6 Python Sockets: A Simple Client-Server Example

Jacques Mock Schindler

03.09.2025

In this section, we explore how to create a simple client-server application using Python sockets. This example will demonstrate the basic concepts of socket programming, including how to establish a connection and send and receive data.

> **Network Socket**
>
> A network socket is an endpoint within a host used to send and receive data over a network.

Communication Structure

The example will consist of a server that listens for incoming connections and a client that connects to the server. We will model the communication as a simple chat service.

Server Code

You don't need a virtual environment. The example uses only the Python standard library.
Below is the working code example for the socket server:

```
# socket_server.py

import socket

def server_program():
    # define host name and port
    host = 'localhost'    # for communication on the local
 ↪  machine
    port = 5000           # use a port number above 1024

    # create socket
    server_socket = socket.socket()
```

```
    # bind the socket to the host and port
    server_socket.bind((host, port))

    # set the server to listen for connections
    server_socket.listen(2) # start listening
                            # with a backlog of 2
                            # (max queued connections)

    # accept new connection from client
    conn, address = server_socket.accept()
    print("Connection from: " + str(address))

    while True:
        # receive up to 1024 bytes per call
        data = conn.recv(1024).decode()
        if not data:
            # if no data is received, break
            break
        print("Received from connected client: " + str(data))
        # prompt the user to enter a message
        data = input(' -> ')
        # send data to the client as bytes
        conn.send(data.encode())

    # close the connection
    conn.close()

if __name__ == '__main__':
    server_program()
```

To run the script directly, open a terminal in the directory where the script is located and run the following command:

```
python socket_server.py
```

What happens when you do so is explained in the following sections.

The code consists of two main parts: the `server_program` function and the `if __name__ == '__main__'` block. The `server_program` function contains the main logic for the server, while the `if` block is used to execute the server code when the script is run directly.

When you run the script, Python first imports the `socket` module.

```
import socket
```

Next, the `server_program` function is called. Inside this function, the first thing that happens is the definition of the host address and the port number. `localhost` is a special address that refers to the local machine. `localhost` is equivalent to the IPv4 address `127.0.0.1`. This means that the server will only accept connections from clients running on the same machine. This is a simulation of a network on the local machine.

```
def server_program():
    # define host name and port
    host = 'localhost'    # for communication on the local
 ↪  machine
    port = 5000           # use a port number above 1024
```

Below, the `server_socket` is created using the `socket.socket()` function, which creates a new socket object.

```
server_socket = socket.socket()
```

This socket will be used to listen for incoming connections from clients. To make the connection work, the `server_socket` must be bound to the host and port using the `bind()`.

```
server_socket.bind((host, port))
```

Next, the server is set to listen for incoming connections with the `listen()` method.

```
server_socket.listen(2) # start listening
                        # with a backlog of 2
                        # (max queued connections)
```

The server sets a backlog of 2 pending connections. The code, however, accepts exactly one client connection (accept() is called once). To handle multiple clients, call accept() in a loop and use threads or asyncio. This limit is set to prevent the server from being overwhelmed by too many connections at once. The `accept()` method is used to accept a new connection from a client. This method returns a tuple containing two items: conn, a new socket object for client communication, and address, the client's address. The next chunk shows, how a message is printed to show the address of the connected client.

```
conn, address = server_socket.accept()
print("Connection from: " + str(address))
```

The while loop lets the server wait for data from the client. The `recv()` method is used to receive data from the client. Each `recv()` call reads up to 1024 bytes; additional data remains in the buffer and can be read by subsequent calls. If no data is received, the server breaks out of the loop and closes the connection.

```
while True:
        # receive up to 1024 bytes per call
        data = conn.recv(1024).decode()
        if not data:
            # if no data is received, break
            break
```

If the server receives data, it prints the data to the console and then prompts the user to enter a response. The response is sent back to the client using the `send()` method.

```
print("Received from connected client: " + str(data))
        # prompt the user to enter a message
        data = input(' -> ')
        # send data to the client as bytes
        conn.send(data.encode())
```

Client Code

Below is the working code of the client example:

```
# socket_client.py

import socket

def client_program():
    # define host name and port (of the server to connect to)
    host = 'localhost'  # as both pieces of code are
                        # running on the same machine
    port = 5000         # socket server port number

    # create socket
    client_socket = socket.socket()

    # connect to the server
    client_socket.connect((host, port))

    # prompt the user to enter a message
    message = input(" -> ")  # take input

```

```
    # message loop
    while message.lower().strip() != 'bye':
        # send message to the server as bytes
        client_socket.send(message.encode())

        # receive response from the server
        data = client_socket.recv(1024).decode()

        print('Received from server: ' + data)

        # prompt the user to enter a new message
        message = input(" -> ")  # take new input

    # close the connection
    client_socket.close()

if __name__ == '__main__':
    client_program()
```

To run this script, follow the same steps as for the server script. Open a terminal in the directory where the script is located and run the following command:

```
python socket_client.py
```

Start the server before the client. If the server is not running, client_socket.connect(…) will raise a ConnectionRefusedError.

When you run the script, Python first imports the `socket` module.

Next, the `client_program` function is called. Inside this function, the first thing that happens is the definition of the host address and the port number. Here, the host address and the port number have to be the ones defined in the server script.

After defining the host and port, the client creates a socket object using the `socket.socket()` function. This socket will be used to connect to the server. The `connect()` method is called on the socket object to establish a connection to the server.

Once the connection is established, the client enters a loop where it can send messages to the server and receive responses. The client prompts the user to enter a message, which is then sent to the server using the `send()` method. The client also waits for a response from the server using the `recv()` method.

If the user enters `bye`, the client will exit the loop and close the connection to the server.

Conclusion

The presented code examples illustrate how Python's built-in socket module (standard library) enables straightforward implementation of networked applications without requiring external dependencies. The tutorial's focus on localhost communication provides a safe, controlled environment for experimentation and learning, while the bidirectional message exchange demonstrates real-world communication patterns found in production systems.

Key achievements of this implementation include:

- Successful establishment of TCP socket connections between separate processes
- Implementation of message-based communication with proper encoding/decoding
- Demonstration of connection lifecycle management from establishment to termination
- Creation of a simple command-line interface for both server and client applications

The knowledge gained from this tutorial serves as a solid foundation for developing more sophisticated networked applications. Future enhancements could include implementing multithreading for concurrent client handling, adding error handling and reconnection logic, or expanding the communication protocol to support structured data formats.