TCP Socket Client-Server Example with Step-by-Step Walkthrough

Overview

This example shows how the client and server communicate using a custom protocol with headers. Let's trace through several message exchanges with actual data.

The Protocol Format

```
[10-byte header][message data]
```

- Header: Message length as 10-digit string with leading zeros
- Message: The actual text data

Example 1: Sending "Hello"

Client Side:

python

```python
# User types: "Hello"
message = "Hello"  # Length = 5

# Create header
header = f"{len(message):0{HEADER_SIZE}d}".encode("utf-8")
# f"{5:010d}" = "0000000005"
# header = b"0000000005"

# Send complete message
con.send(header + message.encode("utf-8"))
# Sends: b"0000000005Hello"
```

Server Side Reception:

python

```python
# Server receives in chunks of 15 bytes
# Chunk 1: con.recv(15) returns b"0000000005Hello" (15 bytes)

# First chunk processing (newmsg = True):
msg_length = int(data[:10].decode("utf-8"))  # "0000000005" → 5
message += data[10:].decode("utf-8")         # "Hello"
newmsg = False

# Check if complete: len("Hello") = 5 >= 5 ✓
# Complete message received: "Hello"
```

Server Echo Back:

python

```python
# Server echoes back
header = f"{len(message):0{HEADER_SIZE}d}".encode("utf-8")
# header = b"0000000005"
con.send(header + message.encode("utf-8"))
# Sends: b"0000000005Hello"
```

Client Reception:

python

```python
# Client receives echo in chunks
# Chunk 1: con.recv(15) returns b"0000000005Hello"

# First chunk processing:
msg_length = int(data[:10].decode("utf-8"))  # 5
message += data[10:].decode("utf-8")         # "Hello"

# Check: len("Hello") = 5 >= 5 ✓
# Prints: "Server echoed: Hello"
```

Example 2: Sending "This is a longer message"

Client Side:

python

```python
# User types: "This is a longer message"
message = "This is a longer message"  # Length = 26

# Create header
header = f"{26:010d}".encode("utf-8")
# header = b"0000000026"

# Send complete message
con.send(header + message.encode("utf-8"))
# Sends: b"0000000026This is a longer message" (36 bytes total)
```

Server Side Reception (Multiple Chunks):

python

```python
# Server receives in 15-byte chunks:
# Chunk 1: con.recv(15) returns b"0000000026This" (15 bytes)
# Chunk 2: con.recv(15) returns b" is a longer m" (15 bytes)
# Chunk 3: con.recv(15) returns b"essage" (6 bytes)

# Processing Chunk 1 (newmsg = True):
msg_length = int(data[:10].decode("utf-8"))  # 26
message += data[10:].decode("utf-8")         # "This"
newmsg = False
# Check: len("This") = 4 < 26, continue...

# Processing Chunk 2 (newmsg = False):
message += data.decode("utf-8")  # "This" + " is a longer m" = "This is a longer m"
# Check: len("This is a longer m") = 19 < 26, continue...

# Processing Chunk 3 (newmsg = False):
message += data.decode("utf-8")  # "This is a longer m" + "essage" = "This is a longer message"
# Check: len("This is a longer message") = 26 >= 26 ✓
# Complete message received!
```

Example 3: Network Buffer Visualization

Let's see exactly how data flows through the network:

Message: "Hello World!" (Length = 12)

Client sends:

```
Bytes sent: [0][0][0][0][0][0][0][0][1][2][H][e][l][l][o][ ][W][o][r][l][d][!]
Positions:   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
          |--------Header (10 bytes)--------|-------Message (12 bytes)-------|
```

Server receives in chunks:

```
Chunk 1 (15 bytes): [0][0][0][0][0][0][0][0][1][2][H][e][l][l][o]
                    |--------Header---------|--Message start-|

Chunk 2 (7 bytes):  [ ][W][o][r][l][d][!]
                    |--Message end-|
```

Example 4: Error Handling Example

Client Disconnects Unexpectedly:

python

```python
# Server side - client disconnects while sending
data = con.recv(BUF_SIZE)
if not data:  # Returns empty bytes b''
    print(f"Client {addr} disconnected")
    break  # Exit message handling loop
```

Server Stops While Client is Connected:

python

```python
# Client side - server stops responding
data = con.recv(BUF_SIZE)
if not data:  # Returns empty bytes b''
    print("Server disconnected")
    break  # Exit reception loop
```

Complete Flow Example

Server Console Output:

```
Server listening on 127.0.0.1:1236...
Connected to client: ('127.0.0.1', 54321)
Client sent: Hello
Client sent: This is a test message
Client sent: exit
Finished serving client: ('127.0.0.1', 54321)
```

Client Console Output:

```
Connected to server at 127.0.0.1:1236
Enter a message ("exit" to quit): Hello
Sent to server: Hello
Server echoed: Hello
Enter a message ("exit" to quit): This is a test message
Sent to server: This is a test message
Server echoed: This is a test message
Enter a message ("exit" to quit): exit
Sent to server: exit
Server echoed: exit
Connection closed.
```

Key Protocol Features Demonstrated

1. Header-Based Length Prefix

python

```python
# Always 10 bytes, zero-padded
"Hello" → "0000000005Hello"
"A" → "0000000001A"
"This is a very long message" → "0000000027This is a very long message"
```

2. Chunk Reconstruction

python

```python
# Message gets split across network packets
Original: "0000000015Hello World Test"
Chunk 1:  "0000000015Hello"  # 15 bytes
Chunk 2:  " World Test"      # 11 bytes
Result:   "Hello World Test"  # Reconstructed correctly
```

3. Buffer Management

python

```python
# Small buffer (15 bytes) forces chunking
BUF_SIZE = 15

# Messages longer than 15 bytes require multiple recv() calls
# This simulates real network conditions where data arrives in chunks
```

4. State Tracking

python

```python
# Variables track message assembly state
newmsg = True     # First chunk of new message
msg_length = 0    # Expected total length
message = ""      # Assembled message so far
```

Why This Design?

1. **Reliable Message Boundaries**: Headers ensure we know exactly how much data to expect
2. **Handles Network Chunking**: Real networks split data unpredictably
3. **Protocol Flexibility**: Can handle messages of any length
4. **Error Detection**: Can detect incomplete messages
5. **Streaming Support**: Can handle continuous message streams

This protocol design is similar to many real-world protocols like HTTP, where headers specify content length, ensuring reliable message transmission over unreliable network connections.