Assignment 1: TCP Client-Server with Urgent Data Handling

Due Date: 2025-05-19

Submission: Submit your `server.cpp` and your `client.cpp` through Totara.

Objective

The goal of this assignment is to implement a basic TCP client and server in C++ that can exchange normal data and also demonstrate the handling of TCP's "Urgent Data" mechanism (signaled by the URG flag). Your submission will be automatically graded based on its adherence to the specified communication protocol.

Background

TCP provides a mechanism for "urgent data" (also known as out-of-band data). When the URG flag is set in a TCP segment, it indicates that the "Urgent Pointer" field in the TCP header is significant. This pointer typically points to the byte *following* the urgent data within the data stream.

- **Sending Urgent Data:** The `send()` system call can be used with the `MSG_OOB` flag to send a single byte of urgent data.
- **Receiving Urgent Data:** When `recv()` is called with `MSG_OOB`, it will attempt to read only the single out-of-band byte.

For this assignment, the server will attempt to read OOB data when a specific command is received from the client.

Requirements

- **Server (`server.cpp`)**

1. **Setup:**
   - Create a TCP listening socket.
   - The server **must** listen on **IP Address `127.0.0.1`** and **Port `8080`**. This is critical for automated testing.
   - Use the `SO_REUSEADDR` socket option to allow quick restarts.
   - The server should be able to handle one client at a time (a simple iterative server is sufficient; you do not need

`select`/`poll` or multi-threading for this assignment).
   - Robust error checking for all socket API calls is required.
2. **Communication Protocol:**
   - When a client connects, the server should be ready to receive messages.
   - **Normal Data:** If the server receives the exact string `"NORMAL_DATA:Hello"` (without newline), it **must** respond with the exact string `"SERVER_ACK:Hello"` (without newline).
   - **Urgent Data Command:** If the server receives the exact string `"SEND_URGENT_REQUEST"` (without newline), it **must** then attempt to receive one byte of out-of-band (urgent) data from the client using `recv(client_sock, &oob_byte, 1, MSG_OOB);`.
     - If an urgent byte is successfully received, the server **must** respond with the string `"SERVER_URGENT_ACK:"` followed by the character received (e.g., `"SERVER_URGENT_ACK:X"` if `'X'` was the urgent byte, without newline).
     - If `recv()` with `MSG_OOB` fails (returns `-1`) or returns `0` (indicating no OOB data at the mark or an issue), the server **must** respond with the exact string `"SERVER_NO_URGENT_DATA"` (without newline).
   - **Other Data:** If any other message is received, the server **must** respond with the exact string `"SERVER_UNKNOWN_COMMAND"` (without newline).
   - **Message Termination:** Assume messages are not newline-terminated unless specified. Your server should handle the exact strings provided.
3. **Error Handling:** Implement robust error checking for all socket API calls (`socket`, `setsockopt`, `bind`, `listen`, `accept`, `send`, `recv`). Print descriptive error messages to `stderr` using `perror()`.
4. **Shutdown:** The server should continue to accept new clients after one disconnects. For testing, it will be started and stopped externally.

- **Client (`client.cpp`)**

1. **Setup:**
   - Create a TCP socket.
   - The client **must** be able to connect to a server at a given IP address and port, which will be `127.0.0.1` and `8080` for testing against your own server.

- (For your own testing, you can make the IP/port configurable, but ensure it defaults or can easily be set to `127.0.0.1:8080`).

2. **Functionality (for your own testing and to ensure you meet server requirements):**
   Your client should be capable of performing the following actions to test your server:
   - **Scenario 1 (Normal Data):**
     - Connect to the server.
     - Send `"NORMAL_DATA:Hello"` to the server.
     - Receive and print the server's response.
     - Close the connection.
   - **Scenario 2 (Urgent Data) (VG grade):**
     - Connect to the server.
     - Send `"SEND_URGENT_REQUEST"` to the server.
     - Immediately after, send a single byte of urgent data (e.g., the character 'U') using `send(sock, "U", 1, MSG_OOB);`.
     - (Optional but good practice for testing: send some normal data *after* the urgent byte, e.g., `send(sock, "trailing", 8, 0);` to ensure the urgent byte is distinct).
     - Receive and print the server's response.
     - Close the connection.
   - **Scenario 3 (Urgent Data Request, but No Urgent Data Sent) (VG grade):**
     - Connect to the server.
     - Send `"SEND_URGENT_REQUEST"` to the server.
     - Do *not* send any urgent data. You can send some normal data instead (e.g., `"NORMAL_AFTER_REQUEST"`) or close the sending side (`shutdown(sock, SHUT_WR)`).
     - Receive and print the server's response.
     - Close the connection.
   - **Scenario 4 (Unknown Command):**
     - Connect to the server.
     - Send an arbitrary string like `"TEST_GARBAGE"` to the server.
     - Receive and print the server's response.
     - Close the connection.
3. **Error Handling:** Implement robust error checking for all socket API calls in your client.
4. **Output:** The client should print what it sends and what it receives for clarity during your manual testing. This output is not directly graded but helps you debug.

Grading Criteria

Your submission will be primarily graded by an automated test suite that will:

- Start your `server` executable.
- Use a test client (similar to the functionality described for your `client.cpp`) to interact with your server.
- Verify that your server responds *exactly* as specified in the "Communication Protocol" section for each scenario.
- Check for correct handling of connections and disconnections.

Therefore, adherence to the specified IP, port, and exact message strings is **critical**.

- **Correctness & Protocol Adherence (80%):**
  - Server binds to `127.0.0.1:8080`.
  - Server correctly implements the `SO_REUSEADDR` option.
  - Server correctly handles the defined communication protocol for normal data, urgent data requests, actual urgent data, and unknown commands, providing the *exact* specified responses.
  - Server handles client connections and disconnections gracefully.
  - Client (as a tool for you to build a correct server) demonstrates the ability to interact according to the protocol.
- **Error Handling & Code Quality (20%):**
  - Robust error checking (checking return values, using `perror`) for socket API calls in both server and client.
  - Sockets are properly closed.
  - Code is well-formatted, readable, and reasonably commented.
  - `CMakeLists.txt` correctly builds both executables with appropriate flags.

Hints and Tips

- **Exact Strings:** Pay very close attention to the exact strings required for commands and responses (e.g., `"NORMAL_DATA:Hello"`, `"SERVER_ACK:Hello"`). Do not add extra spaces, newlines, or change capitalization unless specified.
- **Server Port/IP:** Your server **must** use `127.0.0.1` and port `8080` for the automated tests to work.
- **`MSG_OOB` with `send()`:** Typically sends one byte as urgent.

- **`MSG_OOB` with `recv()`:** Attempts to read one urgent byte. Check its return value carefully. If it's `-1` and `errno` is `EWOULDBLOCK` (on a non-blocking socket) or `EINVAL` (often if no urgent data is pending), it means no OOB data was read. A return of `0` is also possible.
- **Testing Your Server:** Use your own `client.cpp` extensively to test all scenarios before submitting. You are essentially building the tool that will help you pass the automated grading.
- **`shutdown(sockfd, SHUT_WR)`:** In your client for Scenario 3, after sending `"SEND_URGENT_REQUEST"`, you might use `shutdown(sock, SHUT_WR)` to signal to the server that your client will send no more data. This can help the server's `recv()` return `0` if it's expecting more normal data before checking for OOB. This is an alternative to just sending other normal data.