Universidad Rey Juan Carlos

GSyC

Práctica 4 - Python, sockets UDP y Registrar SIP

Protocolos para la Transmisión de Audio y Vídeo en Internet

Versión 9.0 – 18.10.2019

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas. Para las instrucciones de entrega, mira al final del documento. Para la evaluación de esta entrega se valorará el correcto funcionamiento de lo que se pide, el seguimiento de la guía de estilo de Python y el correcto uso (y entrega) con git en GitLab.

1 Introducción

`socketserver` es un módulo en Python que simplifica la tarea de implementar servicios en Internet. Aunque hay cuatro tipos diferentes de servidores básicos, nosotros en esta práctica utilizaremos solamente `UDPServer`, que utiliza datagramas – paquetes discretos que pueden llegar desordenados, incluso se pueden perder por el camino (nótese que esto es en contraposición con `TCPServer` que trabaja con flujos TCP). `UDPServer` trabaja de manera secuencial, lo que significa que las peticiones serán atendidas de una en una, por lo que tendrán que esperar si hay una petición en proceso.

2 Objetivos de la práctica

- Manejar SIP de manera sencilla.
- Crear un esquema cliente-servidor en Python.

3 Conocimientos previos necesarios

1. Nociones de Python (las de las primeras prácticas) y de orientación a objetos.
2. Nociones de SIP (las vistas en clase de teoría)

Tiempo estimado (para un alumno medio): 10 horas

4 Ejercicios

1. Esta práctica requiere hacer capturas con `wireshark`. Para que el administrador de sistemas te incluya en el grupo con permisos para poder hacerlo en las máquinas del laboratorio, abre una *shell* e introduce esta instrucción[1]:

   ```
   $ touch $HOME/.ptavi1819
   ```

2. Con el navegador, dirígete al repositorio `ptavi-p4` en la cuenta de la asignatura en GitLab[2] y realiza un `fork`, de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona el repositorio que acabas de crear a local para poder editar los archivos. Trabaja a partir de ahora en ese repositorio, sincronizando (haciendo commit) los cambios que vayas realizando según los ejercicios que se comentan a continuación.

3. Estudia el código de un sencillo cliente UDP en `client.py` que encontrarás en tu repositorio local. Fíjate en que:

   (a) Importa el módulo `socket`.

   (b) Inicializa varias *variables* constantes (nota que al ser constantes, vienen en mayúsculas, como marca la convención correspondiene en PEP8).

   (c) Crea un `socket`.

   (d) Se conecta con un servidor.

   (e) Envía una **secuencia de bytes** por el `socket` con el método `send()` y lee del `socket` con `recv()`.

   (f) Al recibir con `recv()`, indicamos el valor del *buffer* en bytes.

   (g) No hace falta cerrar explícitamente la conexión, ya que se hace automáticamente al salir del contexto del `with`.

4. Estudia el código de `server.py`, que implementa un servidor de respuesta basado en UDP. Fíjate en que:

   (a) Importa el módulo `socketserver`[3].

   (b) Tenemos una única clase que manejará las peticiones.

[1] Esta instrucción crea un archivo vacío en tu cuenta. El administrador de sistema buscará por este archivo para incluir a los usuarios en el grupo con permisos para capturar con `wireshark`.

[2] `http://gitlab.etsit.urjc.es/ptavi/ptavi-p4`

[3] `https://docs.python.org/3/library/socketserver.html`

(c) Esta clase hereda de una clase `DatagramRequestHandler` que hay en el módulo `socketserver`.

(d) La clase no tiene constructor `__init__`; utiliza el de la clase padre.

(e) La clase sólo tiene un método, llamado `handle()`.

(f) El método `handle()` se ejecuta cada vez que recibimos una petición en el servidor.

(g) `self.wfile` y `self.rfile` son los atributos que abstraen el `socket` (como si fuera un fichero):

- Podemos leer del mismo, iterando sobre `self.rfile` como si fuera un fichero de lectura.
- Podemos escribir en el mismo con `self.wfile.write()`.

(h) Enviamos y recibimos **secuencias de bytes**.

(i) Un programa principal, donde se instancia la clase `EchoHandler`, indicando la IP y el puerto donde se deja al servidor escuchando en un bucle infinito (del que sólo se puede salir desde el terminal con `Ctrl+C`, que lanza una excepción `KeyboardInterrupt`.

5. Cambia el *script* del cliente para que:

   - Se pase como parámetro al *script* la IP y el puerto del servidor, así como a continuación el mensaje que se ha de enviar. Nota que la *shell* el mensaje a enviar nos lo dará como `string` y que tendremos que transformarlo en una secuencia de bytes. Para ejecutar el cliente, deberíamos hacer:

     ```
     $ python3 client.py ip puerto linea
     ```

     Una ejemplo de llamada sería:

     ```
     python3 client.py 127.0.0.1 5060 eco eco, soy yo
     ```

   [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

6. Modifica el *script* del servidor para que:

   - Imprima por pantalla la IP y el puerto del cliente (esta información viene en el atributo `client_address` en forma de tupla[4]).
   - Se pase el puerto al que ha de escuchar como parámetro al *script*.

   [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

[4]Las tuplas son listas especiales, ya que son inmutables. Se definen con paréntesis, no con corchetes.

7. Modifica los *scripts* anteriores para tener un cliente que envíe mensajes de registro SIP y un servidor `Registrar` SIP. Cada vez que el cliente le mande una línea con el método `REGISTER` (en mayúsculas), el servidor guardará la dirección registrada y la IP en un diccionario[5]. Cambia el nombre de la clase del servidor de `EchoHandler` a `SIPRegisterHandler`. La petición SIP del cliente tendrá una pinta parecido a lo siguiente:

   ```
   REGISTER sip:luke@polismassa.com SIP/2.0\r\n\r\n
   ```

   El servidor deberá responder con un mensaje de este estilo:

   ```
   SIP/2.0 200 OK\r\n\r\n
   ```

   El cliente se deberá seguir ejecutándose desde línea de comando, ahora de la siguiente manera:

   ```
   $ python3 client.py ip puerto register luke@polismassa.com
   ```

   [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

8. Añade funcionalidad al cliente y al servidor para ofrecer la posibilidad de darse de baja a un usuario. Para ello, hay que añadir una cabecera `Expires` (cuyos valores vienen dados en segundos). En caso de que el valor de la cabecera `Expires` sea 0, el usuario será borrado del diccionario de registro; en cualquier otro caso, siempre que el tiempo sea positivo, el valor de `Expires` será el tiempo de expiración en el servidor. En este ejercicio no hace falta considerar funcionalidad de servidor para cuando el registro de un cliente caduca. La petición del cliente tendrá una pinta parecida a ésta:

   ```
   REGISTER sip:luke@polismassa.com SIP/2.0\r\n
   Expires: 0\r\n\r\n
   ```

   El servidor deberá borrar al usuario del diccionario y responder con un

   ```
   SIP/2.0 200 OK\r\n\r\n
   ```

   El cliente se deberá seguir ejecutando desde línea de comando, ahora de la siguiente manera:

[5] Este diccionario ha de ser un atributo de clase no de la instancia, véase `http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide`

```
$ python client.py localhost 2500 register luke@polismassa.com 3600
```

donde 3600 es un ejemplo del tiempo de expiración y será el valor de la cabecera `Expires`. En caso de que no se introduzcan los valores necesarios, se imprimirá el siguiente mensaje:

```
Usage: client.py ip puerto register sip_address expires_value
```

[Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

9. Modifica el servidor para añadir el método `register2json` en el que se implemente la siguiente funcionalidad: cada vez que un usuario se registre o se dé de baja, se imprimirá en el fichero `registered.json` con información sobre el usuario, su dirección y la hora de expiración. A continuación, se puede ver un ejemplo[6]:

```
{
    [
        "luke@polismassa.com",
        {
            "address": "localhost",
            "expires": "2019-10-16 10:37:12 +0000"
        }
    ]
}
```

Para los formatos de tiempo, se recomienda utilizar el módulo time, en particular: `time()`, que devuelve los segundos desde el 1 de enero de 1970, `gmtime()`, que toma los segundos desde el 1 de enero de 1970 y te lo devuelve en una tupla, y `strptime()`, que representa el tiempo en un `string`. Así,

```
time.strftime('%Y-%m-%d %H:%M:%S', time.gmtime(time.time()))
```

devolverá un `string` con la hora GMT actual. Y

```
time.strftime('%Y-%m-%d %H:%M:%S', time.gmtime(1233213))
```

devolverá un `string` con la hora GMT del segundo 1.233.213 desde el 1 de enero de 1970. En este ejercicio, se ha de implementar funcionalidad para que el servidor `Registrar` gestione la caducidad de los usuarios registrados.

[Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

[6]Tu fichero JSON no tiene que ser *igual* que lo mostrado; simplemente ha de ser un fichero JSON válido con los datos de usuarios registrados legible por humanos (i.e., la fecha ha de ser entendible).

10. Modifica el servidor, añadiendo el método `json2registered`, para que cuando se lance, compruebe si hay un fichero llamado `registered.json`. Si existe, se leerá su contenido y se usará como diccionario de usuarios registrados. Si da *cualquier* error al leer el fichero, el servidor se ejecutará como si el fichero JSON no existiera.

    [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

11. Documenta tu código con `docstrings`[7]. Comprueba asimismo que los nombres de las variables siguen las indicaciones de PEP8. Ten en cuenta que el programa `pep8` no comprueba ninguna de estas dos circunstancias.

    [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

12. Realiza una captura con `wireshark` (interfaz `lo`) con las siguientes interacciones:

    (a) El cliente `marty.mcfly@sk8ing.com` se registra. Tiempo de expiración: 5.

    (b) El cliente `doc@delorean.com` se registra. Tiempo de expiración: 3600.

    (c) (Se dejan pasar unos segundos, más de cinco).

    (d) El cliente `doc@delorean.com` se da de baja.

    (e) El cliente `marty.mcfly@sk8ing.com` se da de baja.

    Investiga tu captura. Comprueba, en particular, que puedes ver el intercambio de mensajes y que todo va en texto claro por la red. Guarda la captura como `register.libpcap` y añádela al repositorio.

    [Al terminar el ejercicio es recomendable hacer `commit` de los ficheros modificados]

    [Al terminar la práctica, realiza un `push` para sincronizar tu repositorio GitLab]

5 ¿Qué deberías tener al finalizar la práctica?

La entrega de práctica se deberá hacer antes del lunes 28 de octubre de 2019 a las 23:55. Para entonces, se debe:

1. Tener un repositorio git en GitLab con:

    - 2 módulos Python y la captura realizada con `wireshark` (y únicamente estos tres ficheros):
        - `server.py`

[7] La recomendación para `docstrings` en Python se puede encontrar en `http://legacy.python.org/dev/peps/pep-0257/`.

  - `client.py`
  - `register.libpcap`
- 1 clase `SIPRegisterHandler` (en `server.py`) con los métodos:
  (a) `handle`
  (b) `register2json`
  (c) `json2register`
- 4 ficheros adicionales (además de `.git`): `README.md`, `LICENSE`, `check-p4.py` y `.gitignore`.

Se han de tener en cuenta las siguientes consideraciones:

- Se valorará que al menos haya diez `commits` realizados, en dos días diferentes.
- Se valorará que el código entregado siga la guía de estilo de Python (véanse PEP8 y PEP257).
- Se valorará que los programas se invoquen correctamente y que muestren los errores correctamente, según se indica en el enunciado de la práctica.

Se puede comprobar la correcta entrega de la práctica utilizando el programa `check-p4.py`. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 check-p4.py login
```

donde `login` es tu nombre de usuario en el laboratorio. El programa comprueba que se han entregado los ficheros que se solicitan (y sólo esos), y si se sigue la guía de estilo PEP8.