Laboratorio 2: Aplicación Servidor

Cátedra de Redes y Sistemas Distribuidos

Objetivos

- Aplicar la comunicación cliente/servidor por medio de la programación de sockets, desde la perspectiva del servidor.
- Familiarizarse con un protocolo de aplicación diseñado en casa.
- Comprender, diseñar e implementar un programa servidor de archivos en Python.

Protocolo HFTP

Llamaremos *Home-made File Transfer Protocol* (HFTP) a un protocolo de transferencia de archivos casero, creado por nosotros específicamente para este laboratorio.

HFTP es un protocolo de capa de aplicación que usa TCP como protocolo de transporte. TCP garantiza una entrega segura, libre de errores y en orden de todas las transacciones hechas con HFTP. Un servidor de HFTP escucha pedidos en el puerto TCP 19500.

Comandos y Respuestas

El cliente HFTP inicia el intercambio de mensajes mediante pedidos o **comandos** al servidor. El servidor envía una **respuesta** a cada uno antes de procesar el siguiente hasta que el cliente envía un comando de fin de conexión. En caso de que el cliente envíe varios pedidos consecutivos, el servidor HFTP los responde en el orden en que se enviaron. El protocolo HFTP es un protocolo ASCII, no binario, por lo que todo lo enviado (incluso archivos binarios) será legible por humanos como strings.

- Comandos**:** consisten en una cadena de caracteres compuesta por elementos separados por un único espacio y terminadas con un fin de línea estilo DOS (`\r\n`)[1]. El primer elemento del comando define el tipo de acción esperada por el comando y los elementos que siguen son argumentos necesarios para realizar la acción.
- Respuestas**:** comienzan con una cadena terminada en `\r\n`, y pueden tener una continuación dependiendo el comando que las origina. La cadena inicial comienza con una secuencia de dígitos (código de respuesta), seguida de un espacio, seguido de un texto describiendo el resultado de la operación. Por ejemplo, una cadena indicando un resultado exitoso tiene código `0` y con su texto descriptivo podría ser `0 OK`.

Comandos	Descripción y Respuesta

[1] Ver End of Line (EOL) en https://en.wikipedia.org/wiki/Newline:
`\r` = CR (Carriage Return) Usado como un carácter de nueva línea en Mac OS.
`\n` = LF (Line Feed) Usado como un carácter de nueva línea en Unix/Mac OS X.
`\r\n` = CR + LF Usado como un carácter de nueva línea en Windows/DOS y varios protocolos.

`get_file_listing`	Este comando no recibe argumentos y busca obtener la lista de archivos que están actualmente disponibles. El servidor responde con una secuencia de líneas terminadas en `\r\n`, cada una con el nombre de uno de los archivos disponible. Una línea sin texto indica el fin de la lista.
	
	`Comando: get_file_listing`
	`Respuesta: 0 OK\r\n`
	`archivo1.txt\r\n`
	`archivo2.jpg\r\n`
	`\r\n`

`get_metadata FILENAME`	Este comando recibe un argumento `FILENAME` especificando un nombre de archivo del cual se pretende averiguar el tamaño[2]. El servidor responde con una cadena indicando su valor en bytes.
	
	`Comando: get_metadata archivo.txt`
	`Respuesta: 0 OK\r\n`
	`3199\r\n`

`get_slice FILENAME OFFSET SIZE`	Este comando recibe en el argumento `FILENAME` el nombre de archivo del que se pretende obtener un *slice* o parte. La parte se especifica con un `OFFSET` (byte de inicio) y un `SIZE` (tamaño de la parte esperada, en bytes), ambos no negativos[3]. El servidor responde con el fragmento de archivo pedido codificado en base64 y un `\r\n`.
	
	`Byte: 0 5 10 15 20 25 30 35 40`
	`v v v v v v v v v`
	`Archivo: !Que calor que hace hoy, pinta una birra!`
	`Comando: get_slice archivo.txt 5 20`
	`Respuesta: 0 OK\r\n`
	`Y2Fsb3IgcXVlIGhhY2UgaG95LCA=\r\n`[4]

`quit`	Este comando no recibe argumentos y busca terminar la conexión. El servidor responde con un resultado exitoso (`0 OK`) y luego cierra la conexión.

[2] Los nombres de archivos no deberán contener espacios, de lo contrario, el protocolo no puede diferenciar si un espacio corresponde al nombre del archivo o al comienzo de un argumento.

[3] Atención que de acuerdo a la codificación ASCII, algunos caracteres fuera del lenguaje Inglés se representan con dos Bytes. En el archivo del ejemplo, de haber usado ¡ en lugar de ! al comienzo de la frase, la respuesta hubiese sido “ calor que hace hoy,” (con espacio al principio en lugar de al final) ya que el carácter ¡ ocupa dos bytes.

[4] Esta es la codificación base64 de “calor que hace hoy, ”. El sentido de utilizar base64 es que al enviar el archivo posiblemente binario, se codifica en una cadena ASCII.

Manejo de Errores

En caso de algún error, el servidor responderá con códigos de respuestas diferentes a `0`, más algún texto descriptivo a definir por el implementador. En particular:

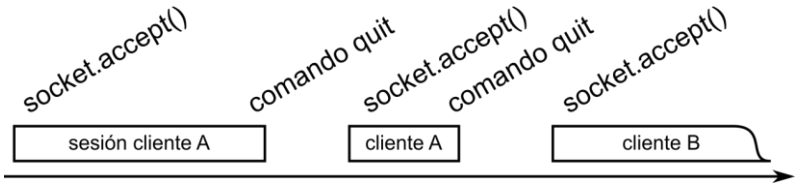
- `0` La operación se realizó con éxito.
- `100` Se encontró un carácter `\n` fuera de un terminador de pedido `\r\n`.
- `101` Alguna malformación del pedido impidió procesarlo[5].
- `199` El servidor tuvo algún fallo interno al intentar procesar el pedido.
- `200` El comando no está en la lista de comandos aceptados.
- `201` La cantidad de argumentos no corresponde o no tienen la forma correcta.
- `202` El pedido se refiere a un archivo inexistente.
- `203` El pedido se refiere a una posición inexistente en un archivo[6].

Los errores con código iniciado en `1` son considerados fatales y derivan en el cierre de la conexión una vez reportados por el servidor. Los errores que inician con `2` permiten continuar con la conexión y recibir pedidos posteriores.

Tarea

Deben diseñar e implementar un **servidor de archivos en Python 3** que soporte completamente el protocolo de transferencia de archivos **HFTP**. El servidor debe ser **robusto** y tolerar comandos incorrectos, ya sea de forma intencional o maliciosa.

El cliente y el servidor a desarrollar podrán estar corriendo en máquinas distintas (sobre la misma red) y el servidor será capaz de manejar varias conexiones a la vez.



A continuación se muestra un ejemplo de ejecución del servidor atendiendo a un único cliente.

```
caro@victoria:~/Ayudantia/Redes$ python server.py
Running File Server on port 19500.
Connected by: ('127.0.0.1', 44639)
Request: get_file_listing
Request: get_metadata client.py
Request: get_slice client.py 0 1868
Closing connection...
```

[5] A diferencia de los errores no fatales 200 y 201, este error es producto de alguna malformación crítica a criterio del implementador. Por ejemplo, un comando malintencionado, de gran longitud, podría provocar un DoS o disminución de performance en el server y podría ser intervenido por un error fatal de este tipo.
[6] Se aplica particularmente al comando `get_slice` y debe generarse cuando no se cumple la condición `OFFSET + SIZE ≤ filesize`.

El servidor debe aceptar en la línea de comandos las opciones:

- `-d` directory para indicarle donde están los archivos que va a publicar.
- `-p` port para indicarle en que puerto escuchar. Si se omite usará el valor por defecto.
- Deben utilizar el comando `telnet <dir IP> <num Port>` para enviar comandos mal formados o mal intencionados y probar la robustez del servidor.

Pasos a seguir

1. **Descargar y preparar el entorno:**
   - Obtener el kickstarter del laboratorio desde el aula virtual.
   - Descomprimirlo con:

     ```
     tar -xvzf kickstart_lab2.tar.gz
     ```
   - El kickstarter incluye:
     - Una estructura base del servidor a completar (**server.py** y **connection.py**).
     - Un archivo con las constantes necesarias (**constants.py**).
     - Un cliente HFTP funcional.
     - Un script de pruebas para el servidor (**server-test.py**).
2. **Configurar un entorno virtual de Python** siguiendo las instrucciones proporcionadas en la documentación.
3. **Ejecutar el laboratorio en su estado inicial** para verificar el funcionamiento del código base.
4. **Modificar el servidor para aceptar conexiones:**
   - Ajustar **server.py** para que acepte conexiones.
   - Al recibir una conexión, debe crear un objeto de tipo **connection**.
   - Probar el funcionamiento usando **telnet**.
5. **Implementar los comandos del protocolo:**
   - Empezar con el comando `quit`.
   - Probar cada comando con **telnet**.
   - Utilizar **client.py** como referencia para manejar las conexiones.
6. **Validar el servidor con el cliente proporcionado:**
   - Una vez implementados los comandos, probar el funcionamiento con **client.py**.
7. **Ejecutar las pruebas automatizadas:**
   - Verificar el manejo de casos de error ejecutando el script **server-test.py**.
8. **Implementar soporte para múltiples clientes utilizando hilos.**
9. **(Opcional - Punto estrella) Implementar múltiples clientes con `poll`.**
   - Pueden utilizar estas referencias:
     - Uso de `poll` para aceptar múltiples clientes en un servidor TCP en C.
     - Cómo usar `poll` en sockets con Python.

Preguntas para responder en el informe:

1. ¿Qué estrategias existen para poder implementar este mismo servidor pero con capacidad de atender *múltiples clientes simultáneamente*? Investigue y responda brevemente qué cambios serían necesarios en el diseño del código.
2. Pruebe ejecutar el servidor en una máquina del laboratorio, mientras utiliza el cliente desde otra, hacia la ip de la máquina servidor. ¿Qué diferencia hay si se corre el servidor desde la IP “localhost”, “127.0.0.1” o la ip “0.0.0.0”?

Tarea Estrella

En caso de *implementar* el servidor capacidad de atender *múltiples clientes simultáneamente con poll,* se otorgarán puntos extras. De acuerdo al funcionamiento del mismo y la capacidad del alumno de explicar lo realizado en la evaluación oral, se podrán dar hasta 2 puntos extras en la 1er evaluación de la defensa de los laboratorios.

Requisitos de la entrega

- Las entregas serán a través del repositorio Git provisto por la Facultad para la Cátedra, con **fecha límite indicada en el cronograma del aula virtual**.
- Junto con el código deberá entregar una presentación (tipo powerpoint) y un video de 10 +/-1 minutos. Les damos una estructura de base como idea, pero pueden modificarla/ ampliarla.

1. **Introducción al proyecto**:
    a. Presenta brevemente el contexto del proyecto y sus objetivos.
    b. Explica la importancia de desarrollar un Protocolo de transferencia de datos.
2. **Responder preguntas como**:
    a. ¿Cómo funciona el paradigma cliente/servidor? ¿Cómo se ve esto en la programación con sockets?
    b. ¿Cual es la diferencia entre Stream (TCP) y Datagram (UDP), desde la perspectiva del socket?
    c. ¿Qué es el protocolo FTP? ¿Para qué sirve?
    d. ¿Qué es base64? ¿Para qué la usamos en el laboratorio?
    e. ¿Qué pasa si queremos enviar un archivo contiene los caracteres \r\n? ¿Cómo lo soluciona esto su código?
3. **Explicar el desarrollo de las funciones principales del servidor**
4. **Errores y dificultades enfrentadas y cómo se resolvieron**
5. **Conclusiones**
    a. Deben agregar un apartado importante aquí mencionando la relación de este lab con el lab anterior de APIS

- Se deberá entregar código con estilo PEP8.

- El trabajo es grupal. Todos los integrantes del grupo deberán ser capaces de explicar el código presentado.
- No está permitido compartir código entre grupos.