Programming Assignment 1: Socket Programming

CS 352 Internet Technology, Fall 2025

Released: September 15, 2025 | **Due:** October 3, 2025

Introduction

The goal of this project is to give you some hands-on practice with both Python and the basics of how two programs can talk to each other over a network. Don't worry if this is your first time writing socket code, we'll build things up step by step. You'll be working with starter files that guide you through each task, and by the end you'll have two separate programs: a server and a client. These two will be able to connect, send messages back and forth, and process text in simple but interesting ways.

We'll begin with a simple first step: when a client connects to the server, the server will send a short greeting message to confirm the connection. From there, you'll expand the programs into an echo service, then add a transformation that flips strings around, and finally, you'll hook everything up to files so you can send and save multiple messages automatically. Along the way, you'll pick up important networking concepts like ports, connections, message framing, and how to handle input/output with Python.

Important Instructions

- **Individual assignment**.
- Run and test only on **Rutgers iLab machines**.
- Use **Python 3**.
- We will grade by running:

```
python3 server.py
# (in another terminal on the same host)
python3 client.py
```

- You are provided with **starter code for each step** (`pa1-stepX-server.py`, `pa1-stepX-client.py`). Do not forget to save your work for each step; each file will be required for submission.

Python Calls and their function

Here are the main Python functions you will use when working with sockets:

- **socket.socket(family, type)**
  Creates a new socket object. You can think of this like plugging in a phone.
  For TCP connections over IPv4, use `socket.AF_INET` for the family and `socket.SOCK_STREAM` for the type.
- **setsockopt(level, optname, value)**
  Changes options on a socket. In this project we use it to allow the server to restart quickly without waiting (`SO_REUSEADDR`).

- **bind((host, port))**
  Attaches the server socket to an address and port on the local machine.
  Example: binding to port 50007 means "listen for calls on port 50007."
- **listen(backlog)**
  Marks the socket as a server socket and starts waiting for incoming calls.
  The `backlog` number controls how many connections can wait in line.
- **accept()**
  Waits until a client connects.
  Returns a new socket (for that client only) and the client's address.
  Think of this as picking up an incoming call.
- **connect((host, port))**
  Used by the client. This "dials" the server's host and port to start the connection.
- **sendall(data)**
  Sends data over the connection. The argument must be in bytes.
  "All" means it keeps sending until all the data has been transmitted.
- **recv(bufsize)**
  Reads data from the connection. The number you provide is the maximum size (in bytes) to read at once. If it ever returns empty (`b""`), it means the other side closed the connection.
- **close()**
  Closes the socket and frees the operating system resources.
  Like hanging up the phone.
- **getsockname()**
  Returns the local endpoint (IP and port) that this socket is currently using.
- **getpeername()**
  Returns the remote endpoint (IP and port) this socket is connected to.

You will also make use of standard Python functions for text and files:

- **encode()**
  Converts a string (human text) into bytes, so it can be sent over a socket.
- **decode()**
  Converts bytes received from a socket back into a string for printing.
- **open(filename, mode)**
  Opens a file on disk. For example, `mode="r"` for reading or `mode="w"` for writing.
- **read()/write()** and **for line in file**
  Standard file operations used to process input and output line by line.

Step 1: Minimal Client and Server

What this step is about: Make your first network connection. We will build a small server that occupies a local port and waits, and a client that connects to that port. Once connected, the server will send a short greeting that the client prints. This shows how a listening endpoint and a connecting endpoint meet to form a conversation channel.

Server Setup

Set up the server's doorway to the network. The server creates a TCP endpoint, enables a quick-restart behavior for development, attaches to a chosen port on the local machine, and begins waiting for calls. Printing the bound address helps confirm the server is listening where you expect.

```
# server.py
import socket
PORT = 50007

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# TODO: allow quick restarts with SO_REUSEADDR
# TODO: bind to all interfaces on PORT
# TODO: listen with backlog 1
print("[S] listening on", s.getsockname())
```

Client Setup

Program the client to dial the server. The client creates its own TCP endpoint and initiates a connection to the server's address and port. After connecting, printing the local and remote endpoints shows both sides of the established connection and helps you verify you reached the intended server.

```
# client.py
import socket
HOST = "127.0.0.1"
PORT = 50007

c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# TODO: connect to (HOST, PORT)
print("local endpoint:", c.getsockname())
print("server endpoint:", c.getpeername())
```

Accept and Greet

When a client arrives, the server gets a new, dedicated channel for that client while the original server endpoint remains available for others. Use this client-specific channel to send a brief greeting. On the client side, read a small amount of data, convert it from bytes to text, and display it.

```
# In server.py
conn, addr = s.accept()
print("[S] accepted connection from", addr)
# TODO: send greeting to client
```

```
# In client.py
# TODO: receive up to 100 bytes, decode, and print
```

Shutdown

Close the client-specific channel after finishing the exchange, and close the listening endpoint when the server is done. The client should also close its socket when finished. Clean shutdown releases operating system resources and prevents issues in later steps.

```
# In server.py
conn.close()
s.close()
```

```
# In client.py
c.close()
```

Question

Explain in your own words the role of the listening socket versus the accepted socket. Why do we need two different sockets on the server side?

Step 2: Echo Loop

What this step is about: Build a simple echo service so you experience TCP as a continuous stream of bytes. The server repeatedly reads whatever bytes the client sends and immediately sends the same bytes back. The client sends user-provided text and displays the server's echo in response.

Server Loop

Reading from a connection can return different-sized chunks depending on timing and buffering. Keep reading until the client closes the connection (which appears as an empty read), and for each non-empty piece received, send it back unchanged. This reinforces the idea that TCP does not preserve message boundaries by itself.

```
# In server.py, after accept()
while True:
    data = conn.recv(1024)
    if not data:
        break
    # TODO: echo it back with conn.sendall()
```

Client Loop

Prompt the user for a line of input and send it to the server. After sending, wait for the echoed bytes to arrive, convert them to text, and print them out. If the user types the designated word to quit, end the loop and close the connection; the server will detect the closure from its side.

```
# In client.py, after connect()
while True:
    line = input("Enter a line (or QUIT): ")
    if line == "QUIT":
        break
    # TODO: send line (encode)
    # TODO: receive reply and print
```

Questions

(2.a) Why must the server continue calling `recv()` in a loop instead of just once? What happens if multiple messages are sent or if a message is larger than the buffer size?

(2.b) TCP does not indicate where one message ends. Describe how the client and server in your echo design can still tell when to stop reading and when a conversation has finished.

Step 3: Define a Protocol

What this step is about: Add meaning to messages with a tiny application protocol. Instead of merely echoing, the server will transform the client's text and reply with the result. Both sides agree that each message ends with a newline so messages can be separated correctly over the byte stream.

Transform Function

Implement a small, self-contained text function that takes a string and returns a new string with the characters reversed and letter case swapped. Keeping this logic independent of networking simplifies testing and reuse within your server code.

```
def transform(s: str) -> str:
    # TODO: reverse string and swap case
    return ???
```

Server Apply

When data arrives from the connection, treat it as text, apply your transformation, and then send the transformed text back with a newline at the end. If several messages arrive back-to-back, handle them one by one; if a message is incomplete (no newline yet), wait for more data.

```
# After receiving data
text = data.decode()
reply = transform(text)
# TODO: send reply + newline
```

Client Print

Send each input line with a newline so the server can find the end. Then keep reading from the connection until the newline that marks the end of the reply. Convert the reply to text and print both the original and the result in the provided format.

```
print(f"sent={line} recv={reply}")
```

Question

TCP gives a continuous stream of bytes with no message boundaries. Why is it useful to define your own simple protocol on top of TCP? Explain specifically how the newline character helps both sides agree on message boundaries.

Step 4: File I/O Integration

What this step is about: Automate the exchange. The client will read many inputs from the file `in-proj.txt` and send them as separate messages, while the server will process each message and also write the results to an output file. This mirrors workloads where programs batch-process text and record outputs for later use.

Client Reads from File

Open `in-proj.txt` and loop over its lines. Remove the file's trailing newline so you don't send two newlines, then send exactly one newline to mark the end of each message. Wait for the server's reply, convert it to text, and print both the original and result to confirm correctness.

```
# client.py (Step 4)
with open("in-proj.txt") as f:
    for line in f:
        msg = line.strip()
        # TODO: send msg (+ "\n") to the server
        # TODO: receive the transformed reply from the server
        # decode it back to string
        print(f"sent={msg} recv={reply}")
```

Server Writes to File

Open the output file once before processing inputs, write one transformed line per message, and close it when done to flush all data. For each message, convert incoming bytes to text, apply the transformation, write the result with a newline to the file, and send the same newline-terminated result back to the client.

```
# server.py (Step 4)
out = open("out-proj.txt", "w")
while True:
    data = conn.recv(1024)
    if not data:
        break
    text = data.decode().strip()
    reply = transform(text)
    out.write(reply + "\n")
    # TODO: send reply back to client (with newline)
out.close()
```

Questions

(4.a) How does replacing interactive input with file-based input change the behavior of your client program? Discuss the advantages of automation here.

(4.b) Why is line-by-line processing a natural model for combining file I/O with sockets? Explain how this matches with the protocol chosen in the previous step.

Submission

Submit a single Zip file named `pa1-{netid}.zip` with:

- `servers/` with `pa1-stepX-server.py`
- `clients/` with `pa1-stepX-client.py`
- `report.pdf` with answers

Policies

- Individual work only.
- Use only Python standard library.
- Code must run on iLab with Python 3.