Python Socket Communication

Source: Medium - Python Pandemonium

Author: The Fellow

Publication: Python Pandemonium

Original Date: February 2, 2019

Saved: February 2, 2026

What is a Socket?

A socket is one endpoint of a two-way communication link between two programs running on a node in a computer network. One socket (the server) listens on a particular port on an IP address, while another socket (the client) connects to the listening server to achieve communication.

Primarily, the way sockets send data is controlled by two properties:

- **Address Family** - Determines the network layer protocol used
- **Socket Type** - Determines the transport layer protocol used

Socket Types

1. `SOCK_DGRAM` - Message-oriented datagram transport (UDP)
    - Unreliable delivery of individual messages
    - Used when order of messages is not important
    - Common for multicasting data to multiple clients
2. `SOCK_STREAM` - Stream-oriented transport (TCP)
    - Reliable and ordered delivery of bytes
    - Includes error handling and control
    - Useful for large data transfers

Address Families

Address Family	Description	Usage
`AF_INET`	IPv4 network addressing	Most common, standard internet networking
`AF_INET6`	IPv6 network addressing	Next generation internet protocol
`AF_UNIX`	Unix Domain Sockets	Inter-process communication on POSIX systems

Python Socket Module Overview

To use sockets in Python, import the socket module:

```
import socket
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Network Related Functions

- `gethostname()` - Get official name of current host
- `gethostbyname()` - Convert server name to IP address
- `gethostbyname_ex()` - Get detailed naming information
- `gethostbyaddr()` - Perform reverse DNS lookup
- `getfqdn()` - Convert partial domain to fully qualified domain name

TCP/IP Client-Server Communication

Server Methods

- `socket()` - Creates a new socket
- `bind()` - Binds socket to a particular address (host:port)
- `listen()` - Starts accepting connections
- `accept()` - Accepts incoming connections, returns (conn, address)
- `close()` - Marks socket as closed

Client Methods

- `socket()` - Creates a new socket
- `connect()` - Connects to server address
- `sendall(data)` - Sends all data to server
- `recv(bufsize)` - Receives data in chunks
- `close()` - Closes the connection

Echo Server Example

The server continuously accepts connections and echoes back received data:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('localhost', 10000))
sock.listen(1)
while True:
    connection, client_address = sock.accept()
    try:
        while True:
            data = connection.recv(16)
            if data:
                connection.sendall(data)
    finally:
        connection.close()
```

Echo Client Example

The client connects, sends a message, and receives the echo:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 10000))
try:
```

```
    message = b'This is our message'
    sock.sendall(message)
    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
finally:
    sock.close()
```

UDP Client-Server Communication

Key Differences from TCP

- No connection establishment (connectionless)
- Message-oriented rather than stream-oriented
- No ordered delivery guarantee
- Faster but less reliable

UDP Methods

- `recvfrom(bufsize)` - Receives data, returns (bytes, address)
- `sendto(bytes, address)` - Sends data to specific address

UDP Server Example

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('localhost', 10000))
while True:
    data, address = sock.recvfrom(4096)
    if data:
        sock.sendto(data, address)
```

Multicast Communication

Multicast enables efficient one-to-many communication. Messages are sent to multicast groups (addresses 224.0.0.0 to 239.255.255.255).

Key Concepts

- **TTL (Time To Live)** - Controls broadcast range (default: 1)
- **Multicast Group** - Special IP addresses for group communication
- **IP_MULTICAST_TTL** - Socket option to set TTL
- **IP_ADD_MEMBERSHIP** - Socket option to join a group

Sending Binary Data

Use the `struct` module to pack and unpack binary data:

```
import struct

# Packing (send)
format_string = 'I 2s f'  # unsigned int, 2-char string, float
values = (1, b'ab', 2.7)
```

```
packed = struct.pack(format_string, *values)

# Unpacking (receive)
unpacked = struct.unpack(format_string, packed)
# Result: (1, b'ab', 2.700000047683716)
```

> *Note: Floating point numbers may lose precision when packed and unpacked. Both sender and receiver must use the same format string.*

Blocking and Timeouts

Blocking Mode (Default)

By default, sockets operate in blocking mode. Sending or receiving data pauses execution until the socket is ready.

Non-Blocking Mode

```
sock.setblocking(0)  # Disable blocking
```

Problem: Raises `socket.error` if not ready

Timeout (Recommended)

```
sock.settimeout(0.2)  # 200ms timeout
```

Provides a compromise between blocking and non-blocking modes.

Unix Domain Sockets

Similar to TCP/IP sockets but use filesystem paths instead of host:port combinations.

Key Differences

- Address is a filesystem path: `'./socket_file'`
- Socket file persists after closure
- Must be removed before server restarts
- Standard file permissions control access

```
import os
try:
    os.unlink('./socket_file')
except:
    pass

sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
sock.bind('./socket_file')
```

Troubleshooting

Using ping

Check host connectivity using ICMP echo requests:

```
$ ping -c 4 127.0.0.1
```

Using netstat

View socket information and connection states:

```
$ netstat -an
```

Look for:

- Protocol (tcp/udp)
- State (LISTEN, ESTABLISHED, etc.)
- Local and remote addresses
- Port numbers

Summary

Protocol	Connection	Reliability	Use Case
TCP	Connected	Reliable, ordered	File transfer, HTTP, Telnet
UDP	Connectionless	Unreliable, unordered	DNS, Video streaming, Games
Multicast	Group	One-to-many broadcast	Live events, Notifications
UDS	Local	IPC only	Inter-process communication

Key Takeaways

- TCP provides reliable, ordered delivery
- UDP is faster but delivery not guaranteed
- Multicast enables efficient broadcasting
- Proper error handling prevents deadlocks
- Timeouts are preferred over non-blocking sockets
- Use netstat and ping for troubleshooting
