Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Principios de Comunicaciones

Proyecto N°1

Sockets

Integrantes: Jean Cherubini
Claudio Urbina
Profesor: Cesár Azurdia
Auxiliares: Sandy Bolufe
Alejandro Cuevas
Ayudante: Nicolas Ortega
Fecha de realización: 22 de abril de 2018
Fecha de entrega: 22 de abril de 2018
Santiago, Chile

Índice de Contenidos

1. Introducción

En el presente informe se presentará la implementación y los resultados obtenidos a partir del proyecto N°1 del curso Principios de Comunicaciones, donde se pide implementar un algoritmo de *servidor* y *cliente*.

Los algoritmos fueron desarrollados utilizando *Python 3.6*[1]. El proyecto tiene como objetivo crear un chat basado en *Sockets* TCP/IP y *Threads*. Se utilizó el código base proporcionado por el cuerpo docente para empezar, y a partir de ahí se modificó para satisfacer los requerimientos:

- Soportar al menos 6 clientes.
- Identificador único para cada cliente.
- Mensajes recibidos deben mostrarse en la consola del servidor.
- Cuando se conecta un cliente, debe avisarse a través de la consola del servidor que un usuario se ha conectado, mostrando su identificación. Además, se debe enviar un mensaje del servidor al cliente y desplegarlo en su consola confirmando la conexión.
- Crear comando de escape "**:q**", de forma que si el cliente lo envía se desconecte el socket.
- Hacer la implementación en base a threads, donde se debe levantar un thread por cada conexión.

[1] https://www.python.org/

2. Marco Teórico

Para poder entender lo realizado, primero es necesario conocer como mínimo los términos *socket* y *thread*.

2.1. Sockets

El socket de red es una abstracción[1] que provee un *endpoint* (nodo de comunicación de red) para la comunicación entre dispositivos en una red, no es algo que "exista" físicamente. Este socket permite enviar y recibir datos en ambos sentidos una vez hecha la conexión.

Los sockets funcionan con un modelo cliente-servidor, en donde el servidor genera un socket de cierto tipo que tiene una familia de direcciones, es decir, un formato de direcciones con las que trabaja (en este caso, direcciones IPv4 o dominios de internet)[2]. Este socket genera un 'bind' (enlace) con la dirección IP del dispositivo y la asocia a un puerto, que es por donde establecerá comunicación con otros sockets.

El socket servidor es capaz de detectar la solicitud de conexión de sockets clientes (que a su vez tienen su propia direccion IP) a un puerto, identificar las direcciones que están asociadas a cada cliente y finalmente aceptarlos. Esto generan un pseudo-socket de forma local que representa al cliente y mediante el cual se pueden enviar y recibir datos. Este último está relacionado directamente con la dirección asociada al cliente detectada durante la aceptación.

Existen diferentes tipos de sockets, pero los más utilizados son:

- Datagram Socket: Es un socket bidireccional (capaz de enviar y recibir información) no orientado a la conexión, ya que utiliza el protocolo UDP para la comunicación y por lo tanto no es seguro que la información llegue al destino de forma íntegra.
- Stream Socket: Es un socket también bidireccional orientado a la conexión, pues utiliza el protocolo TCP, lo cual garantiza la llegada completa de información de un punto a otro.

En el caso particular de este trabajo, se utilizaron sockets de tipo Stream, pues al tratarse de un chat era necesario que los mensajes llegaran de forma correcta y sin pérdidas.

2.2. Threads

Un thread (o hilo) es un subproceso o un conjunto de tareas en específico. Su ventaja es que varios threads pueden ser mantenidos en ejecución de forma prácticamente simultánea y manejarse de forma independiente de los demás. A esto se le llama *multithreading*. En equipos con multiples procesadores, varios threads pueden ser ejecutados de forma realmente simultánea, lo que permitiría una respuesta inmediata y sin retrasos en cada uno de ellos, en caso de ser necesario.

La ventaja de estos subprocesos radica en que pueden existir en el contexto de un solo proceso mayor y compartir los recursos de dicho proceso, pero además son capaces de ejecutarse de forma independiente.

3. Implementación del Servidor-Cliente

A continuación se detallará como fueron implementados los códigos para el cliente y el servidor. El lenguaje utilizado fue *Python 3.6* y se hizo uso de las siguientes librerías nativas de este lenguaje: *threading*[2], *socket*[3] *y sys*[4].

3.1. Implementación del Servidor

Para el diseño del servidor se utilizó la base proporcionada en material docente. El cual permitía la conexión de un solo cliente, además, este no podía desconectarse sin botar la conexión.

Dado que el servidor fue construido de las funciones nativas de *socket*, se estudió la documentación de esta librearía, donde se pudo percatar que algunas funciones detenían el código hasta que fueran accionadas, este fue el caso de *socket.accept()* y *socket.recv()*. Estas funciones son fundamentales en el código, dado que la primera es la que acepta que un cliente pueda conectarse al servidor, mientras que la segunda, recibe la información enviada por estos mismos. Dado que la conexión de nuevos clientes y la recepción de información puede ser en cualquier momento de tiempo se entiende que estas funciones no pueden interrumpir el proceso de lectura del algoritmo, es decir, si el código se detuviera esperando que un nuevo cliente se conecte, no se podría obtener la información de otro al mismo tiempo hasta que esta acción ocurra.

A partir de los motivos anteriormente señalados, se entendió la necesidad de utilizar la librería de *threading*. De esta manera, la funciones mencionadas pueden esperar a ser accionadas de manera paralela sin interrumpir a otras ni el resto del código en sí. Es por esto que se crearon dos clases en base a threading: para aceptar clientes y para recibir la información de estos. Estas son las clases *clientThread* y *acceptThread*.

La clase *acceptThread* recibe como parámetro el servidor (para aceptar clientes para dicho servidor y no otro). Cada *'accept'* es un thread per sé y como tal, queda de forma paralela esperando la conexión de un nuevo cliente. Cuando esto ocurre imprime un mensaje de conexión en consola (que incluye la identificación del cliente), crea un nuevo thread de la clase *clientThread* (con la información del cliente aceptado) y lo acciona.

```
1 class acceptThread(threading.Thread):
2       def __init__(self,serverSocket):
3         threading.Thread.__init__(self)
4         self.serverSocket=serverSocket
5       def run(self):
6         cliente,direccion=self.serverSocket.accept()
7         print("[SERVER] Client %s connected!" % str(cliente.getpeername()))
8         usuario=clientThread(cliente,direccion)
9         usuario.start()
```

Por otra parte, la clase *clientThread* recibe como parámetros la información de socket del cliente y su dirección. Este cliente, al igual que accept es un thread per sé, por lo que al momento de

[2] https://docs.python.org/3/library/threading.html
[3] https://docs.python.org/3/library/socket.html
[4] https://docs.python.org/3/library/sys.html

ejecutarse *cliente.start()* se accionara la función *run()* que se mantendrá en un loop infinito (de forma paralela) esperando recibir mensajes que el cliente envíe al servidor. Así, cada vez que el cliente envíe un mensaje se imprimirá en la consola, si este mensaje es *':q'* el cliente se desconecta del servidor deteniéndo el socket de este. Cada vez que un cliente se desconecta, un nuevo thread de accept es activado, de esta manera, un nuevo cliente puede ocupar su lugar.

```
class clientThread(threading.Thread):
  def __init__(self, socket, direccion):
    threading.Thread.__init__(self)
    self.socket = socket
    self.direccion = direccion
  def run(self):
    while True:
      message=self.socket.recv(MSG_BUFFER)
      if message.decode('ascii') == ':q':
        print("[SERVER] Client %s disconnected!" % str(self.socket.getpeername()))
        self.socket.shutdown(socket.SHUT_RDWR)
        self.socket.close()
        break
      else:
        print('<'+str(self.socket.getpeername())+'>: '+ message.decode('ascii'))
    accept = acceptThread(serverSocket)
    accept.start()
```

Finalmente, el *main* del servidor contiene la creación de este: host, puerto, tipo y cantidad de clientes a escuchar. También, imprime la información del servidor y posteriormente crea y acciona 6 threads de la clase *acceptThread*. De esta manera el servidor puede tener hasta un máximo de 6 clientes conectados simultáneamente.

```
# Message Buffer size
MSG_BUFFER = 1024

# Obtaining the arguments using command line
try:
    HOST = sys.argv[1]
except:
    HOST = 'localhost'
try:
    PORT = int(sys.argv[2])
except:
    PORT = 8889

# Creating the client socket. AF_INET IP Family (v4)
# and STREAM SOCKET Type.
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((HOST, PORT))
serverSocket.listen(6) # Listen one client clients

print('Client connected to %s:%s' % (HOST, PORT))
accept = acceptThread(serverSocket)
accept2 = acceptThread(serverSocket)
accept3 = acceptThread(serverSocket)
```

```
    accept4 = acceptThread(serverSocket)
    accept5 = acceptThread(serverSocket)
    accept6 = acceptThread(serverSocket)
    accept.start()
    accept2.start()
    accept3.start()
    accept4.start()
    accept5.start()
    accept6.start()
    while True:
      pass
```

3.2. Implementación del Cliente

Para el diseño de este algoritmo también se utilizó como base el código otorgado en material docente. Éste, básicamente define el cliente como socket y lo conecta al servidor mediante el host y el puerto. Dentro de las modificaciones realizadas al código original esta la utilización de *str.encode()* esto debido a que el algoritmo proporcionado fue realizado en python 2, en el cual la información enviada desde un cliente al servidor podía ser un string, pero, en python 3 la información solo puede ser en *bytes*. Por otra parte, dentro de los requisitos del proyecto, se pide que el cliente pueda desconectarse del servidor escribiendo la combinación *':q'*. Finalmente, basto con analizar el mensaje escrito, si este correspondía a la combinación antes mencionada, la consola imprime un mensaje de despedida y detiene el socket del cliente para luego cerrarlo y desconectarlo del servidor.

```
import sys
import socket

clientSocket = socket.socket()
try:
    host = sys.argv[1]
except:
    host = 'localhost'
try:
    port = sys.argv[2]
except:
    port = 8889

# Connecting
clientSocket.connect((host, port))
print('Welcome to the best chat in the universe!')

while True:
    message = str(input('You: '))
    clientSocket.send(str.encode(message))
    if message == ':q':
     print('Goodbye!')
     clientSocket.shutdown(socket.SHUT_RDWR)
     clientSocket.close()
     break
```

4. Conclusión

La existencia de sockets es fundamental en comunicaciones de cualquier tipo, la gran documentación que poseen las librerías de socket lo transforman en una materia sencilla de utilizar (y programar), lo cual permite que diversas áreas puedan aplicarlo. Además, posee ventajas como:

1. Optimización de Ancho de Banda.
2. Compatibilidad con distintos lenguajes de programación.
3. Compatibilidad con distintos sistemas operativos.

En el proyecto se logró cumplir con los requisitos y objetivos del proyecto debido a que:

- Se comprendió como trabajan las librerías de socket y threads.
- Se pudo establecer una comunicación servidor-cliente con los siguientes requisitos.
    1. El servidor es capaz de soportar 6 clientes conectados simultáneamente.
    2. El servidor otorga una identificación única para cada cliente.
    3. El servidor muestra los mensajes recibidos en consola.
    4. El servidor reconoce y muestra en consola cuando un cliente nuevo se conecta a este.
    5. El cliente reconoce y muestra en consola cuando se conecta a un servidor.
    6. El cliente puede desconectarse en cualquier momento del servidor utilizando el comando de escape ':q'.
    7. Se utilizó la librería de Threads para paralelizar las procesos de conexión en el servidor.

Finalmente, se determinó que trabajar sockets en base a threads es mas cómodo cuando se requiere una mayor cantidad de clientes conectados, pues la conexión y desconexión de nuevos clientes no interfiere con los mensajes en que son enviados al servidor.

Referencias

[1] Clase_Socket_I.pdf - Material Docente, EL4005-1 Principios de Comunicaciones, FCFM, Otoño 2018

[2] The Python Standard Library, 18. Interprocess Communication and Networking https://docs.python.org/3/library/socket.html

5. Anexo

5.1. Servidor

```
import socket
import sys
import threading

class clientThread(threading.Thread):
  def __init__(self, socket, direccion):
    threading.Thread.__init__(self)
    self.socket = socket
    self.direccion = direccion
  def run(self):
    while True:
      message=self.socket.recv(MSG_BUFFER)
      if message.decode('ascii') == ':q':
        print("[SERVER] Client %s disconnected!" % str(self.socket.getpeername()))
        self.socket.shutdown(socket.SHUT_RDWR)
        self.socket.close()
        break
      else:
        print('<'+str(self.socket.getpeername())+'>: '+ message.decode('ascii'))
    accept = acceptThread(serverSocket)
    accept.start()

class acceptThread(threading.Thread):
  def __init__(self,serverSocket):
    threading.Thread.__init__(self)
    self.serverSocket=serverSocket
  def run(self):
    cliente,direccion=self.serverSocket.accept()
    print("[SERVER] Client %s connected!" % str(cliente.getpeername()))
    usuario=clientThread(cliente,direccion)
    usuario.start()


# Message Buffer size
MSG_BUFFER = 1024

# Obtaining the arguments using command line
try:
    HOST = sys.argv[1]
except:
    HOST = 'localhost'
try:
    PORT = int(sys.argv[2])
except:
    PORT = 8889

# Creating the client socket. AF_INET IP Family (v4)
```

```
# and STREAM SOCKET Type.
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((HOST, PORT))
serverSocket.listen(6) # Listen one client clients

print('Client connected to %s:%s' % (HOST, PORT))
accept = acceptThread(serverSocket)
accept2 = acceptThread(serverSocket)
accept3 = acceptThread(serverSocket)
accept4 = acceptThread(serverSocket)
accept5 = acceptThread(serverSocket)
accept6 = acceptThread(serverSocket)
accept.start()
accept2.start()
accept3.start()
accept4.start()
accept5.start()
accept6.start()
while True:
  pass
```

5.2. Cliente

```
import sys
import socket

clientSocket = socket.socket()
try:
    host = sys.argv[1]
except:
    host = 'localhost'
try:
    port = sys.argv[2]
except:
    port = 8889

# Connecting
clientSocket.connect((host, port))
print('Bienvenido al mejor chat del universo!')

while True:
    message = str(input('Tu: '))
    clientSocket.send(str.encode(message))
    if message == ':q':
      print('Adios!')
      clientSocket.shutdown(socket.SHUT_RDWR)
      clientSocket.close()
      break
```