Laboratorio: hftp

Cátedra de Redes y Sistemas Distribuidos

Revisión 2015, Matias Molina

Revisión 2012-2014, Carlos Bederián

Revisión 2011, Nicolás Wolovick

Revisión 2010, Daniel Moisset, Eduardo Sánchez

Original 2009, Carolina Dania, Daniel Moisset

Objetivos

- Comprender y realizar un programa servidor de archivos secuencial.
- Dado un protocolo definido de manera rigurosa, estudiarlo e implementarlo.
- Aprender a usar las primitivas de sockets para hacer servidores.
- Entender que las aplicaciones de red son agnósticas respecto a la arquitectura, el sistema operativo y el lenguaje de programación utilizados.
- Comprender la problemática de una aplicación servidor (funcionamiento permanente, robustez, tolerancia a fallas, seguridad, uso apropiado de los recursos), e implementar algunas de estas características.

Motivación

Los protocolos basados en el modelo cliente-servidor son muy comunes en redes. Por lo general, el servidor acepta conexiones simultáneas desde un gran número de clientes (en ciertos casos el número máximo de peticiones puede estar limitado). En este tipo de diseños, normalmente un servidor inicia su tarea esperando por solicitudes de los clientes. Los clientes realizan algún pedido al servidor, éste lo recibe, lo procesa, y envía una respuesta al cliente.

Esta comunicación entre servidor y cliente puede requerir muchos mensajes entre ambos con el fin de procesar debidamente una solicitud. La definición de cuáles son los mensajes que tanto el cliente como el servidor entienden, e incluso la definición del orden de los mismos, está dada por un *protocolo*.

Un **protocolo** es un conjunto de reglas que define qué mensajes son enviados entre los participantes, y en que orden (en este caso, cliente y servidor). Existen infinidad de ejemplos de protocolos en redes para cada una de sus capas, desde la de acceso al medio (802.11 - WLAN, CAN - redes de control del auto, etc.), hasta la capa de aplicaciones (POP3 - lectura de correo, Gopher - precursor de la web, etc.).

Tarea

La aplicación a desarrollar en este laboratorio, consta de un programa servidor y uno cliente, que se pueden ejecutar en el mismo o en diferentes sistemas. El cliente será provisto por la cátedra, y ustedes deberán implementar la parte del servidor.

Para ello, deberán:

1. Implementar el servidor en Python,
2. modularizando adecuadamente el código que desarrollen.

El servidor debe **manejar una conexión por vez** (secuencial), y los clientes podrán realizar las siguientes solicitudes:

- **Listar** todos los archivos que se encuentren disponibles en el servidor.
- Dado un archivo existente en el servidor, obtener los **metadatos** del mismo, que para este laboratorio solo será el *tamaño* del archivo.
- Dado un archivo existente en el servidor, obtener una **fragmento** del mismo. Éste fragmento estará dado por un par *offset, size* (offset es el comienzo de la porción deseada).
- **Interrumpir** la conexión.

Ejemplo

A continuación daremos un ejemplo de la ejecución:

```
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...
```

Definición del protocolo HFTP

El protocolo HFTP es un protocolo de capa de aplicación, definido en casa (**H**ome-made **F**ile **T**ransfer**P**rotocol). Funciona usando como capa de transporte el protocolo TCP de internet.

El servidor debe escuchar permanentemente, atendiendo conexiones en el puerto TCP `19500`. Un cliente tiene permitido conectarse a este puerto y enviar pedidos.

Todos los pedidos del cliente consisten en una cadena terminada en un fin de línea estilo DOS (`"\r\n"`). Todo el tráfico del **cliente al servidor** debe consistir de **pedidos**. Todo el tráfico del **servidor a un cliente** se realiza en **respuesta** a un pedido.

El intercambio de mensajes lo inicia el cliente. El servidor contesta cada pedido antes de procesar el siguiente pedido. Esto se repite de manera intercalada, hasta que el cliente envía un pedido de fin de conexión. Es legal que el cliente envíe varios pedidos consecutivos, en cuyo caso el servidor debe contestarlos en el orden que se enviaron.

El texto de un pedido está compuesto por elementos separados por un único espacio. El carácter `'\n'` no puede aparecer en un pedido excepto en el terminador `'\r\n'`. Si el servidor recibe un `'\n'` debería devolver un código de error `100` (ver manejo de errores más abajo) e interrumpir la conexión. El primer elemento del pedido es llamado **comando**, y debe ser uno de los siguientes:

- `get_file_listing`
- `get_metadata FILENAME`
- `get_slice FILENAME OFFSET SIZE`
- `quit`

Si el servidor recibe un pedido cuyo comando no es uno de los anteriores debería devolver un código de error `200`, y no realizar más acciones con el pedido.

Los elementos que siguen al comando son los **argumentos**. Cada uno de los comandos anteriores acepta una cantidad *fija* de argumentos (notar que algunos comandos reciben 0 argumentos). Si el servidor recibe un pedido con una cantidad incorrecta de argumentos para el comando solicitado, debe contestar un código de error `201` y no seguir procesando el pedido.

Todas las respuestas del servidor al cliente comienzan con una cadena terminada en `\r\n`, y pueden tener una continuación dependiendo el comando. La cadena inicial siempre comienza con una secuencia de dígitos, seguida de un espacio, seguido de un texto describiendo el resultado de la operación. Este texto es a elección del implementador, pero debería ser descriptivo del problema encontrado. Los valores numéricos posibles se dividen en 3 grupos (`0`, `1xx`, `2xx`) y son:

`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.
`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 al comando, o los argumentos 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.

Luego de enviar un código de error de tipo `1xx`, el servidor debería cerrar la conexión hacia el cliente (estos errores se consideran "fatales"). Si el servidor puede atender un pedido correctamente, entonces debe contestar `0`. Si el servidor contesta un valor distinto de `0`, no debería hacer modificaciones en su estado ni el del sistema más allá del envío de mensaje de estado, y posible cierre de la conexión (si y solo si el comando fuese `quit`). Si el servidor responde un código de tipo `2xx`, la conexión debe permanecer abierta y recibir pedidos posteriores.

Descripción de cada comando

`get_file_listing`

El comando `get_file_listing` recibe 0 argumentos. Si termina correctamente (con estado 0), luego de la línea de estado, se envía una secuencia de líneas terminadas en `\r\n`, cada una con un nombre de archivo disponible a través del servicio. Luego de esta secuencia de líneas, sigue una línea sin más texto que el terminador `\r\n`, para indicar el fin de la respuesta.

Debajo damos como ejemplo una conversación entre `C:`liente y `S:`ervidor, mostrando explícitamente los caracteres especiales de retorno del carro y fin de línea.

```
C: get_file_listing\r\n
S: 0 OK\r\n
S: archivo1.txt\r\n
S: archivo2.jpg\r\n
S: \r\n
```

`get_metadata`

El comando `get_metadata` toma un argumento que debería ser un nombre de archivo. Si este archivo

no está disponible el servidor debería retornar un error `202`. Si está disponible, el servidor debe responder luego de la línea de encabezado una línea terminada en `\r\n` con una secuencia de dígitos indicando la longitud del archivo en decimal.

```
C: get_metadata archivo1.txt\r\n
S: 0 OK\r\n
S: 3199\r\n

C: get_metadata noexiste.txt\r\n
S: 202 No existe ese archivo\r\n
```

`get_slice`

El comando `get_slice` recibe tres argumentos. Los dos últimos deben ser secuencia de dígitos (si no lo son el servidor debe dar una respuesta `201`). El primero debe indicar un archivo disponible (de no serlo, corresponde una respuesta `202`). Los dos argumentos restantes son el *offset* y el *size*. Si el offset es mayor o igual a la longitud del archivo, o el offset sumado al size es mayor a la longitud del archivo, el servidor devolverá un error `203`. Notar como su comportamiento es similar al `man 2 read`, salvo que no se puede leer una parte por dentro y otra por fuera del archivo, es decir cuando vale $offset \leq filesize < offset+size$.

Luego de la línea de encabezado, la respuesta al comando `get_slice` está formada por una secuencia de 1 o más **fragmentos**. Cada fragmento consiste en:

1. Una secuencia de dígitos indicando en base 10 la longitud de la cadena de datos del fragmento, en bytes.
2. Un espacio en blanco.
3. Una cadena de datos que debe ser exactamente de la cantidad de bytes indicada al principio del fragmento.
4. Un terminador `\r\n`.

El último fragmento debe tener siempre longitud 0, y todos los fragmentos anteriores deben tener tamaño positivo. Nótese que dentro de la cadena de datos puede haber `\r` y/o `\n` intercalados, que no serán tomados como delimitadores. La concatenación de las cadenas de datos contenidas en los fragmentos, en el orden que son transmitidos, debería ser el contenido del archivo a partir del *offset* (donde como es de esperar el offset 0 corresponde al primer byte del archivo), y tener a lo sumo *size* bytes. El servidor debe intentar enviar *size*bytes, a menos que encuentre que el archivo termina o se produzca algún error de lectura.

```
C: get_slice archivo1.txt 10 30\r\n
S: 0 Todo ok\r\n
S: 10 r de La Ma\r\n
S: 20 ncha de cuyo nombre \r\n
S: 0 \r\n
```

`quit`

El comando `quit` recibe 0 argumentos. La respuesta a este comando debe ser siempre `0`, y luego de enviarla debe cerrarse la conexión al cliente.

```
C: quit\r\n
S: 0 Listo!\r\n
```

Otros requisitos

- El servidor debe aceptar en la línea de comandos las **opciones**:
  - `-d directory` para indicarle donde estan los archivos que va a publicar.
  - `-p port` para indicarle en que puerto escuchar. Si se omite debería usarse el valor por defecto que se indica en el protocolo.
- `get_file_listing` debe devolver **datos actualizados** si se copian o borran archivos en el directorio de datos mientras el servidor corre.
- El servidor debe funcionar con **archivos muy grandes** (varios gigabytes, por ejemplo). Esto implica que no van a poder leer todo el archivo en memoria. En Python esto puede solucionarse usando*generadores*.
- El servidor **no deberá interrumpirse nunca** salvo que ocurran eventos excepcionales que no puedan ser solucionados (fallas de memoria, fallas en los algoritmos y las estructuras de datos), o bien que el proceso que lo contiene sea finalizado por el sistema operativo subyacente.

Recomendaciones

- Tener en cuenta al hacer un `recv()` que puede venir más de un pedido, o menos de un pedido (la red puede partir o unir mensajes). Van a tener que pensar **mecanismos de buffering** que soporten eso.
- Al codificar el server, tengan en cuenta lo que puede pasar si el cliente (por error o intención) no respeta el protocolo. Traten de que el servidor sea **robusto** en esos casos.

Requisitos del código a entregar

- Las entregas serán a través del repositorio Git provisto por la Facultad para la Cátedra.
- Junto con el código, se deberá entregar un informe en el cual se explique detalladamente la estructuración del servidor, las decisiones de diseño tomadas, dificultades con las que se encontraron, y cómo las resolvieron. El informe debe estar en escrito en texto plano, Markdown o reStructuredText.
- 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.
- El plazo de entrega es hasta el **Lunes 30 de Marzo a las 23:59**.