Redes y Comunicación II

**Sockets**

2.7.1 Programación de sockets con UDP

En esta sección, escribiremos programas cliente-servidor simples que usan UDP; en la siguiente sección, vamos a escribir programas similares que usen TCP.

Recuerde de la Sección 2.1 que los procesos que se ejecutan en diferentes máquinas se comunican entre sí por enviar mensajes a los sockets. Dijimos que cada proceso es análogo a una casa y el proceso socket es análogo a una puerta. La aplicación reside en un lado de la puerta de la casa; el protocolo de la capa de transporte reside al otro lado de la puerta en el mundo exterior. La aplicación desarrollada tiene el control de todo en el lado de la capa de aplicación del socket; sin embargo, tiene poco control del lado de la capa de transporte.

Ahora echemos un vistazo más de cerca a la interacción entre dos procesos de comunicación que usan UDP sockets. Antes de que el proceso de envío pueda empujar un paquete de datos por la puerta socket, cuando se usa UDP, primero debe adjuntar una dirección de destino al paquete. Después de que el paquete pase a través del remitente socket, Internet usará esta dirección de destino para enrutar el paquete a través de Internet al socket en el proceso de recepción. Cuando el paquete llega al socket de recepción, el proceso de recepción

recuperará el paquete a través del socket, y luego inspeccionará el contenido del paquete y tomará las medidas apropiadas.

Entonces, es posible que ahora se pregunte, ¿qué se incluye en la dirección de destino que se adjunta al paquete?

Como era de esperar, la dirección IP del host de destino es parte de la dirección de destino. Incluyendo la dirección IP de destino en el paquete, los enrutadores en Internet podrán enrutar el paquete a través de Internet al host de destino. Pero debido a que un host puede estar ejecutando muchas procesos de redes de aplicación, cada uno con uno o más sockets, también es necesario identificar el particular el socket en el host de destino. Cuando se crea un socket, se asigna un identificador, llamado número de puerto. Entonces, como era de esperar, la dirección de destino del paquete también incluye el número de puerto del socket.

En resumen, el proceso de envío adjunta al paquete una dirección de destino, que consiste en la dirección IP del host de destino y el número de puerto del socket de destino. Además, como pronto veremos, la dirección de origen del remitente, que consta de la dirección IP del host de origen y el número de puerto del socket de origen: también se adjuntan al paquete. Sin embargo, adjuntar la dirección de origen al paquete normalmente no lo hace el código de la aplicación UDP; en cambio, lo hace automáticamente el subyacente Sistema operativo.

Usaremos la siguiente aplicación cliente-servidor simple para demostrar la programación de sockets para ambos

UDP y TCP:

1. El cliente lee una línea de caracteres (datos) de su teclado y envía los datos al servidor.

2. El servidor recibe los datos y convierte los caracteres a mayúsculas.

3. El servidor envía los datos modificados al cliente.

4. El cliente recibe los datos modificados y muestra la línea en su pantalla

La figura 2.27 destaca la principal actividad relacionada con el socket del cliente y el servidor que se comunican a través del servicio de transporte UDP.

Ahora ensuciémonos las manos y echemos un vistazo al par de programas cliente-servidor para una implementación UDP de esta sencilla aplicación. También proporcionamos un análisis detallado, línea por línea, después de cada programa. Bien comience con el cliente UDP, que enviará un mensaje de nivel de aplicación simple al servidor.

Diagrama

Descripción generada automáticamente

Figura 2.27 La aplicación cliente-servidor usando UDP el servidor para poder recibir y responder al mensaje del cliente, debe estar listo y funcionando, es decir, debe estar ejecutándose como un proceso antes de que el cliente envíe su mensaje. El programa cliente se llama UDPClient.py y el programa servidor se llama UDPServer.py. Con el fin de enfatizar los problemas clave, intencionalmente proporcionamos un código que es mínimo. "Buen código" sin duda tener algunas líneas auxiliares más, en particular para manejar casos de error. Para esta aplicación, tenemos elegido arbitrariamente 12000 para el número de puerto del servidor. UDPClient.py Aquí está el código para el lado del cliente de la aplicación:

from socket import \*

serverName = ’hostname’

serverPort = 12000

clientSocket = socket(AF\_INET, SOCK\_DGRAM)

message = raw\_input(’Input lowercase sentence:’) clientSocket.sendto(message.encode(),(serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048) print(modifiedMessage.decode())

clientSocket.close()

Ahora echemos un vistazo a las diversas líneas de código en UDPClient.py

from socket import \*

El módulo de socket forma la base de todas las comunicaciones de red en Python. Al incluir esta línea, podrá crear sockets dentro de nuestro programa

serverName = ’hostname’

serverPort = 12000

La primera línea establece la variable serverName en la cadena 'hostname'. Aquí, proporcionamos una cadena que contiene la dirección IP del servidor (por ejemplo, "128.138.32.126") o el nombre de host del servidor (por ejemplo, “cis.poly.edu”). Si usamos el nombre de host, se realizará automáticamente una búsqueda de DNS para obtener la dirección IP). La segunda línea establece la variable entera serverPort en 12000.

clientSocket = socket(AF\_INET, SOCK\_DGRAM)

Esta línea crea el socket del cliente, llamado clientSocket. El primer parámetro indica la dirección. familia; en particular, AF\_INET indica que la red subyacente está usando IPv4. (No te preocupes por esto ahora hablaremos de IPv4 en el Capítulo 4.) El segundo parámetro indica que el socket es del tipo SOCK\_DGRAM , lo que significa que es un socket UDP (en lugar de un socket TCP). Tenga en cuenta que no estamos especificando el número de puerto del socket del cliente cuando lo creamos; en cambio, estamos dejando que el funcionamiento del sistema haga esto por nosotros. Ahora que se ha creado la puerta del proceso del cliente, querremos crear un mensaje para enviar a través de la puerta.

message = raw\_input(’Input lowercase sentence:’)

raw\_input() es una función integrada en Python. Cuando se ejecuta este comando, el usuario en el cliente se le solicita con las palabras "Ingrese una oración en minúsculas:" El usuario luego usa su teclado para ingresar una línea, que se pone en la variable mensaje. Ahora que tenemos un socket y un message, querremos enviar el mensaje a través del socket al host de destino

clientSocket.sendto(message.encode(),(serverName, serverPort))

En la línea anterior, primero convertimos el mensaje de tipo cadena a tipo byte, ya que necesitamos enviar bytes en un zócalo; esto se hace con el método encode(). El método sendto() adjunta el destino dirección ( serverName, serverPort ) al mensaje y envía el paquete resultante al socket del proceso, clientSocket . (Como se mencionó anteriormente, la dirección de origen también se adjunta al paquete, aunque esto se hace automáticamente en lugar de explícitamente por el código). Enviar un mensaje de cliente a servidor ¡El mensaje a través de un socket UDP es así de simple! Después de enviar el paquete, el cliente espera recibir datos del servidor.

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

Con la línea anterior, cuando llega un paquete de Internet al socket del cliente, los datos del paquete se coloca en la variable modifiedMessage y la dirección de origen del paquete se coloca en la variable dirección del servidor. La variable serverAddress contiene tanto la dirección IP del servidor como el número de puerto del servidor. El programa UDPClient en realidad no necesita esta información de dirección del servidor, ya que ya conoce la dirección del servidor desde el principio; pero esta línea de Python proporciona el servidor dirección, sin embargo. El método recvfrom también toma el tamaño de búfer 2048 como entrada. (Este tamaño de búfer funciona para la mayoría de los propósitos).

print(modifiedMessage.decode())

Esta línea imprime el mensaje modificado en la pantalla del usuario, después de convertir el mensaje de bytes a cadena. Debe ser la línea original que el usuario escribió, pero ahora en mayúsculas.

clientSocket.close()

Esta línea cierra el socket. El proceso entonces termina.

UDPServer.py

Ahora echemos un vistazo al lado del servidor de la aplicación:

from socket import \*

serverPort = 12000

serverSocket = socket(AF\_INET, SOCK\_DGRAM)

serverSocket.bind((’’, serverPort))

print(”The server is ready to receive”)

while True:

message, clientAddress = serverSocket.recvfrom(2048)

modifiedMessage = message.decode().upper() serverSocket.sendto(modifiedMessage.encode(), clientAddress)

Tenga en cuenta que el comienzo de UDPServer es similar a UDPClient. También importa el módulo de socket, también establece la variable entera serverPort en 12000 y también crea un socket de tipo SOCK\_DGRAM (un enchufe UDP). La primera línea de código que es significativamente diferente de UDPClient es

serverSocket.bind((’’, serverPort))

La línea anterior vincula (es decir, asigna) el número de puerto 12000 al socket del servidor. Así en UDPServer, el código (escrito por el desarrollador de la aplicación) asigna explícitamente un número de puerto al enchufe. De esta manera, cuando alguien envía un paquete al puerto 12000 en la dirección IP del servidor, es el paquete será dirigido a este socket. UDPServer luego ingresa a un ciclo while; el ciclo while permitirá UDPServer para recibir y procesar paquetes de clientes de forma indefinida. En el ciclo while, UDPServer espera para que llegue un paquete.

message, clientAddress = serverSocket.recvfrom(2048)

Esta línea de código es similar a lo que vimos en UDPClient. Cuando un paquete llega al socket del servidor, los datos del paquete se colocan en el message variable y la dirección de origen del paquete se coloca en la dirección de cliente variable. La variable clientAddress contiene tanto la dirección IP del cliente como el número de puerto del cliente. Aquí, UDPServer hará uso de esta información de dirección, ya que proporciona un retorno dirección, similar a la dirección del remitente con el correo postal ordinario. Con esta información de dirección de origen, el servidor ahora sabe hacia dónde debe dirigir su respuesta.

modifiedMessage = message.decode().upper()

Esta línea es el corazón de nuestra sencilla aplicación. Toma la línea enviada por el cliente y, después de convertir el mensaje a una cadena, usa el método upper() para ponerlo en mayúsculas.

serverSocket.sendto(modifiedMessage.encode(), clientAddress)

Esta última línea adjunta la dirección del cliente (dirección IP y número de puerto) al mensaje en mayúscula (después de convertir la cadena en bytes) y envía el paquete resultante al socket del servidor. (Como mencionado anteriormente, la dirección del servidor también se adjunta al paquete, aunque esto se hace automáticamente en lugar de explícitamente por el código.) Internet entregará el paquete a esta dirección de cliente. Después el servidor envía el paquete, permanece en el ciclo while, esperando que llegue otro paquete UDP (de cualquier cliente ejecutándose en cualquier host).

Para probar el par de programas, ejecuta UDPClient.py en un host y UDPServer.py en otro host. Ser asegúrese de incluir el nombre de host o la dirección IP adecuados del servidor en UDPClient.py. A continuación, ejecutas UDPServer.py, el programa de servidor compilado, en el host del servidor. Esto crea un proceso en el servidor que permanece inactivo hasta que algún cliente lo contacta. Luego ejecuta UDPClient.py, el cliente compilado programa, en el cliente. Esto crea un proceso en el cliente. Finalmente, para usar la aplicación en el cliente,

escribe una oración seguida de un retorno de carro.

Para desarrollar su propia aplicación cliente-servidor UDP, puede comenzar modificando ligeramente el cliente o programas de servidor. Por ejemplo, en lugar de convertir todas las letras a mayúsculas, el servidor podría contar el número de veces que aparece la letra s y devolver este número. O puede modificar el cliente para que después de recibir una oración en mayúscula, el usuario puede continuar enviando más oraciones al servidor.

2.7.2 Programación de sockets con TCP

A diferencia de UDP, TCP es un protocolo orientado a la conexión. Esto significa que antes de que el cliente y el servidor puedan comienzan a enviarse datos entre sí, primero deben establecer un protocolo de enlace y establecer una conexión TCP. Un final de la conexión TCP se conecta al conector del cliente y el otro extremo se conecta al conector del servidor.

Al crear la conexión TCP, le asociamos la dirección del socket del cliente (dirección IP y puerto

número) y la dirección del socket del servidor (dirección IP y número de puerto). Con la conexión TCP establecido, cuando un lado quiere enviar datos al otro lado, simplemente coloca los datos en el TCP conexión a través de su enchufe. Esto es diferente de UDP, para el cual el servidor debe adjuntar un destino dirección al paquete antes de soltarlo en el socket.

Ahora echemos un vistazo más de cerca a la interacción de los programas cliente y servidor en TCP. El cliente tiene el trabajo de iniciar el contacto con el servidor. Para que el servidor pueda reaccionar a la solicitud inicial del cliente contacto, el servidor tiene que estar listo. Esto implica dos cosas. Primero, como en el caso de UDP, el servidor TCP debe estar ejecutándose como un proceso antes de que el cliente intente iniciar el contacto. En segundo lugar, el programa del servidor debe tener una puerta especial, más precisamente, un enchufe especial, que da la bienvenida a algún contacto inicial de un proceso de cliente que se ejecuta en un host arbitrario. Usando nuestra analogía casa/puerta para un proceso/socket, vamos a

a veces se refieren al contacto inicial del cliente como "llamar a la puerta de bienvenida".

Con el proceso del servidor en ejecución, el proceso del cliente puede iniciar una conexión TCP con el servidor. Esto es hecho en el programa cliente creando un socket TCP. Cuando el cliente crea su socket TCP, especifica la dirección del socket de bienvenida en el servidor, es decir, la dirección IP del host del servidor y el número de puerto del socket. Después de crear su socket, el cliente inicia un protocolo de enlace de tres vías y establece una conexión TCP con el servidor. El apretón de manos de tres vías, que tiene lugar dentro de la capa de transporte, es completamente invisible para los programas cliente y servidor.

Durante el apretón de manos de tres vías, el proceso del cliente llama a la puerta de bienvenida del servidor proceso. Cuando el servidor "escucha" los golpes, crea una nueva puerta, más precisamente, un nuevo enchufe que está dedicado a ese cliente en particular. En nuestro ejemplo a continuación, la puerta de bienvenida es un socket TCP objeto que llamamos serverSocket; el socket recién creado dedicado al cliente que realiza la conexión se llama connectionSocket. Estudiantes que se encuentran con sockets TCP por primera vez

El tiempo a veces confunde la toma de bienvenida (que es el punto de contacto inicial para todos los clientes que desean para comunicarse con el servidor), y cada socket de conexión del lado del servidor recién creado que es creado posteriormente para la comunicación con cada cliente.

Desde la perspectiva de la aplicación, el socket del cliente y el socket de conexión del servidor están directamente conectado por una tubería. Como se muestra en la Figura 2.28, el proceso del cliente puede enviar bytes arbitrarios a su socket, y TCP garantiza que el proceso del servidor recibirá (a través del socket de conexión) cada byte en el pedido enviado. TCP proporciona así un servicio fiable entre los procesos del cliente y del servidor.

Además, así como las personas pueden entrar y salir por la misma puerta, el proceso del cliente no solo envía bytes pero también recibe bytes de su socket; De manera similar, el proceso del servidor no solo recibe bytes de sino también envía bytes a su socket de conexión.

Usamos la misma aplicación cliente-servidor simple para demostrar la programación de sockets con TCP: El cliente envía una línea de datos al servidor, el servidor pone en mayúsculas la línea y la envía de vuelta al cliente.

La figura 2.29 destaca la principal actividad relacionada con el socket del cliente y el servidor que se comunican a través del servicio de transporte TCP.

from socket import \*

serverName = ’servername’

serverPort = 12000

clientSocket = socket(AF\_INET, SOCK\_STREAM)

clientSocket.connect((serverName, serverPort))

sentence = raw\_input(’Input lowercase sentence:’)

clientSocket.send(sentence.encode())

modifiedSentence = clientSocket.recv(1024)

print(’From Server: ’, modifiedSentence.decode())

clientSocket.close()

Ahora echemos un vistazo a las diversas líneas en el código que difieren significativamente del UDP implementación. La primera línea de este tipo es la creación del socket del cliente.

clientSocket = socket(AF\_INET, SOCK\_STREAM)

Esta línea crea el socket del cliente, llamado clientSocket. El primer parámetro nuevamente indica que la red subyacente está usando IPv4. El segundo parámetro

Diagrama

Descripción generada automáticamente

Figura 2.29

La aplicación cliente-servidor usando TCP indica que el socket es del tipo SOCK\_STREAM , lo que significa que es un socket TCP (en lugar de un UDP enchufe). Tenga en cuenta que nuevamente no estamos especificando el número de puerto del socket del cliente cuando lo creamos; nosotros en cambio, están dejando que el sistema operativo haga esto por nosotros. Ahora la siguiente línea de código es muy diferente de lo que vimos en UDPClient:

clientSocket.connect((serverName, serverPort))

Recuerde que antes de que el cliente pueda enviar datos al servidor (o viceversa) usando un socket TCP, un TCP, primero se debe establecer la conexión entre el cliente y el servidor. La línea anterior inicia el TCP conexión entre el cliente y el servidor. El parámetro del método connect() es la dirección del lado del servidor de la conexión. Después de ejecutar esta línea de código, el protocolo de enlace de tres vías es se realiza y se establece una conexión TCP entre el cliente y el servidor.

sentence = raw\_input(’Input lowercase sentence:’)

Al igual que con UDPClient, lo anterior obtiene una sentence del usuario. La sentence de cadena continúa reunir caracteres hasta que el usuario termine la línea escriba un return de carro. La siguiente línea de código también es muy diferente de UDPClient

clientSocket.send(sentence.encode())

La línea anterior envía la sentence a través del socket del cliente y hacia la conexión TCP. Tenga en cuenta que el programa no crea un paquete y adjunta la dirección de destino al paquete, como era el caso de los sockets UDP. En cambio, el programa cliente simplemente elimina los bytes en la oración de cadena en la conexión TCP. Luego, el cliente espera recibir bytes del servidor.

modifiedSentence = clientSocket.recv(2048)

Cuando los caracteres llegan del servidor, se colocan en la cadena modifiedSentence. Los caracteres continúan acumulándose en la modifiedSentencehasta que la línea termina con un return de carro personaje. Después de imprimir la oración en mayúscula, cerramos el socket del cliente:

clientSocket.close()

Esta última línea cierra el socket y, por lo tanto, cierra la conexión TCP entre el cliente y el servidor. Hace que TCP en el cliente envíe un mensaje TCP a TCP en el servidor (consulte la Sección 3.5). ServidorTCP.py Ahora echemos un vistazo al programa del servidor.

from socket import \*

serverPort = 12000

serverSocket = socket(AF\_INET, SOCK\_STREAM)

serverSocket.bind((’’, serverPort))

serverSocket.listen(1)

print(’The server is ready to receive’)

while True:

connectionSocket, addr = serverSocket.accept()

sentence = connectionSocket.recv(1024).decode()

capitalizedSentence = sentence.upper() connectionSocket.send(capitalizedSentence.encode())

connectionSocket.close()

Ahora echemos un vistazo a las líneas que difieren significativamente de UDPServer y TCPClient. Al igual que con TCPClient, el servidor crea un socket TCP con:

serverSocket=socket(AF\_INET, SOCK\_STREAM)

Similar a UDPServer, asociamos el número de puerto del servidor, serverPort, con este socket:

serverSocket.bind((’’, serverPort))

Pero con TCP, serverSocket será nuestro socket de bienvenida. Después de establecer esta puerta de bienvenida, esperará y escuchará a que algún cliente toque a la puerta:

serverSocket.listen(1)

Esta línea hace que el servidor escuche las solicitudes de conexión TCP del cliente. El parámetro especifica el número máximo de conexiones en cola (al menos 1).

connectionSocket, addr = serverSocket.accept()

Cuando un cliente llama a esta puerta, el programa invoca el método accept() para serverSocket, que crea un nuevo socket en el servidor, llamado connectionSocket, dedicado a este cliente en particular. El cliente y el servidor luego completan el protocolo de enlace, creando una conexión TCP entre el cliente clientSocket y el connectionSocket del servidor. Con la conexión TCP establecida, el el cliente y el servidor ahora pueden enviarse bytes entre sí a través de la conexión. Con TCP, todos los bytes enviados desde no solo se garantizan que un lado llegue al otro lado, sino que también se garantiza que llegue en orden.

connectionSocket.close()

En este programa, luego de enviar la sentencia modificada al cliente, cerramos el socket de conexión. Pero dado que serverSocket permanece abierto, otro cliente ahora puede llamar a la puerta y enviarle un mensaje al servidor frase a modificar.

Esto completa nuestra discusión sobre la programación de sockets en TCP. Se le anima a ejecutar los dos programas en dos hosts separados, y también modificarlos para lograr objetivos ligeramente diferentes. Debería compare el par de programas UDP con el par de programas TCP y vea en qué se diferencian. también deberías hacer muchas de las asignaciones de programación de sockets descritas al final de los capítulos 2, 4 y 9. Finalmente, Espero que algún día, después de dominar estos y otros programas de socket más avanzados, escriba su propio popular aplicación de red, vuélvete muy rico y famoso, ¡y recuerda a los autores de este libro de texto!