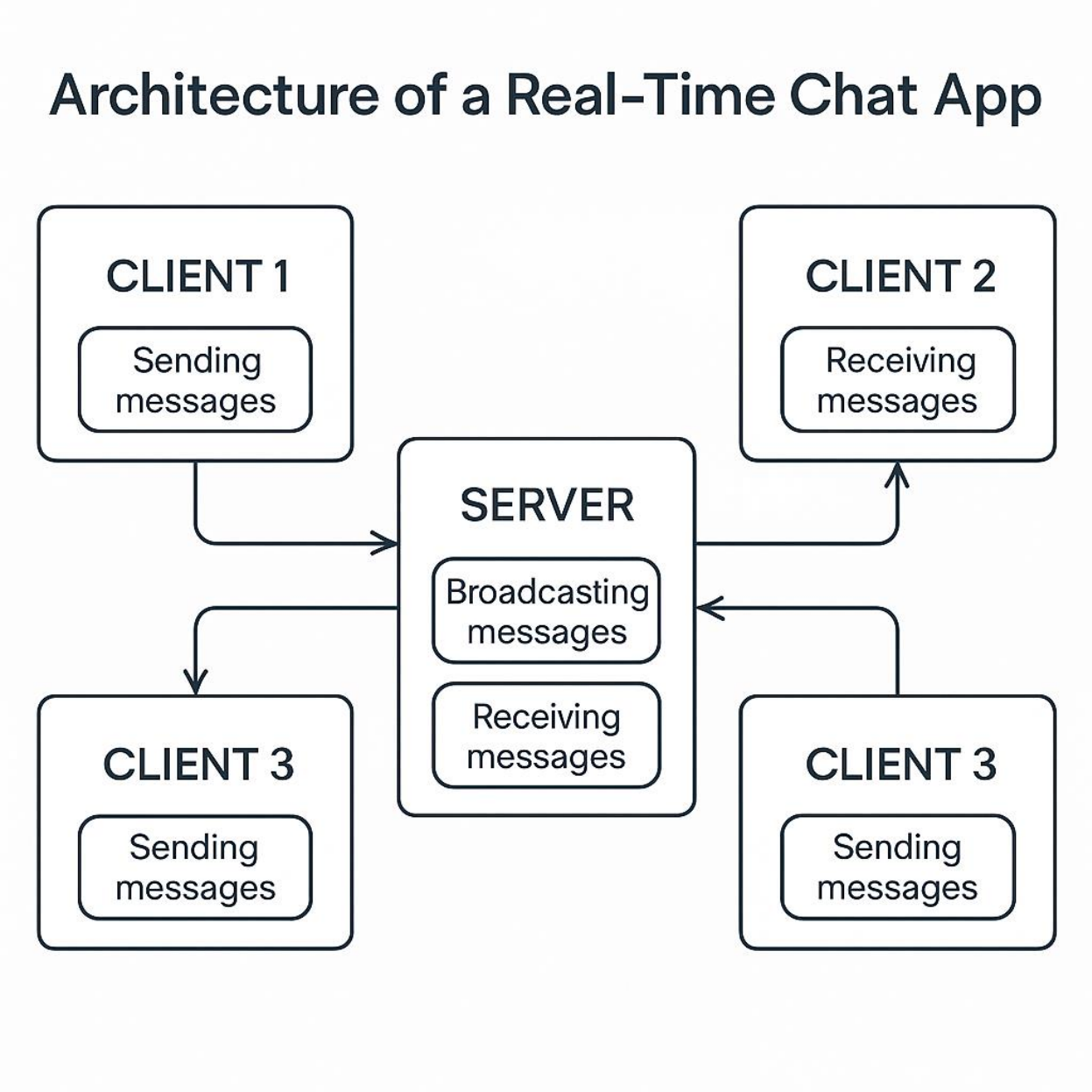
BUILDING A REAL-TIME CHAT APP IN PYTHON USING SOCKET PROGRAMMING



Executive Summary

This case study explains the development of a real-time chat application using Python's socket and threading libraries. The project introduces students to client-server architecture, concurrent message handling, and error management in network-based applications. The system supports multiple clients connecting to a single server, real-time broadcasting of messages, and clean

disconnection handling. All code components are annotated and explained to help students understand the logic and reasoning behind implementation choices.

---

1. Introduction

Real-time communication systems like WhatsApp, Slack, and Teams are underpinned by core principles of socket programming. Understanding how messages are sent, received, and broadcast across clients using network sockets is critical for students learning computer networks and backend development. This case study walks through the logic of developing a simple chat application in Python that supports multiple users.

---

2. Problem Definition

Students often struggle with:

- Understanding how servers listen for and handle multiple client connections
- Managing synchronous vs asynchronous data transmission
- Keeping the chat logic separate for sending and receiving messages
- Avoiding crashes when clients disconnect unexpectedly

---

3. System Requirements

- **Language**: Python 3.x
- **Libraries Used**: socket, threading
- **Type**: Command-line interface (CLI)
- **Architecture**: Client-server, multiple clients, one central server

---

4. Design Objectives

- Create a server that listens on a port and accepts multiple clients
- Use threading to manage each client's incoming messages

- Broadcast messages to all clients except the sender
- Handle disconnections and exceptions gracefully

---

5. Server Code with Logic Explanation

python

CopyEdit

```
import socket
import threading

# Step 1: Set up server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', 5555))
server.listen()

clients = []
aliases = []

def broadcast(message, sender):
    for client in clients:
        if client != sender:
            client.send(message)

def handle_client(client):
    while True:
        try:
            message = client.recv(1024)
```

```
            broadcast(message, client)
        except:
            index = clients.index(client)
            clients.remove(client)
            client.close()
            alias = aliases[index]
            broadcast(f"{alias} left the chat.".encode('utf-8'), client)
            aliases.remove(alias)
            break

def receive_connections():
    while True:
        client, address = server.accept()
        client.send("ALIAS?".encode('utf-8'))
        alias = client.recv(1024).decode('utf-8')
        aliases.append(alias)
        clients.append(client)
        print(f"Connected with {alias}")
        broadcast(f"{alias} joined the chat!".encode('utf-8'), client)
        client.send("You are now connected!".encode('utf-8'))

        thread = threading.Thread(target=handle_client, args=(client,))
        thread.start()

print("Server is running...")
receive_connections()
```

Key Concepts Explained:

- **socket.AF_INET, socket.SOCK_STREAM**: IPv4 and TCP
- **server.listen()**: Prepares the server to accept connections
- **broadcast()**: Sends a message to all clients except the sender
- **threading.Thread**: Each client has its own thread for simultaneous interaction

---

6. Client Code with Logic Explanation

python

CopyEdit

```
import socket
import threading

alias = input("Enter your name: ")
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost', 5555))

def receive():
    while True:
        try:
            message = client.recv(1024).decode('utf-8')
            if message == "ALIAS?":
                client.send(alias.encode('utf-8'))
            else:
                print(message)
        except:
            print("An error occurred.")
            client.close()
```

```
            break


def write():
    while True:
        message = f'{alias}: {input("")}'
        client.send(message.encode('utf-8'))


receive_thread = threading.Thread(target=receive)
receive_thread.start()


write_thread = threading.Thread(target=write)
write_thread.start()
```

Key Concepts Explained:

- **Two Threads**:
    - One receives messages
    - One sends messages
- **Error Handling**: Client closes on failure to receive or send
- **Alias Handling**: Ensures usernames are attached to each message

---

7. Evaluation and Enhancements

Initial Testing:

- 3 clients connected successfully
- Messages broadcast in real-time
- Clients disconnect without crashing others

Proposed Enhancements:

- Add timestamps to messages

- Build a GUI using Tkinter
- Encrypt messages using ssl or cryptography module
- Allow private messaging with @username logic

---

8. Conclusion

This case study demonstrated how to design, code, and explain a real-time chat application using socket programming in Python. It offers students a practical understanding of networking, multithreading, and basic backend logic. The modular structure supports further extension into GUI-based apps, encrypted messaging, or deployment over public IPs.

---

9. References

- Python Docs: socket and threading Modules
- RFC 793 – Transmission Control Protocol (TCP)
- Real Python Tutorials: Socket Programming
- Network Programming Principles (Kurose & Ross)