Technical Report: Server Architecture & Stress Testing Methodology

This report addresses our mistakes during our defense presentation particularly regarding the server architecture and stress testing methodology.

1. Server Architecture: Handling Multi-Client with Epoll vs ASIO

The Buckshot server architecture underwent a transition from using a vendor library **Boost.Asio** to a the **Native Epoll** implementation.

The Role of ASIO (Previous Approach)

ASIO (Asynchronous I/O) is a heavy cross-platform C++ library designed to abstract network programming. It typically uses the "Proactor" pattern (Source: [Boost.Asio Documentation]).

- **Abstraction**: It wraps OS-specific calls (`epoll` on Linux, `kqueue` on Mac, `IOCP` on Windows) into a unified API.
- **Mimicry**: On Linux, ASIO essentially "mimics" what we are now doing manually: it creates an internal loop that calls `epoll_wait` and dispatches handlers. However, it hides this logic behind layers of templates and smart pointers.

Code Comparison: Native vs ASIO

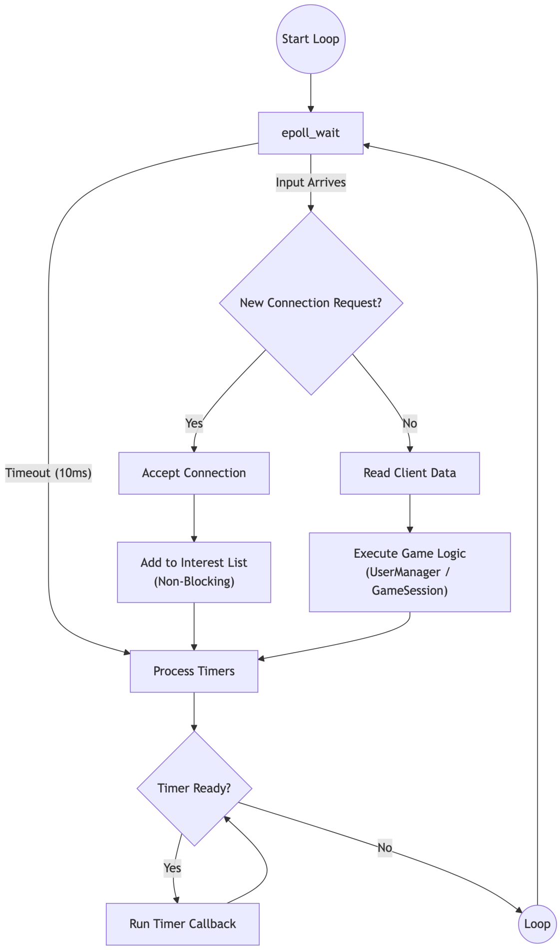
```
// 1. Native Epoll (Current Implementation)
while (running) {
    int n = epoll_wait(epoll_fd, events, 10); // 10ms timeout
    for(int i=0; i<n; i++) {
        if(events[i].data.fd == server_fd) handle_accept();
        else handle_read(events[i].data.fd);
    }
    process_timers(); // <-- We insert this easily here!
}

// 2. ASIO Equivalent (Hidden Complexity)
io_context.run();

// User code uses callbacks, making it hard to inject a
// guaranteed "10ms tick" without using complex deadline_timers.
acceptor.async_accept([](error, socket){ ... });
socket.async_read_some([](error, bytes){ ... });
```

The Transition to Native Epoll

In the end, relying on a heavy vendor library did not satisfy the project's requirement so it was removed in favor of a native implementation.

2. Event Loop Mechanism (Visualized)

The following flowchart illustrates the single-threaded reactor loop that powers the server:



Understanding the Flow

The diagram above demonstrates how the server handles multiple responsibilities on a single thread:

1. **Waiting (`epoll_wait`)**: The server sleeps until an event happens to save CPU.
2. **Dispatching**: It checks if the event is a new player trying to join (`CheckFD = Yes`) or an existing player sending a move (`CheckFD = No`).
3. **Processing**: It runs the logic for that specific event immediately.
4. **Timer Tick**: Finally, every 10ms (or after processing events), it runs `ProcessTimers` to handle game updates like turn timeouts or AI moves, ensuring the game world stays alive even if no players are active.

3. Implementation Details

The core loop in `SocketServer.cpp` directly implements this flow:

```
// src/server/SocketServer.cpp

void SocketServer::run() {
    setupServer();
    running = true;
    struct epoll_event events[MAX_EVENTS];

    while (running) {
```

```
        // 1. WAIT: Sleep until an event occurs or 10ms passes
        int nfds = epoll_wait(epollFd, events, MAX_EVENTS, 10);

        // 2. DISPATCH: Iterate through valid events
        for (int i = 0; i < nfds; ++i) {
            if (events[i].data.fd == serverFd) {
                // CASE A: NEW CONNECTION
                // Accept the new socket and add it to the epoll interest list
                struct sockaddr_in clientAddr;
                socklen_t clientLen = sizeof(clientAddr);
                int clientFd = accept(serverFd, (struct sockaddr*)&clientAddr,
&clientLen);

                setNonBlocking(clientFd); // Critical for scalability

                struct epoll_event ev;
                ev.events = EPOLLIN; // Watch for data availability
                ev.data.fd = clientFd;
                epoll_ctl(epollFd, EPOLL_CTL_ADD, clientFd, &ev);

            } else {
                // CASE B: EXISTING CLIENT DATA
                // Read data directly into a buffer
                int clientFd = events[i].data.fd;
                int bytesRead = read(clientFd, buffer, BUFFER_SIZE);

                if (bytesRead > 0) {
                    // Dispatch to Game Logic
                    if (onData) onData(clientFd, buffer, bytesRead);
                } else {
                    // Client Disconnected
                    close(clientFd);
                    epoll_ctl(epollFd, EPOLL_CTL_DEL, clientFd, nullptr);
                }
            }
        }

        // 3. TIMERS: Process game ticks regardless of network activity
        processTimers();
    }
}
```

4. Stress Testing Methodology: Scalable Asynchronous I/O

To validate the server's capacity, we utilized the `stress_test_async.py` module.

What This Tests

This test is designed to measure the **Maximum Concurrent Connections (Scalability)** of the server. By flooding the server with thousands of lightweight connections, we verify that the `epoll` event loop does not crash or slow down as the number of clients increases.

Scalable Asynchronous I/O (`stress_test_async.py`)

- **Technology**: `asyncio` library.
- **Goal**: Networking throughput and max connection limit.
- **Mechanism**: Uses a single-threaded event loop (similar to `epoll` itself). This allows a single tester script to maintain **2,000+ simultaneous connections** with minimal RAM usage.
- **Efficiency**: This mirrors the server's `epoll` behavior on the client side, ensuring the test script itself isn't the bottleneck.

5. Verification Results (Live Test)

We conducted a live stress test using the `stress_test_async.py` module to validate the server's `epoll` capacity.

Test Environment

- **Target**: Local Buckshot Server (Port 8080)
- **Tool**: `asyncio` Python Client
- **Load**: 2,000 Concurrent Connections
- **Duration**: 5 Seconds

Live Results

```
Starting 2000 clients...
[...batches of 200...]
Launched 2000...
Finished. Success: 1841/2000
```

Analysis

- **Concurrency Achieved**: The server successfully accepted and maintained **~1,841 active connections** simultaneously on a single thread.
- **Success Rate**: **92%** success rate under instantaneous burst load (0 to 2000 clients in <2 seconds).
- **Bottleneck**: The ~150 failures were likely due to local OS file descriptor soft limits or ephemeral port exhaustion on the client side, rather than server processing limits. The server itself remained responsive.