# Application Layer: Socket Programming with UDP and TCP

Study-Ready Notes

Compiled by Andrew Photinakis

October 17, 2025

## Contents

**1 Socket Programming Fundamentals** **2**
- 1.1 Socket Definition and Purpose . . . 2
- 1.2 Two Socket Types . . . 2
- 1.3 Application Example . . . 2

**2 Socket Programming with UDP** **2**
- 2.1 UDP Characteristics . . . 2
- 2.2 Client/Server Socket Interaction: UDP . . . 3
- 2.3 Example Application: UDP Client . . . 3
- 2.4 Example Application: UDP Server . . . 3

**3 Socket Programming with TCP** **3**
- 3.1 TCP Characteristics . . . 3
- 3.2 Client/Server Socket Interaction: TCP . . . 4
- 3.3 Example Application: TCP Client . . . 4
- 3.4 Example Application: TCP Server . . . 4

**4 Chapter 2: Summary** **4**
- 4.1 Key Concepts Covered . . . 4
- 4.2 Protocol Design Principles . . . 5

# 1 Socket Programming Fundamentals

## 1.1 Socket Definition and Purpose

- **Socket**: A door between application process and end-to-end transport protocol
- Acts as an interface between application layer and transport layer
- Application developer controls the application process
- Operating system controls the socket and underlying transport protocols

## 1.2 Two Socket Types

- **UDP (User Datagram Protocol)**: Unreliable datagram service
- **TCP (Transmission Control Protocol)**: Reliable, byte stream-oriented service

## 1.3 Application Example

1. Client reads a line of characters (data) from its keyboard and sends data to server
2. Server receives the data and converts characters to uppercase
3. Server sends modified data to client
4. Client receives modified data and displays line on its screen

[Summary: Socket programming enables communication between client and server applications. UDP provides connectionless, unreliable service while TCP provides connection-oriented, reliable service.]

[Mnemonic: UDP = Unreliable Datagram Protocol; TCP = Trustworthy Connection Protocol]

# 2 Socket Programming with UDP

## 2.1 UDP Characteristics

- No "connection" between client and server
- No handshaking before sending data
- Sender explicitly attaches IP destination address and port number to each packet
- Receiver extracts sender IP address and port number from received packet
- Transmitted data may be lost or received out-of-order
- Provides *unreliable* transfer of groups of bytes ("datagrams")

## 2.2 Client/Server Socket Interaction: UDP

| Server | Client |
|---|---|
| create socket, port=x:<br>serverSocket = socket(AF_INET,SOCK_DGRAM) | create socket:<br>clientSocket = socket(AF_INET,SOCK_DGRAM) |
| read datagram from serverSocket | Create datagram with serverIP address<br>and port=x; send datagram via clientSocket |
| write reply to serverSocket<br>specifying client address, port number | read datagram from clientSocket |
| | close clientSocket |

Figure 1: UDP Client/Server Interaction Sequence

## 2.3 Example Application: UDP Client

```
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()
```

## 2.4 Example Application: UDP Server

```
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)
```

[Summary: UDP socket programming involves connectionless communication where each datagram is sent independently. The server binds to a port and waits for incoming datagrams, while the client sends datagrams to the server's address and port.]

[Concept Map: UDP → Connectionless → No handshaking → Explicit addressing → Unreliable → May lose/reorder data → Suitable for real-time applications]

# 3 Socket Programming with TCP

## 3.1 TCP Characteristics

- Client must contact server first
- Server process must be running first
- Server must have created socket (welcoming socket) for client contact
- When client creates socket: client TCP establishes connection to server TCP
- When contacted by client, server TCP creates new socket for that particular client
- Allows server to communicate with multiple clients simultaneously
- Source port numbers used to distinguish clients
- Provides reliable, in-order byte-stream transfer ("pipe")

## 3.2 Client/Server Socket Interaction: TCP

| Server | Client |
|---|---|
| create socket, port=x, for incoming request: serverSocket = socket() | |
| wait for incoming connection request connectionSocket = serverSocket.accept() | create socket, connect to hostid, port=x clientSocket = socket() |
| read request from connectionSocket | send request using clientSocket |
| write reply to connectionSocket | read reply from clientSocket |
| close connectionSocket | close clientSocket |

Figure 2: TCP Client/Server Interaction Sequence

## 3.3 Example Application: TCP Client

```
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()
```

### 3.4 Example Application: TCP Server

```
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()
```

[Summary: TCP socket programming involves connection-oriented communication with a three-way handshake. The server creates a welcoming socket, accepts client connections, and creates dedicated sockets for each client, providing reliable, in-order data transfer.]

## 4 Chapter 2: Summary

### 4.1 Key Concepts Covered

- **Application Architectures**:
  - Client-server architecture
  - P2P (Peer-to-Peer) architecture
- **Service Requirements Specification**:
  - Reliability requirements
  - Bandwidth requirements
  - Delay requirements
- **Internet Transport Service Model**:
  - Connection-oriented, reliable: TCP
  - Unreliable, datagrams: UDP
- **Specific Protocols Studied**:
  - HTTP (HyperText Transfer Protocol)
  - SMTP (Simple Mail Transfer Protocol), IMAP (Internet Message Access Protocol)
  - DNS (Domain Name System)

  - P2P: BitTorrent

- **Advanced Topics**:
  - Video streaming techniques
  - Content Delivery Networks (CDNs)
  - Socket programming with TCP and UDP

## 4.2 Protocol Design Principles

- **Typical request/reply message exchange**:
  - Client requests information or service
  - Server responds with data, status code
- **Message formats**:
  - *Headers*: Fields giving information about data
  - *Data*: Information (payload) being communicated
- **Important design themes**:
  - Centralized vs. decentralized architectures
  - Stateless vs. stateful protocols
  - Scalability considerations
  - Reliable vs. unreliable message transfer
  - "Complexity at network edge" principle

[Summary: The application layer encompasses diverse protocols and architectures for network applications. Key considerations include reliability, performance requirements, and the choice between connection-oriented (TCP) and connectionless (UDP) transport services.]

[Concept Map: Application Layer → Architectures (Client-Server, P2P) → Transport Services (TCP, UDP) → Protocols (HTTP, SMTP, DNS) → Advanced Topics (Streaming, CDNs) → Socket Programming]

## Key Differences: TCP vs UDP

## Study Tips

- Practice writing both UDP and TCP client-server code
- Understand the socket API method sequence for both protocols
- Memorize the key differences between connection-oriented and connectionless protocols
- Be able to explain when to use TCP vs UDP for different application requirements
- Review the protocol headers and message formats for HTTP, SMTP, and DNS

| **Feature** | **TCP** | **UDP** |
|---|---|---|
| Connection | Connection-oriented | Connectionless |
| Reliability | Reliable | Unreliable |
| Ordering | In-order delivery | No ordering guarantees |
| Handshaking | 3-way handshake required | No handshaking |
| Overhead | Higher | Lower |
| Use Cases | Web, email, file transfer | DNS, streaming, VoIP |

Table 1: Comparison of TCP and UDP Protocols