Python Network Client Server

Contents

Time required: 60 minutes

- Comment each line of code as shown in the tutorials and other code examples.
- Follow all directions carefully and accurately.
- Think of the directions as minimum requirements.

Socket Background

Sockets have a long history. Their use originated with ARPANET in 1971 and later became an API in the Berkeley Software Distribution (BSD) operating system released in 1983 called Berkeley sockets.

When the Internet took off in the 1990s with the World Wide Web, so did network programming. Web servers and browsers weren't the only applications taking advantage of newly connected networks and using sockets. Client-server applications of all types and sizes came into widespread use.

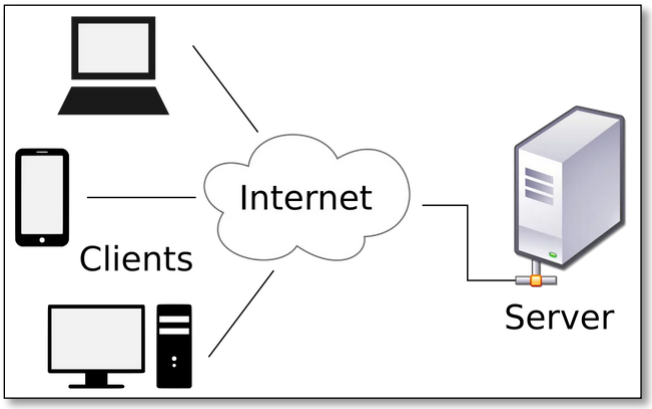
Today, although the underlying protocols used by the socket API have evolved over the years, and new ones have developed, the low-level API has remained the same.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that you'll be creating in this tutorial. More specifically, you'll focus on the socket API for Internet sockets, sometimes called Berkeley or BSD sockets.

What is a Socket?

A socket, in networking terminologies, serves as an intermediate connecting the application layer to the transport layer in the TCP/IP protocol suite. These network sockets are present on the client-side and the server-side.

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while another socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.



Client Server Model

Network Programming with Python

Python provides two levels of access to network services. You can access the socket capabilities in the operating system to create TCP (connection oriented) and UDP (connectionless) sockets. Python also has libraries for specific network protocols such as FTP, HTTP, and many others.

TCP Sockets

Sockets provide the endpoints of a bidirectional communications channel. Sockets can transfer data between two end points using a network.

You're going to create a socket object using **socket.socket()**, specifying the socket type as **socket.SOCK_STREAM**. When you do that, the default protocol that's used is the Transmission Control Protocol (TCP). This is a good default and probably what you want.

```
client_socket = socket.socket(AF_INET, SOCK_STREAM)
server_socket = socket.socket(AF_INET, SOCK_STREAM)
```

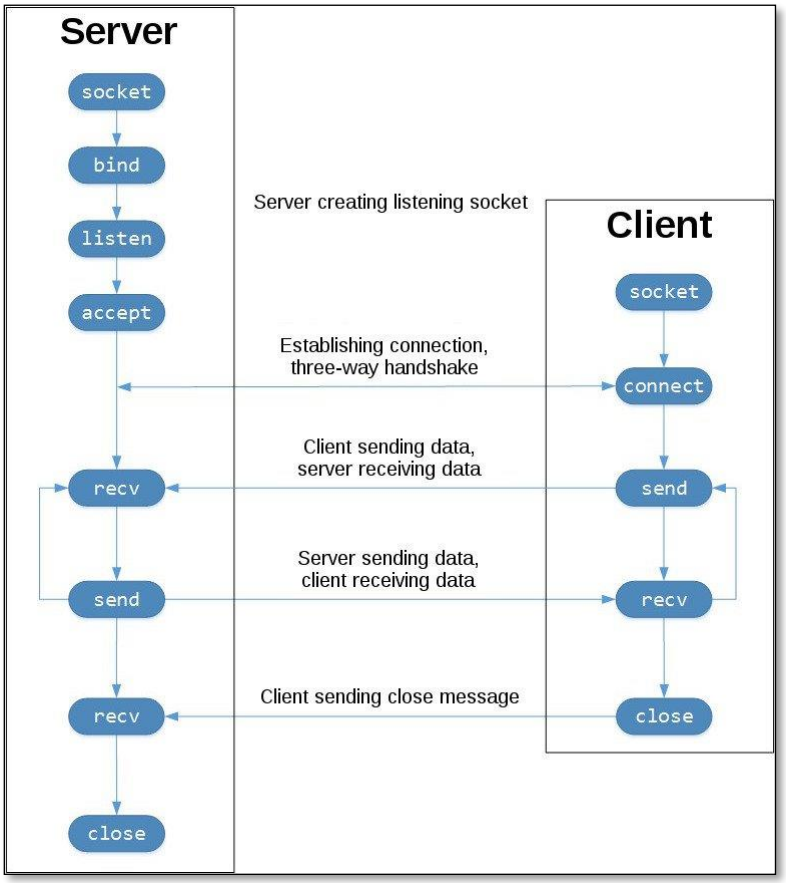
Why should you use TCP? The Transmission Control Protocol (TCP):

- **Is reliable:** Packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** Data is read by your application in the order it was written by the sender.

In contrast, **User Datagram Protocol (UDP)** sockets created with **socket.SOCK_DGRAM** aren't reliable, and data read by the receiver can be out-of-order from the sender's writes.

Why is this important? Networks are a best-effort delivery system. There's no guarantee that your data will reach its destination or that you'll receive what's been sent to you.

Network devices, such as routers and switches, have finite bandwidth available and come with their own inherent system limitations. They have CPUs, memory, buses, and interface packet buffers, just like your clients and servers. TCP relieves you from having to worry about packet loss, out-of-order data arrival, and other pitfalls that invariably happen when you're communicating across a network.



The left-hand column represents the server. On the right-hand side is the client.

Starting in the top left-hand column, note the API calls that the server makes to set up a "listening" socket:

- socket() (Create socket object)
- .bind() (Bind a hostname and port to a socket)
- .listen() (Setup and start a TCP listener on a socket)
- .accept() (Passively accept a TCP client connection. .accept() waits until a connection arrives.)

A listening socket does just what its name suggests. It listens for connections from clients. When a client connects, the server calls **.accept()** to accept, or complete, the connection.

The client calls **.connect()** to establish a connection to the server and initiate the three-way handshake. The handshake step is important because it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. It may be that only one host, client, or server can reach the other.

In the middle is the round-trip section, where data is exchanged between the client and server using calls to **.send()** and **.recv()**.

At the bottom, the client and server close their respective sockets.

Create Network Server

NOTE: This tutorial will work in **Windows** or **Linux**.

We are going to write a small client server program in Python. This program will demonstrate the use of sockets, IP address, and ports.

1. Create a Python program named: **network_server.py**

The first step is to import the **socket** library. This Python library contains the necessary functions to implement sockets.

2. **Create the Socket**

```python
"""
    Name: network_server.py
    Author:
    Created:
    Purpose: A simple network server
"""
# Python built in socket library
import socket

# The server will accept a connection on any interface
SERVER_IP = ""

# Port to listen on (non-privileged ports are > 1023)
# Listening server port
PORT = 8081

def main():
    # Create a socket object
    server_socket = socket.socket(
        socket.AF_INET,          # TCP/IP v4 address
        socket.SOCK_STREAM       # Create TCP transport layer socket
    )
```

SERVER_IP = "" allows the server to accept a connection on any interface.

The **PORT** is assigned as 8081. This port is chosen because this is a default-free port on most machines. You can also run it on any port such as '1234'.

The **socket()** function is a constructor of the socket library to create a socket. This will create a TCP socket with an IPV4 address.

3. **Bind the Host and Port**

```python
    # Bind socket to a tuple (IP Address, Port Address)
    server_socket.bind((SERVER_IP, PORT))
```

We will bind the port and host together using the bind function which is invoked on the socket object **server_socket**

4. **Listen for Connections**

```
27      # Start server socket listener, waiting for a connection
28      server_socket.listen()
29      print(f"Listening for incoming connections on port {PORT} . . .")
```

The **listen()** function which takes in one argument **number_of_connections.** This parameter can be any whole number such as 1,2,3. If it is blank, the **listen()** function automatically provides a reasonable number of available connections.

5. **Accept Incoming Connections**

```
31      while True:
32          # Accept a connection
33          # Return socket object and IP address
34          connection, client_address = server_socket.accept()
35          # Print returned socket information for debugging
36          print(connection)
37          print(client_address)
38          print(f"Connection from: {client_address}")
39
40          # Send message to client, encode string to bytestream
41          # for transmission over the network
42          connection.send("You are connected".encode("utf-8"))
43
44          # Receive data into a 1024 byte buffer
45          data = connection.recv(1024)
46          print(f"The client has connected")
```

The first variable **connection** is the connection to the socket. The variable **address** is assigned to the IP address of the client. The connection receives a maximum of 1024 bytes of binary data into the buffer.

6. **Store Incoming Connection Data**

```
48          if data:
49              # If the server recieves data
50              # Convert bytestream to text/string
51              message = data.decode("utf-8")
52              print(message)
53
54              # Close the connection
55              connection.close()
56              print("Client disconnected")
57              print(f"Listening for incoming connections on port {PORT} . . .")
58
59  main()
```

The data decoded from binary to a utf-8 string from the incoming connection is stored in the **data** variable. The **data** can be a maximum of 1024 bytes. It is decoded on the server and prints a message that it has been connected.

The connection is closed. The server listens for another incoming connection.

Create Network Client

Let's create network client to connect and communicate with the network server.

Create a Python file named **network_client.py**

```
1   """
2       Name: network_client.py
3       Author:
4       Created:
5       Purpose: Simple network client
6   """
7   # Python built in socket library
8   import socket
9
10  # Use 127.0.0.1 if you are running the server program
11  # on the same computer you are running the client program
12  # Change the IP address if the server program
13  # is on another computer
14  SERVER_IP = "127.0.0.1"
15
16  # Specify the destination port
17  PORT = 8081
```

We import the same socket library we used on the server-side program

SERVER_IP is the IP address of the server we want to connect to. If you are running both programs on the same computer, the server address would be 127.0.0.1

If you have the server software running on another computer, put in the IP address of the other computer.

This port must match with the port mentioned on the server-side code.

```
20  def main():
21      # Create TCP IPV4 socket
22      client_socket = socket.socket(
23          socket.AF_INET,         # TCP/IP v4 address
24          socket.SOCK_STREAM      # Create TCP socket
25      )
```

The client creates a socket exactly like the server did. There must be a socket on each side to make a connection.

```
27      message = f"Hello Server, I have connected on port {PORT}"
28
29      # Convert Unicode utf-8 string to byte data type
30      data = message.encode("utf-8")
```

Any data to be sent over the network must be converted/encoded into byte data. utf-8 (Unicode) is standard text encoding in Python and most operating systems and programming languages.

```
32      # Connect to the server on the specified IP address and port
33      client_socket.connect((SERVER_IP, PORT))
34
35      # Send the bytestream
36      client_socket.send(data)
37      # Print message string to confirm transmission
38      print(message)
```

The client makes a connection with the server. The message string is printed out to let the client know the connection was made and the data was sent.

```
41      # Pause to allow commmunication to finish
42      time.sleep(1)
43      # Close the connection
44      client_socket.close()
45      print("Message sent. Socket closed.")
46
47
48  main()
```

After a successful transmission of data, the connection is closed.

Example run:

1. Run the server program first.
2. Run the client program next.

Server example run:

```
Listening for incoming connections on port 8080 . . .
Connection from: ('127.0.0.1', 19115)
The client has connected
Client connected to the server on port 8080
Client disconnected
Listening for incoming connections on port 8080 . . .
```

Client example run:

```
Client connected to the server on port 8080
Message sent. Socket closed.
```

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.