Assignment 2: Robust Concurrent TCP Server with I/O Multiplexing and Threaded Request Processing

Due Date: 2025-05-28
Submission: Submit your `robust_server.cpp`, a `CMakeLists.txt` file, and a `README.md` explaining your design choices, how to build/run, and how you addressed the robustness requirements.

Objective

The goal of this assignment is to design and implement a robust TCP server in C++ capable of handling multiple concurrent clients efficiently. The server will use an I/O multiplexing mechanism (`select` or `poll`) to manage client connections and detect incoming data. Actual data processing for each client request will then be offloaded to a worker thread (potentially from a thread pool) to prevent CPU-bound tasks from blocking the main I/O loop.

Your server must be resilient to various "messy" client behaviors.

Background

We've explored:

1. **I/O Multiplexing (`select` / `poll`):** Allows a single thread to monitor many file descriptors for I/O readiness, preventing blocking on any single client.
2. **Threading (`std::thread`):** Allows concurrent execution of tasks, useful for offloading CPU-intensive work or handling client requests in parallel.

This assignment combines these. The main server thread will use `select` (or `poll`) for I/O event detection. When data arrives from a client, instead of processing it directly in the I/O loop (which could block if processing is slow), the task of processing that data will be handed off to a worker thread.

Core Architecture Requirements

1. **Main I/O Thread:**
    - Responsible for:
        - Setting up the listening TCP socket on a predefined IP (e.g., `127.0.0.1`) and port (e.g., `8080`). Use `SO_REUSEADDR`.

    - Using `select()` or `poll()` to monitor the listening socket for new connections and all connected client sockets for incoming data.
    - Accepting new connections and adding their sockets to the monitored set.
    - When `select()`/`poll()` indicates a client socket is readable, it should read the available data (handle partial reads if necessary to get a "complete" request, or define a simple request delimiter).
    - **Crucially:** Instead of processing the request and sending a response directly in this I/O thread, it should **dispatch the request (e.g., the client socket descriptor and the received data) to a worker thread for processing.**
2. **Worker Threads / Thread Pool:**
    - You must implement a mechanism for processing client requests using threads. Options include:
        - **Simple Thread-per-Request (Less Ideal but Acceptable for a first pass):** For each complete request received by the I/O thread, spawn a new `std::thread` to process it and send the response. Remember to `detach()` or manage joining.
        - **Thread Pool (Preferred & More Robust):** Implement a fixed-size thread pool. The I/O thread places tasks (client socket + data) onto a synchronized task queue. Worker threads in the pool pick up tasks from this queue, process them, and send responses. This requires `std::mutex` and `std::condition_variable` for the task queue.
3. **Non-Blocking Sockets:** All client sockets (and the listening socket for `accept`) should be set to non-blocking mode using `fcntl()`.

Application Protocol (Simple Echo or Basic Command Server)

To keep the focus on the networking architecture, the application protocol should be simple:

- **Echo Server:** The server reads a line of text (terminated by a newline `\n`) from the client, and the worker thread echoes the same line back to the client.
- **OR Basic Command Server (Choose one command):**
    - Client sends: `"TIME\n"` -> Worker thread responds with current server time.
    - Client sends: `"UPPERCASE:some text\n"` -> Worker thread responds with `"SOME TEXT\n"`.
    - **Clearly document which protocol you implemented.**

Robustness Requirements ("Messy" Client Handling)

Your server **must** gracefully handle the following scenarios without crashing or becoming unresponsive to other clients:

1. **Multiple Concurrent Connections:** Successfully serve at least 5-10 clients connected and sending data simultaneously.
2. **Partial Reads/Writes:**
   - Your I/O thread should be prepared for `recv()` to return fewer bytes than requested. If your protocol defines messages (e.g., newline-terminated), you'll need to buffer incoming data until a complete message is received before dispatching to a worker.
   - Worker threads should be prepared for `send()` to send fewer bytes than requested and loop if necessary to send the entire response.
3. **Slow Clients:**
   - **Slow Data Sender:** A client that sends data very slowly (e.g., one byte at a time with pauses). Your server should not get stuck on this client.
   - **Slow Data Receiver:** A client that reads the server's response very slowly. Your worker thread's `send()` might block or return `EAGAIN` if non-blocking. The worker thread should handle this without freezing other operations (this is where non-blocking sends in workers or careful buffering becomes important).
4. **Abrupt Disconnections:** A client that closes the connection without a proper TCP FIN (e.g., process killed). `recv()` should return `0` or `-1` with `ECONNRESET`. The server must detect this, close its end of the socket, and remove the client from its active set.
5. **Clients Sending Excessive Data (Basic Handling):**
   - If a client sends a single "message" (e.g., a line for the echo server) that is larger than your server's reasonable buffer (e.g., > 4KB), your server should handle this gracefully. It could either:
     - Process the first part up to its buffer limit and respond.
     - Or, detect the oversized message, send an error response (e.g., `"ERROR: Message too long\n"`), and close the connection.
   - The server should not crash due to a buffer overflow.
6. **Client Timeout (Application-Level):**
   - Implement a simple timeout for idle clients. If a connected client sends no data for a specified period (e.g., 30-60 seconds), the server should close that client's connection. This requires tracking the last activity time for each client.

`CMakeLists.txt`

- Provide a `CMakeLists.txt` that can build your `robust_server` executable.

- Use C++11 or newer.
- Enable reasonable compiler warnings (e.g., `-Wall -Wextra -pedantic`).
- Link against the pthreads library (`-pthread` flag for g++).

`README.md`

- Explain your server's architecture (how `select` / `poll` interacts with your threading model – thread-per-request or thread pool).
- Describe your chosen application protocol (echo or specific command).
- Detail how you addressed each of the "Robustness Requirements."
- Instructions on how to compile and run your server.
- Any known limitations or assumptions made.

Grading Criteria

- **Correctness of Concurrent Architecture (50%):**
    - Proper use of `select()` or `poll()` for I/O event detection in the main thread.
    - Client sockets are non-blocking.
    - Successful dispatch of requests from the I/O thread to worker threads.
    - Correct implementation of the chosen threading model (thread-per-request or thread pool).
    - If using a thread pool: correct synchronization of the task queue.
    - Server handles multiple clients concurrently without blocking the accept loop or other clients unnecessarily.
- **Robustness & "Messy" Client Handling (30%):**
    - Demonstrable handling of the specified robustness requirements (partial reads/writes, slow clients, abrupt disconnections, basic oversized message handling, client timeout).
    - Server remains stable and responsive under these conditions.
- **Application Protocol & Error Handling (10%):**
    - Correct implementation of the chosen simple application protocol.
    - Thorough error checking for socket API calls and system calls.
    - Sockets are properly closed, and resources are managed.
- **Code Quality & Documentation (10%):**
    - Code is well-formatted, readable, and reasonably commented.
    - `CMakeLists.txt` is correctly configured.
    - `README.md` is clear and provides the required explanations.

Hints and Tips

- **Start Incrementally:**
  1. Get a `select()` / `poll()` based server working that handles I/O in the main thread (like Day 4 lab).
  2. Then, introduce the threading model: first, maybe a simple thread-per-request.
  3. If aiming for a thread pool, implement the synchronized queue and worker threads.
- **Client State:** You'll need to manage state for each connected client (at least its socket FD, maybe a buffer for partial messages, last activity timestamp). A `std::map<int, ClientStateStruct>` or similar can be useful.
- **Partial Message Buffering:** When `recv()` returns, append data to a client-specific buffer. Check if this buffer now contains a complete message (e.g., ends with `\n`). If so, extract the message, process it, and remove it from the buffer.
- **Thread Pool Task Queue:** A `std::queue` protected by a `std::mutex` and coordinated with a `std::condition_variable` is a common way to implement this. Worker threads wait on the condition variable when the queue is empty. The I/O thread pushes tasks and notifies a worker.
- **Testing "Messy" Clients:** You might need to write small, specialized client programs (or use `netcat` creatively with pipes and `sleep`) to simulate some of these behaviors for your own testing. For example:
  - `echo -n "Hello" | nc localhost 8080` (sends "Hello" without newline)
  - `(echo -n "FirstPart"; sleep 5; echo "SecondPart") | nc localhost 8080` (slow client)
- **Timeouts:** In your `select()` / `poll()` loop, you can use the timeout argument. If `select` / `poll` returns 0 (timeout), iterate through your connected clients and check their last activity time.