujo:**
  1. El cliente establece la conexión con el servidor.
  2. El cliente solicita la lista de archivos disponibles con la información.
  3. El servidor envía la lista de archivos.
- **Capturas de pantalla:**

```
./list-files -H "localhost" -p 3000 -v -a
20:28:59 - DEBUG - [Socket] Creating socket...
20:28:59 - DEBUG - [Socket] Socket created.
20:28:59 - DEBUG - [Socket] Connecting to localhost:3000...
20:28:59 - DEBUG - [Socket] Connected to localhost:3000.
Archivos disponibles (3):
> [13-May-2021 (00:48:00)] audio.mp3 - 15.24 MB
> [13-May-2021 (20:22:29)] lorem_ipsum.txt - 97.97 KB
> [13-May-2021 (19:06:01)] text.txt - 0.00 B
20:28:59 - DEBUG - [Socket] Closing socket...
20:28:59 - DEBUG - [Socket] Socket closed.
```

Figura 5: Vista del cliente

```
./start-server -H "localhost" -p 3000 -v
20:28:52 - DEBUG - [Socket] Creating socket...
20:28:52 - DEBUG - [Socket] Socket created.
20:28:52 - DEBUG - [Socket] Binding to localhost:3000...
20:28:52 - DEBUG - [Socket] Bound to localhost:3000.
20:28:52 - DEBUG - [Accepter] Waiting for client...
20:28:52 - DEBUG - [Socket] Accepting client...
20:28:52 - INFO - Listening on port 3000.
Enter `s` to print stats, or `q` to exit.
20:28:59 - DEBUG - [Socket] Client accepted from 127.0.0.1:47518.
20:28:59 - DEBUG - [Socket] Creating socket...
20:28:59 - DEBUG - [Socket] Socket created.
20:28:59 - DEBUG - [ClientHandler:0] Started.
20:28:59 - DEBUG - [Accepter] Waiting for client...
20:28:59 - DEBUG - [Socket] Accepting client...
20:28:59 - INFO - Files list downloaded from 127.0.0.1:47518.
20:28:59 - DEBUG - [ClientHandler:0] Finished.
```

Figura 6: Vista del servidor

4.4. Ejecución de múltiples consultas

Se muestran a continuación capturas de pantalla evidenciando el uso de la arquitectura para manejar múltiples consultas.

```
~/Facultad/2021 1C/ID/TPs/TP1/src  main ±  ./start-server -p 8080
20:06:39 - INFO - Listening on port 8080.
Enter `s` to print stats, or `q` to exit.
20:06:42 - INFO - Files list downloaded from 186.22.17.220:32797.
20:07:49 - INFO - File algo.pdf uploaded from 152.170.95.236:39028.
20:08:28 - INFO - File testita.md downloaded from 186.22.17.220:32807.
20:10:06 - INFO - Files list downloaded from 152.170.95.236:39046.

==========================================
=                  STATS                 =
==========================================

> Run time: 0:04:38.372077
> Established connections: 4
> Requests:
  * upload-file: 1
  * download-file: 1
  * list-files: 2
> Bytes transferred:
  * Sent: 15.24 MB
  * Received: 17.46 MB
> Files:
  * Uploads: 1
  * Downloads: 1
==========================================
```

Figura 7: Vista de un servidor que atendió distintas consultas

```
tony@tony-Surface-Book-2:~/Desktop/Intro/tps/tp1_intro_distri/src$ ./lis
t-files -v -H localhost -p 8080
20:30:23 - DEBUG - [Socket] Creating socket...
20:30:23 - DEBUG - [Socket] Socket created.
20:30:23 - DEBUG - [Socket] Connecting to localhost:8080...
20:30:23 - DEBUG - [Socket] Connected to localhost:8080.
Archivos disponibles (1):
> [13-May-2021 (20:29:46)] Computer-Networking-A-Top-Down-Approach-7th-e
dition.pdf - 17.46 MB
20:30:23 - DEBUG - [Socket] Closing socket...
20:30:23 - DEBUG - [Socket] Socket closed.
tony@tony-Surface-Book-2:~/Desktop/Intro/tps/tp1_intro_distri/src$ ./upl
oad-file -H localhost -p 8080 -s Computer-Networking-A-Top-Down-Approach
-7th-edition.pdf -n Computer-Networking-A-Top-Down-Approach-7th-edition.
pdf
20:30:25 - INFO - Uploading file...
[====================>] 100% (17 MB/17 MB)
20:30:25 - INFO - File uploaded.
tony@tony-Surface-Book-2:~/Desktop/Intro/tps/tp1_intro_distri/src$ ./lis
t-files -v -H localhost -p 8080
20:30:27 - DEBUG - [Socket] Creating socket...
20:30:27 - DEBUG - [Socket] Socket created.
20:30:27 - DEBUG - [Socket] Connecting to localhost:8080...
20:30:27 - DEBUG - [Socket] Connected to localhost:8080.
Archivos disponibles (1):
> [13-May-2021 (20:30:25)] Computer-Networking-A-Top-Down-Approach-7th-e
dition.pdf - 17.46 MB
20:30:27 - DEBUG - [Socket] Closing socket...
20:30:27 - DEBUG - [Socket] Socket closed.
```

Figura 8: Vista de un cliente que realiza diversas consultas

```
tony@tony-Surface-Book-2:~/Desktop/Intro/tps/tp1_intro_distri/src$ ./st
art-server -H localhost -p 8080 -s ./files
20:30:16 - INFO - Listening on port 8080.
Enter `s` to print stats, or `q` to exit.
20:30:23 - INFO - Files list downloaded from 127.0.0.1:41220.
20:30:25 - INFO - File Computer-Networking-A-Top-Down-Approach-7th-edit
ion.pdf uploaded from 127.0.0.1:41222.
20:30:27 - INFO - Files list downloaded from 127.0.0.1:41224.
q

========================================
=                 STATS                =
========================================

> Run time: 0:00:13.830336
> Established connections: 3
> Requests:
  * upload-file: 1
  * download-file: 0
  * list-files: 2
> Bytes transferred:
  * Sent: 197.00 B
  * Received: 17.46 MB
> Files:
  * Uploads: 1
  * Downloads: 0
========================================
```

Figura 9: Vista de un servidor que atiende consultas

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor

La arquitectura Cliente-Servidor es un modelo de diseño en el que dos procesos establecen una comunicación. El proceso que inicia la conexión (normalmente realizando una consulta al servidor) se denomina **Cliente**, mientras que el proceso que escucha consultas, las procesa, y (opcionalmente) responde, es el **Servidor**.

En general, suele haber un host Servidor que ofrece un servicio para resolver consultas de uno o más clientes, a veces incluso de forma simultánea.

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

El protocolo de capa de aplicación es lo que hace posible la comunicación entre dos procesos en primer lugar, puesto que define las reglas para que puedan intercambiar mensajes. En particular define:

- Los tipos de mensajes intercambiados.
- El sintaxis de los tipos de mensajes.
- La semántica de los campos.
- Las reglas que determinan cuando y como cada proceso envía o recibe mensajes.

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

(Ver sección 3.4 *Protocolo de aplicación*)

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

5.4.1. ¿Qué servicios proveen dichos protocolos?

Ambos protocolos proveen servicios de la capa de transporte (siguiendo el modelado de 5 capas de la red), esto quiere decir que proveen a la capa de aplicación una abstracción para el envío de sus mensajes. El servicio de **UDP** solo provee envío de mensaje sin establecer una conexión, mientras que **TCP** provee *full duplex*, esto es, una conexión entre cliente y servidor que permite enviar y recibir datos de forma concurrente, en ambas direcciones.

5.4.2. ¿Cuáles son sus características?

- **UDP:**
  - Provee de los servicios mínimos e indispensables que debe proveer un protocolo de transporte.
  - No es orientado a la conexión.
  - No garantiza ni la correcta llegada (sin errores ni alteración en los bytes) de los mensajes, ni su entrega, ni tampoco que de llegar, la información llegue de forma ordenada.
- **TCP:**
  - Orientado a la conexión: requiere que se establezca una conexión antes de que se puedan comenzar a enviar los segmentos.

- Garantiza la llegada de forma ordenada y correcta de los mensajes.
- Incluye control de congestión, siendo beneficioso para toda la red en general.

5.4.3. ¿Cuando es apropiado utilizar cada uno?

La principal ventaja de UDP es también su principal desventaja: no provee ningún tipo de garantías, por lo que no es confiable. Sin embargo, al no implementar mecanismos que permitan proveer estas garantías, el mismo también es más simple, lo que resulta útil en ciertos casos.

Puede ser utilizado por aplicaciones que no requieran la comprobación inmediata de llegada, como la aplicación de DNS por ejemplo. Es muy útil también en aplicaciones que puedan soportar una cierta tasa de pérdida de paquetes, como son las aplicaciones en tiempo real de vídeo llamadas (Meet, Zoom, y muchas más).

No está de más decir que si bien UDP es un protocolo sin garantías, puede implementarse en la capa de aplicación un mecanismo que provea las garantías necesarias, como es el caso del protocolo QUIC de Google.

TCP, a diferencia de UDP, permite la comunicación instantánea y confiable entre aplicaciones a costo de un sistema más complejo y mayor uso de tráfico de datos por cada mensaje a enviar. A su vez, TCP implementa controles de tráfico, por lo que si bien esto es beneficioso para toda la red, una aplicación podría querer elegir UDP por sobre el mismo para evitar dicha restricción.

Se utiliza principalmente en aplicaciones que no puedan soportar pérdida de paquetes y que necesiten de las garantías que este protocolo provee, como podrían ser aplicaciones de home banking, de e-mail, y demás.

6. Dificultades encontradas

La primera dificultad real que nos surgió durante el desarrollo del presente trabajo práctico ocurrió una vez terminado el mismo, cuando quisimos probarlo entre nosotros, utilizando cada uno su red local. Para esto, debimos *forwardear* el puerto en el servidor, pero cuando lo hicimos, no logramos conectarnos usando los clientes.

Después de mucha investigación, nos dimos cuenta de que esto se debía a que estábamos haciendo el bind a `localhost`, cuando debíamos hacerlo a la dirección que se había forwardeado en el router.

Este error se lo atribuimos a que en materias anteriores, utilizando sockets TCP en C, al levantar el socket usábamos `getaddrinfo(2)`, por lo que el bind funcionaba distinto y no le estábamos pasando de forma directa el host.

Lo pudimos solucionar gracias a pruebas locales, llegando a la conclusión de que hay que usar la dirección para la cual *forwardeamos* el puerto como parámetro.

7. Conclusión

A partir de este trabajo práctico, se puede afirmar que la implementación de una aplicación cliente-servidor de almacenamiento remoto de archivos resulta sencilla cuando se hace sobre el protocolo TCP.

Esto se debe a que las garantías que dicho protocolo nos ofrece (entrega, orden, y completitud) facilitan el diseño del protocolo de aplicación a implementar por sobre este, ya que no es necesario preocuparse por implementar ningún control adicional a la lógica de negocio en la capa de aplicación.

Finalmente, se considera muy formativa la experiencia de diseñar y utilizar una aplicación construida en `Python` ya que permite conocer la interfaz que provee este lenguaje para el manejo de sockets TCP.