Project 1: Socket Programming Practice

This project began with the assumption that the provided code from the lecture notes would be a sufficient foundation. Initially, I constructed the client and server using nearly identical logic to the example -- single-process, single-threaded, with `read()` and `write()` calls assumed to send and receive full payloads in one go. This approach quickly broke down under real usage.

The first issue was concurrency. The example server could only handle one connection at a time. To support multiple clients, I moved to a prefork model using `fork()` to spawn 5 child processes. Each child blocks on `accept()` and independently handles a connection. This design ensured at least 5 concurrent clients could be served without blocking.

Next came the realization that `read()` does not guarantee complete data in one call. I had initially assumed the client could send a header and body in a single write, and the server could receive it in a single read. In practice, TCP stream behavior causes headers and bodies to arrive fragmented. I added logic to read one byte at a time until \r\n\r\n was found, up to a 1024-byte cap. Anything beyond that is considered malformed.

Parsing the header required more than simple string checks. I implemented `parse_request_header()` to tokenize lines, normalize field names, and validate that `Host:` and `Content-length:` exist. The function also parses and verifies the first line (`POST message SIMPLE/1.0`). These checks were not in the example, but were required to meet the spec.

On the client side, the example used `fgets()` and `strlen()` to read input and compute message length. This failed with binary input. I replaced it with `fread()` to support up to 10MB, and used manual write loops to ensure full transmission of both header and body.

For receiving the server response, I implemented a separate loop to read until \r\n\r\n was seen, then parsed the `Content-length` field and read exactly that many bytes. Again, the example assumed a single `read()` was enough, which often failed.

Several helper functions like `send_400_response()` were introduced to handle bad requests clearly. I also used `signal(SIGPIPE, SIG_IGN)` early in both programs to prevent crashes from broken pipes.

Many of these fixes -- including `strtoull()` parsing, header normalization, or multi-phase reads -- were developed through debugging, not referenced from elsewhere. Most were motivated by failed test runs, mismatched `Content-length`, or client disconnects mid-request.

In the end, while the example provided a starting point, nearly every assumption it made had to be revised.

This project became more about debugging, testing, and handling edge cases than about following a reference. That process taught me far more about real-world socket behavior than any static code ever could.