CS 576
Computer Networks and Distributed Systems

Programming Assignment 1

TCP Socket Client-Server Application

Team Members:

Arellano, Jorge
Dennis, Benjamin Percy
Giangregorio, Ryan
Hafezi, Parsa
Hua, Alice Celina
Nguyen, Anh Huy
Nguyen, Thy
Rauniyar, Bishal
Smith, Jurne Jul
Thapa, Manish
Vargas, Jeremiah
Viraldo, Isabelle Jane

Contents

Executive Summary

This report documents the implementation of a TCP client-server application that encodes and decodes text messages using ASCII character shifting. The server accepts connections from clients, receives messages (up to 256 characters), and returns encoded or decoded versions by shifting each character in the ASCII sequence.

The implementation successfully demonstrates:

- TCP socket programming with reliable connection handling
- Client-server communication protocol
- ASCII character manipulation for encoding/decoding
- Comprehensive error handling and validation
- Both interactive and single-message operation modes

1 Project Overview

1.1 Objectives

The primary objectives of this programming assignment were to:

1. Implement a TCP server that:
   - Accepts client connections on a specified port
   - Receives text messages up to 256 characters
   - Encodes messages by shifting each character to the next ASCII character
   - Optionally decodes messages using a flag-based protocol
   - Handles errors gracefully and continues operation
2. Implement a TCP client that:
   - Connects to the server using TCP sockets
   - Sends messages for encoding or decoding
   - Receives and displays the processed message
   - Provides both interactive and single-message modes
3. Demonstrate proper socket programming practices including:
   - Error checking and exception handling
   - Protocol design and implementation
   - Resource management (socket cleanup)
   - User-friendly command-line interface

1.2 Encoding Algorithm

The encoding algorithm is straightforward:

- Each character is replaced with the next character in the ASCII sequence
- Implementation: `encoded_char = chr(ord(char) + 1)`
- Example: 'A' (ASCII 65) → 'B' (ASCII 66)

Example Transformation:

```
Original:  "Hello World"
Encoded:   "Ifmmp!Xpsme"
```

Character-by-character breakdown:

- H (72) → I (73)
- e (101) → f (102)
- l (108) → m (109)
- l (108) → m (109)
- o (111) → p (112)
- (space) (32) → ! (33)
- W (87) → X (88)

- o (111) → p (112)
- r (114) → s (115)
- l (108) → m (109)
- d (100) → e (101)

2 System Architecture

2.1 Overall Architecture

The system follows a classic client-server architecture using TCP/IP sockets. Figure 1 illustrates the high-level architecture of the system.

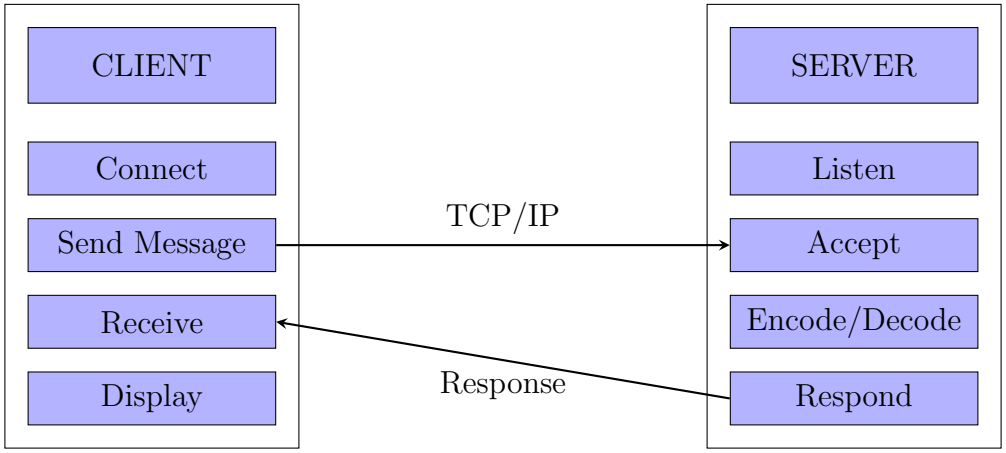


Figure 1: Client-Server Architecture

2.2 Protocol Design

Message Format:

- `ENCODE:<message>` - Request encoding (default behavior)
- `DECODE:<message>` - Request decoding
- Plain `<message>` - Also encodes (for simplicity)

Response Format:

- Success: Processed message string
- Error: `ERROR: <error description>`

2.3 Technology Stack

- **Language:** Python 3.6+
- **Core Library:** `socket` (built-in)
- **Additional Libraries:** `argparse` (command-line parsing), `sys` (system operations)
- **Platform:** Cross-platform (Windows, Linux, macOS)

3 Implementation Details

3.1 Server Implementation

3.1.1 Server Structure

The server (`tcp_server.py`) consists of the following key components:

Main Functions:

1. `encode_message(message)` - Shifts characters forward
2. `decode_message(message)` - Shifts characters backward
3. `process_client_message(message)` - Determines operation based on prefix
4. `handle_client(client_socket, address)` - Processes individual client
5. `start_server(host, port)` - Main server loop

3.1.2 Server Flow

Figure 2 shows the detailed flowchart for the server operation.

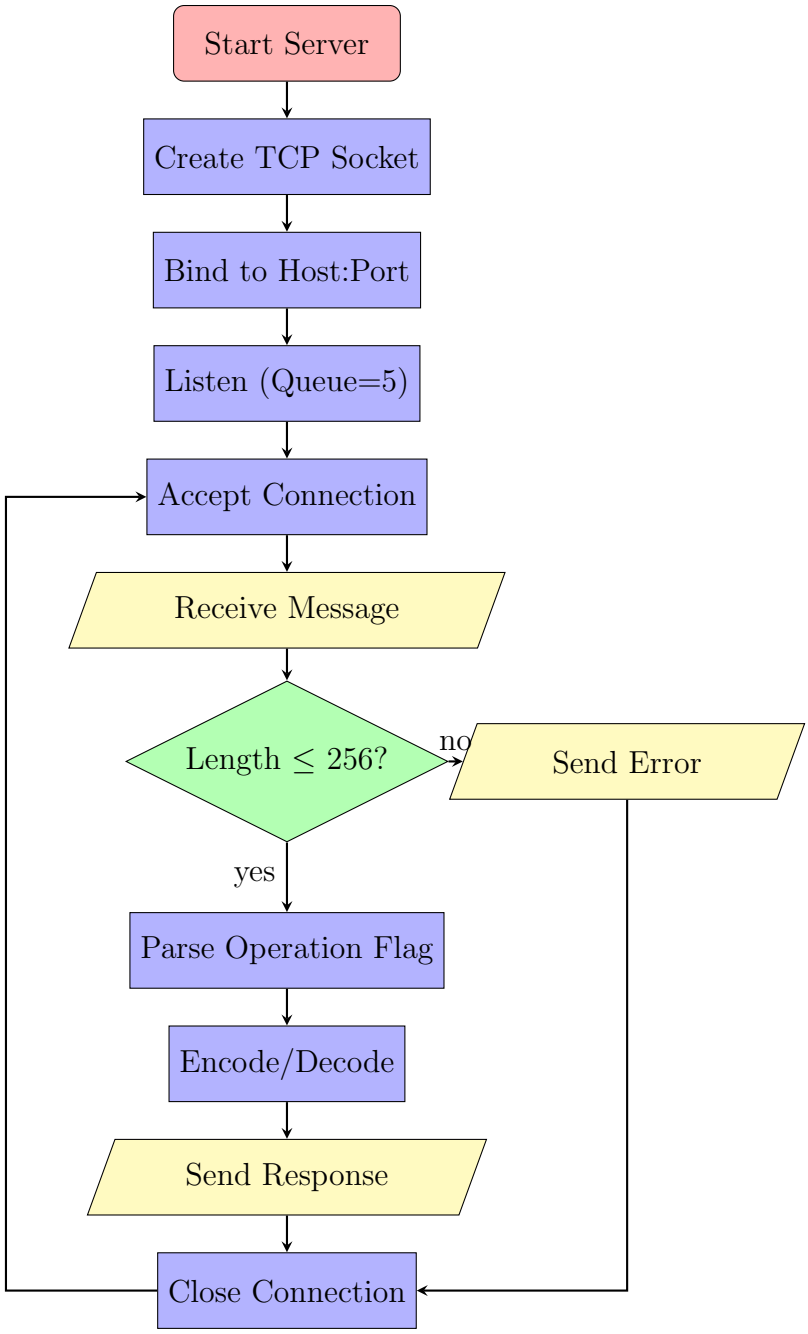


Figure 2: Server Operation Flowchart

3.1.3 Key Server Code Snippets

Socket Creation and Binding:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((host, port))
```

```
server_socket.listen(5)
```

Listing 1: Server Socket Setup

Encoding Function:

```
def encode_message(message):
    try:
        encoded = ''.join(chr(ord(char) + 1) for char in message)
        return encoded
    except Exception as e:
        raise ValueError(f"Error encoding message: {e}")
```

Listing 2: Message Encoding

Client Connection Handling:

```
client_socket, client_address = server_socket.accept()
data = client_socket.recv(BUFFER_SIZE).decode('utf-8')

if len(data) > MAX_MESSAGE_LENGTH:
    error_msg = f"ERROR: Message exceeds maximum length"
    client_socket.send(error_msg.encode('utf-8'))
    return

response = process_client_message(data)
client_socket.send(response.encode('utf-8'))
client_socket.close()
```

Listing 3: Client Handler

3.2 Client Implementation

3.2.1 Client Structure

The client (`tcp_client.py`) provides two operation modes:

Mode 1: Single Message Mode

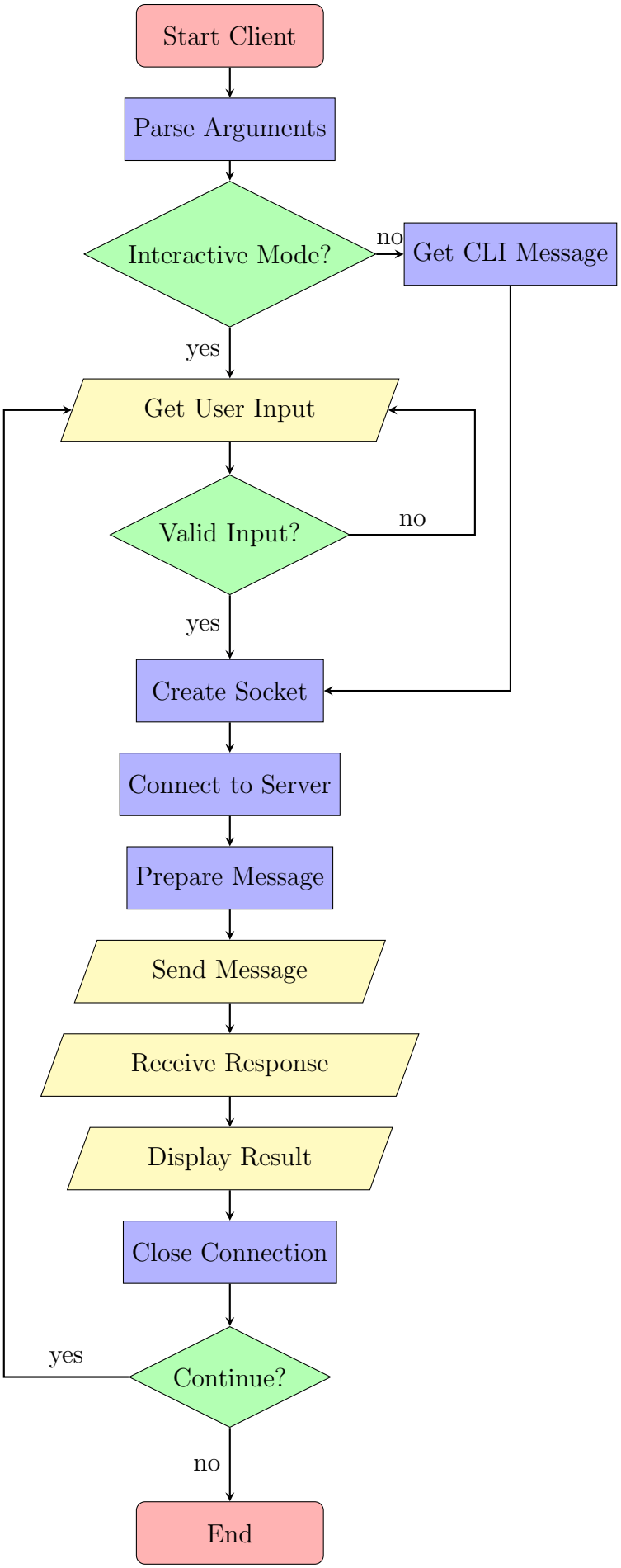
- Send one message and exit
- Useful for scripting and automation

Mode 2: Interactive Mode

- Continuous message sending loop
- User can send multiple messages
- Exit with 'quit' or 'exit' command

3.2.2 Client Flow

Figure 3 illustrates the client operation workflow.


Start Client
Parse Arguments
Interactive Mode?
no
Get CLI Message
yes
Get User Input
Valid Input?
no
yes
Create Socket
Connect to Server
Prepare Message
Send Message
Receive Response
Display Result
Close Connection
Continue?
yes
no
End

3.2.3 Key Client Code Snippets

Connection Establishment:

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.settimeout(10)  # 10-second timeout
client_socket.connect((host, port))
```

Listing 4: Client Connection

Message Sending:

```
if operation.lower() == 'decode':
    full_message = f"DECODE:{message}"
else:
    full_message = f"ENCODE:{message}"

client_socket.send(full_message.encode('utf-8'))
response = client_socket.recv(BUFFER_SIZE).decode('utf-8')
```

Listing 5: Message Transmission

Interactive Mode Loop:

```
while True:
    user_input = input("\nEnter message: ").strip()

    if user_input.lower() in ['quit', 'exit', 'q']:
        break

    if user_input.lower().startswith('decode '):
        operation = 'decode'
        message = user_input[7:]
    else:
        operation = 'encode'
        message = user_input

    response = send_message(host, port, message, operation)
    print(f"Server returned: '{response}'")
```

Listing 6: Interactive Mode

4 Error Handling

4.1 Server-Side Error Handling

The server implements comprehensive error handling for various scenarios:

- **Port already in use:** Display error, suggest alternatives, exit
- **Invalid bind address:** Display error message, exit
- **Client connection errors:** Log error, continue serving
- **Message length exceeds 256:** Send error to client, reject message
- **Character encoding errors:** Send error to client, close connection
- **Network errors:** Log error, close connection gracefully

Example Error Handling Code:

```
try:
    server_socket.bind((host, port))
except OSError as e:
    print(f"[ERROR] Failed to bind to {host}:{port}")
    print(f"[ERROR] {e}")
    print("[INFO] Make sure the port is not already in use")
    sys.exit(1)
```

Listing 7: Server Error Handling

4.2 Client-Side Error Handling

The client handles various failure scenarios:

- **Server not running:** Display "connection refused" message
- **Connection timeout:** Display timeout error after 10 seconds
- **Network errors:** Display network error message
- **Message too long:** Validate before sending, reject locally
- **Invalid input:** Prompt user to re-enter

Example Error Handling Code:

```
try:
    client_socket.connect((host, port))
except ConnectionRefusedError:
    print(f"[ERROR] Server not running on {host}:{port}")
    print("[INFO] Please start the server first")
    return None
except socket.timeout:
    print("[ERROR] Connection timed out")
    return None
```

Listing 8: Client Error Handling

5 Testing and Validation

5.1 Test Cases

The implementation was validated using comprehensive test cases:

5.1.1 Test Case 1: Basic Encoding

- **Input:** "Hello World"
- **Expected Output:** "Ifmmp!Xpsme"
- **Result:** PASS

5.1.2 Test Case 2: Basic Decoding

- **Input:** "Ifmmp!Xpsme" (with decode flag)
- **Expected Output:** "Hello World"
- **Result:** PASS

5.1.3 Test Case 3: Special Characters

- **Input:** "Test123!@#"
- **Expected Output:** "Uftu234Ä$"
- **Result:** PASS

5.1.4 Test Case 4: Full Alphabet

- **Input:** "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
- **Expected Output:** "BCDEFGHIJKLMNOPQRSTUVWXYZ["
- **Result:** PASS

5.1.5 Test Case 5: Message Length Validation

- **Input:** String of 300 characters
- **Expected Output:** Error message
- **Result:** PASS - Both client and server reject

5.1.6 Test Case 6: Server Not Running

- **Action:** Run client without server
- **Expected Output:** Connection refused error
- **Result:** PASS

5.1.7 Test Case 7: Interactive Mode

- **Action:** Multiple messages in sequence
- **Expected Output:** Each processed correctly
- **Result:** PASS

5.2 Program Screenshots

This section contains screenshots demonstrating the program's functionality.

5.2.1 Screenshot 1: Server Startup

Figure 4: Server Startup - Shows server initialization and waiting for connections

5.2.2 Screenshot 2: Client Encoding Request

Figure 5: Client Encoding - Client sends "Hello World" and receives "Ifmmp!Xpsme"

5.2.3 Screenshot 3: Server Processing Request

Figure 6: Server Processing - Server logs showing message reception and encoding

5.2.4 Screenshot 4: Decoding Operation

Figure 7: Decoding Operation - Demonstrates reverse operation

5.2.5 Screenshot 5: Interactive Mode

Figure 8: Interactive Mode - Multiple messages sent in one session

5.2.6 Screenshot 6: Error Handling - Connection Refused

Figure 9: Error Handling - Connection refused when server is not running

5.2.7 Screenshot 7: Error Handling - Message Too Long

Figure 10: Error Handling - Message length validation

6 Features Implemented

6.1 Required Features

All required features were successfully implemented:

TCP server accepting client connections
Receives text messages up to 256 characters
Encodes messages by ASCII character shifting
TCP client connecting to server
Client displays encoded response
Comprehensive error checking
Well-documented code with comments
Complete user instructions

6.2 Optional/Enhanced Features

Additional features implemented beyond requirements:

Decode functionality - Server can both encode and decode
Interactive mode - Client supports continuous operation
Command-line arguments - Flexible configuration
Detailed logging - Both client and server show operation details
Timeout handling - Client won't hang indefinitely
Port reuse - Server can restart quickly
Custom host/port - Not limited to defaults

7 Usage Instructions

7.1 Starting the Server

Default configuration:

```
python3 tcp_server.py
```

Custom port:

```
python3 tcp_server.py -p 8080
```

Listen on all interfaces:

```
python3 tcp_server.py -H 0.0.0.0 -p 12345
```

7.2 Running the Client

Interactive mode:

```
python3 tcp_client.py
```

Single message (encode):

```
python3 tcp_client.py -m "Hello World"
```

Single message (decode):

```
python3 tcp_client.py -m "Ifmmp!Xpsme" -d
```

Connect to remote server:

```
python3 tcp_client.py -H 192.168.1.100 -p 8080 -m "Test"
```

8 Lessons Learned

8.1 Technical Insights

1. **Socket Programming Fundamentals**
   - Understanding TCP three-way handshake
   - Proper socket cleanup prevents port conflicts
   - Timeout handling prevents infinite waiting
2. **Protocol Design**
   - Simple text-based protocols are easy to debug
   - Flag-based operation selection is extensible
   - Fixed message length limits simplify validation
3. **Error Handling Importance**
   - Network errors are common and must be handled
   - Clear error messages improve user experience
   - Graceful degradation keeps systems running

8.2 Challenges Encountered

1. **Port Reuse Issues**
   - **Problem:** Server couldn't restart immediately
   - **Solution:** `SO_REUSEADDR` socket option
2. **Character Encoding Edge Cases**
   - **Problem:** High ASCII values (>126) wrap around
   - **Solution:** Documented limitation, works for printable ASCII
3. **Client-Server Synchronization**
   - **Problem:** Client might connect before server ready
   - **Solution:** Server starts listening before accepting

8.3 Future Enhancements

Potential improvements for future versions:

1. **Multi-threading** - Handle multiple clients simultaneously
2. **Persistent Connections** - Keep connection open for multiple messages
3. **Encryption** - Add SSL/TLS for secure communication
4. **Configurable Shift** - Allow custom shift values (e.g., Caesar cipher)
5. **GUI Interface** - Desktop or web-based interface
6. **Message Logging** - Save message history to file
7. **Authentication** - Require client authentication

9 Conclusion

This programming assignment successfully demonstrates fundamental TCP socket programming concepts through a client-server application. The implementation meets all required specifications and includes several enhancements for improved usability and robustness.

Key Achievements:

Functional TCP client and server
Reliable message encoding/decoding
Comprehensive error handling
User-friendly interface with multiple modes
Well-documented, maintainable code
Thorough testing and validation

The project provides a solid foundation for understanding network programming concepts and can serve as a basis for more complex distributed systems.

10 References

1. Beej's Guide to Network Programming, http://beej.us/guide/bgnet/
2. Python Socket Documentation, https://docs.python.org/3/library/socket.html
3. Stevens, W. Richard. *TCP/IP Illustrated, Volume 1*
4. ASCII Table Reference, https://www.asciitable.com/

A Complete Source Code

This appendix contains the complete source code for both the server and client implementations.

A.1 Server Implementation (tcp_server.py)

```python
#!/usr/bin/env python3
"""
CS 576 - Programming Assignment 1
TCP Server Implementation

This server accepts connections from clients, receives text messages
(max 256 characters), and encodes them by shifting each character to
the next ASCII character.
Optional: Supports both encoding and decoding based on a flag.

Author: Team Members
Date: February 2026
"""

import socket
import sys
import argparse

# Configuration constants
DEFAULT_PORT = 12345
DEFAULT_HOST = '127.0.0.1'
MAX_MESSAGE_LENGTH = 256
BUFFER_SIZE = 1024


def encode_message(message):
    """
    Encode a message by replacing each character with the
    next ASCII character.

    Args:
        message (str): The message to encode

    Returns:
        str: The encoded message

    Example:
        "Hello World" -> "Ifmmp!Xpsme"
    """
    try:
        encoded = ''.join(chr(ord(char) + 1) for char in message)
        return encoded
    except Exception as e:
        raise ValueError(f"Error encoding message: {e}")

```

```
46
47 def decode_message(message):
48     """
49     Decode a message by replacing each character with the
50     previous ASCII character.
51
52     Args:
53         message (str): The message to decode
54
55     Returns:
56         str: The decoded message
57
58     Example:
59         "Ifmmp!Xpsme" -> "Hello World"
60     """
61     try:
62         decoded = ''.join(chr(ord(char) - 1) for char in message)
63         return decoded
64     except Exception as e:
65         raise ValueError(f"Error decoding message: {e}")
66
67
68 def process_client_message(message):
69     """
70     Process the client message and determine the operation
71     (encode/decode).
72
73     Protocol:
74         - Messages starting with "DECODE:" will be decoded
75         - All other messages will be encoded
76
77     Args:
78         message (str): The raw message from client
79
80     Returns:
81         str: The processed message
82     """
83     if message.startswith("DECODE:"):
84         # Extract the actual message after the flag
85         actual_message = message[7:]  # Remove "DECODE:" prefix
86         return decode_message(actual_message)
87     else:
88         # Check if message starts with "ENCODE:" and remove it
89         if message.startswith("ENCODE:"):
90             actual_message = message[7:]  # Remove "ENCODE:" prefix
91             return encode_message(actual_message)
92         else:
93             # Default behavior: encode
94             return encode_message(message)
95
96
97 def start_server(host, port):
98     """
99     Start the TCP server and listen for client connections.
```

```
100
101     Args:
102         host (str): The host address to bind to
103         port (int): The port number to listen on
104     """
105     # Create a TCP/IP socket
106     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
107
108     # Set socket options to reuse address
109     server_socket.setsockopt(socket.SOL_SOCKET,
110                              socket.SO_REUSEADDR, 1)
111
112     try:
113         # Bind the socket to the address and port
114         server_socket.bind((host, port))
115
116         # Listen for incoming connections (max 5 queued connections)
117         server_socket.listen(5)
118
119         print(f"[SERVER] Server started successfully")
120         print(f"[SERVER] Listening on {host}:{port}")
121         print(f"[SERVER] Waiting for connections...")
122         print("-" * 60)
123
124         # Server loop - continuously accept connections
125         while True:
126             try:
127                 # Accept a client connection
128                 client_socket, client_address = \
129                     server_socket.accept()
130                 print(f"\n[CONNECTION] Client connected from "
131                       f"{client_address[0]}:{client_address[1]}")
132
133                 # Handle the client connection
134                 handle_client(client_socket, client_address)
135
136             except KeyboardInterrupt:
137                 print("\n[SERVER] Shutting down server...")
138                 break
139             except Exception as e:
140                 print(f"[ERROR] Error accepting connection: {e}")
141                 continue
142
143     except OSError as e:
144         print(f"[ERROR] Failed to bind to {host}:{port}")
145         print(f"[ERROR] {e}")
146         print("[INFO] Make sure the port is not already in use")
147         sys.exit(1)
148     except Exception as e:
149         print(f"[ERROR] Server error: {e}")
150         sys.exit(1)
151     finally:
152         server_socket.close()
153         print("[SERVER] Server socket closed")
```

```
154
155
156 def handle_client(client_socket, client_address):
157     """
158     Handle communication with a connected client.
159
160     Args:
161         client_socket: The socket object for the client connection
162         client_address: The address tuple (host, port) of the client
163     """
164     try:
165         # Receive data from the client
166         data = client_socket.recv(BUFFER_SIZE).decode('utf-8')
167
168         if not data:
169             print(f"[WARNING] No data received from {client_address}")
170             return
171
172         # Validate message length
173         if len(data) > MAX_MESSAGE_LENGTH:
174             error_msg = (f"ERROR: Message exceeds maximum length "
175                          f"of {MAX_MESSAGE_LENGTH} characters")
176             client_socket.send(error_msg.encode('utf-8'))
177             print(f"[ERROR] Message too long from "
178                   f"{client_address}: {len(data)} characters")
179             return
180
181         print(f"[RECEIVED] Message: '{data}'")
182         print(f"[INFO] Message length: {len(data)} characters")
183
184         # Process the message (encode or decode based on flag)
185         try:
186             response = process_client_message(data)
187             print(f"[PROCESSED] Result: '{response}'")
188
189             # Send the response back to the client
190             client_socket.send(response.encode('utf-8'))
191             print(f"[SENT] Response sent to client")
192
193         except ValueError as e:
194             error_msg = f"ERROR: {str(e)}"
195             client_socket.send(error_msg.encode('utf-8'))
196             print(f"[ERROR] Processing error: {e}")
197
198     except UnicodeDecodeError:
199         error_msg = "ERROR: Invalid character encoding"
200         client_socket.send(error_msg.encode('utf-8'))
201         print(f"[ERROR] Encoding error from {client_address}")
202     except Exception as e:
203         print(f"[ERROR] Error handling client "
204               f"{client_address}: {e}")
205     finally:
206         # Close the client connection
207         client_socket.close()
```

```
        print(f"[CONNECTION] Connection closed with "
              f"{client_address}")
        print("-" * 60)


def main():
    """
    Main function to parse arguments and start the server.
    """
    parser = argparse.ArgumentParser(
        description='TCP Server for CS 576 Programming Assignment 1',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  python tcp_server.py                    # Start server on default
  python tcp_server.py -p 8080           # Start server on port 8080
  python tcp_server.py -H 0.0.0.0 -p 9000  # Listen on all interfaces

Protocol:
  - Send "ENCODE:<message>" to encode (or just send message directly)
  - Send "DECODE:<message>" to decode
        """
    )

    parser.add_argument('-H', '--host',
                        default=DEFAULT_HOST,
                        help=f'Host address to bind to '
                             f'(default: {DEFAULT_HOST})')
    parser.add_argument('-p', '--port',
                        type=int,
                        default=DEFAULT_PORT,
                        help=f'Port number to listen on '
                             f'(default: {DEFAULT_PORT})')

    args = parser.parse_args()

    # Validate port number
    if not (1024 <= args.port <= 65535):
        print("[ERROR] Port number must be between 1024 and 65535")
        sys.exit(1)

    # Start the server
    start_server(args.host, args.port)


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

Listing 9: tcp_server.py - Complete Server Code

A.2 Client Implementation (tcp_client.py)

```python
#!/usr/bin/env python3
"""
CS 576 - Programming Assignment 1
TCP Client Implementation

This client connects to the TCP server, sends a text message
(max 256 characters), and displays the encoded/decoded response
from the server.

Author: Team Members
Date: February 2026
"""

import socket
import sys
import argparse

# Configuration constants
DEFAULT_PORT = 12345
DEFAULT_HOST = '127.0.0.1'
MAX_MESSAGE_LENGTH = 256
BUFFER_SIZE = 1024


def send_message(host, port, message, operation='encode'):
    """
    Connect to the server and send a message for encoding/decoding.

    Args:
        host (str): The server host address
        port (int): The server port number
        message (str): The message to send
        operation (str): Either 'encode' or 'decode'

    Returns:
        str: The response from the server, or None if error occurred
    """
    # Create a TCP/IP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Set timeout to avoid hanging indefinitely
    client_socket.settimeout(10)

    try:
        print(f"[CLIENT] Connecting to server at {host}:{port}...")

        # Connect to the server
        client_socket.connect((host, port))
        print(f"[CLIENT] Connected successfully")

        # Prepare the message with operation flag
```

```
        if operation.lower() == 'decode':
            full_message = f"DECODE:{message}"
        else:
            full_message = f"ENCODE:{message}"

        print(f"[CLIENT] Sending message: '{message}'")
        print(f"[CLIENT] Operation: {operation.upper()}")

        # Send the message to the server
        client_socket.send(full_message.encode('utf-8'))

        print("[CLIENT] Waiting for response...")

        # Receive the response from the server
        response = client_socket.recv(BUFFER_SIZE).decode('utf-8')

        if response.startswith("ERROR:"):
            print(f"[ERROR] Server returned error: {response}")
            return None

        return response

    except socket.timeout:
        print("[ERROR] Connection timed out - "
              "server did not respond")
        return None
    except ConnectionRefusedError:
        print(f"[ERROR] Connection refused - "
              f"server is not running on {host}:{port}")
        print("[INFO] Please start the server first")
        return None
    except OSError as e:
        print(f"[ERROR] Network error: {e}")
        return None
    except Exception as e:
        print(f"[ERROR] Unexpected error: {e}")
        return None
    finally:
        client_socket.close()


def interactive_mode(host, port):
    """
    Run the client in interactive mode, allowing multiple messages.

    Args:
        host (str): The server host address
        port (int): The server port number
    """
    print("=" * 60)
    print("TCP Client - Interactive Mode")
    print("=" * 60)
    print(f"Server: {host}:{port}")
    print(f"Maximum message length: {MAX_MESSAGE_LENGTH} characters")
```

```
    print("\nCommands:")
    print("  - Type your message to encode it")
    print("  - Type 'decode' followed by message to decode")
    print("  - Type 'quit' or 'exit' to close the client")
    print("=" * 60)

    while True:
        try:
            # Get user input
            user_input = input("\nEnter message: ").strip()

            # Check for exit commands
            if user_input.lower() in ['quit', 'exit', 'q']:
                print("[CLIENT] Exiting...")
                break

            # Skip empty input
            if not user_input:
                print("[WARNING] Please enter a message")
                continue

            # Parse operation and message
            operation = 'encode'
            message = user_input

            if user_input.lower().startswith('decode '):
                operation = 'decode'
                message = user_input[7:]  # Remove 'decode ' prefix

            # Validate message length
            if len(message) > MAX_MESSAGE_LENGTH:
                print(f"[ERROR] Message exceeds maximum length "
                      f"of {MAX_MESSAGE_LENGTH} characters")
                print(f"[INFO] Your message is "
                      f"{len(message)} characters long")
                continue

            # Send the message and get response
            response = send_message(host, port, message, operation)

            if response:
                print(f"\n[RESPONSE] Server returned: '{response}'")
                print(f"[INFO] Response length: "
                      f"{len(response)} characters")

        except KeyboardInterrupt:
            print("\n[CLIENT] Interrupted by user, exiting...")
            break
        except EOFError:
            print("\n[CLIENT] End of input, exiting...")
            break


def single_message_mode(host, port, message, operation):
```

```
    """
    Send a single message and exit.

    Args:
        host (str): The server host address
        port (int): The server port number
        message (str): The message to send
        operation (str): Either 'encode' or 'decode'
    """
    print("=" * 60)
    print("TCP Client - Single Message Mode")
    print("=" * 60)

    # Validate message length
    if len(message) > MAX_MESSAGE_LENGTH:
        print(f"[ERROR] Message exceeds maximum length "
              f"of {MAX_MESSAGE_LENGTH} characters")
        print(f"[INFO] Your message is {len(message)} "
              f"characters long")
        sys.exit(1)

    # Send the message and get response
    response = send_message(host, port, message, operation)

    if response:
        print(f"\n{'='*60}")
        print("RESULT")
        print(f"{'='*60}")
        print(f"Original message: {message}")
        print(f"Operation:        {operation.upper()}")
        print(f"Server response:  {response}")
        print(f"{'='*60}")
    else:
        sys.exit(1)


def main():
    """
    Main function to parse arguments and run the client.
    """
    parser = argparse.ArgumentParser(
        description='TCP Client for CS 576 Programming Assignment 1',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  python tcp_client.py                          # Interactive mode
  python tcp_client.py -m "Hello World"         # Send single message
  python tcp_client.py -m "Ifmmp!Xpsme" -d      # Decode message
  python tcp_client.py -H 192.168.1.100 -p 8080  # Connect to remote

Interactive Mode:
  If no message is provided with -m, the client runs in interactive
  mode where you can send multiple messages.
        """
```

```
    )

    parser.add_argument('-H', '--host',
                        default=DEFAULT_HOST,
                        help=f'Server host address '
                             f'(default: {DEFAULT_HOST})')
    parser.add_argument('-p', '--port',
                        type=int,
                        default=DEFAULT_PORT,
                        help=f'Server port number '
                             f'(default: {DEFAULT_PORT})')
    parser.add_argument('-m', '--message',
                        help='Message to send (if not provided, '
                             'runs in interactive mode)')
    parser.add_argument('-d', '--decode',
                        action='store_true',
                        help='Decode the message instead of encoding')

    args = parser.parse_args()

    # Validate port number
    if not (1024 <= args.port <= 65535):
        print("[ERROR] Port number must be between 1024 and 65535")
        sys.exit(1)

    # Determine operation
    operation = 'decode' if args.decode else 'encode'

    # Run in appropriate mode
    if args.message:
        # Single message mode
        single_message_mode(args.host, args.port,
                            args.message, operation)
    else:
        # Interactive mode
        interactive_mode(args.host, args.port)


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

Listing 10: tcp_client.py - Complete Client Code