Event Loop with epoll: Design Document

Overview

This document designs a single-threaded event loop server capable of handling 10,000+ concurrent connections using Linux's epoll mechanism. It solves the C10K problem by implementing the Reactor pattern with non-blocking I/O, efficient timer management via a hierarchical timer wheel, and asynchronous task scheduling. The key architectural challenge is managing massive concurrency with low overhead and predictable latency without using threads.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** Milestone 1 (foundational understanding)

This section establishes why traditional server architectures fail at scale and introduces the fundamental concepts that enable handling 10,000+ concurrent connections with a single thread. We'll explore the limitations of thread-per-connection models, introduce the event-driven mental model through analogy, and compare the available I/O multiplexing technologies on Linux systems.

The C10K Problem and the Threading Bottleneck

Imagine a small post office with one clerk who can only serve one customer at a time. When a customer arrives, the clerk helps them completely—filling out forms, weighing packages, calculating postage—while other customers wait in line. This works fine when 5-10 customers arrive per hour, but completely collapses when 10,000 customers show up simultaneously. The clerk becomes overwhelmed, the line grows indefinitely, and most customers give up waiting. This is precisely what happens with **thread-per-connection** server architectures under high concurrency.

In traditional blocking I/O server designs, each incoming client connection spawns a new operating system thread (or process). This thread calls blocking system calls like `accept()`, `read()`, and `write()`, which cause the thread to enter a suspended state ("blocked") until data arrives or can be sent. While conceptually simple, this approach suffers from severe scalability limitations known as the **C10K problem** (handling ten thousand concurrent connections).

> **Decision: Reject Thread-Per-Connection Architecture**
>
> - **Context**: Need to handle 10,000+ concurrent connections with predictable latency and minimal resource overhead
> - **Options Considered**:
>   1. Thread-per-connection with blocking I/O
>   2. Thread pool with blocking I/O
>   3. Single-threaded event loop with non-blocking I/O
> - **Decision**: Single-threaded event loop with non-blocking I/O
> - **Rationale**: Thread-per-connection requires 10K threads, each with its own stack (typically 8MB), consuming ~80GB RAM just for stacks. Context switching between 10K threads introduces massive CPU overhead. Thread pools with blocking I/O still waste threads waiting on I/O, limiting concurrency to pool size
> - **Consequences**: Must implement non-blocking I/O and event multiplexing; programming model becomes callback-driven rather than linear; must handle partial reads/writes explicitly

Bottleneck	Thread-Per-Connection Impact at 10K Connections	Technical Explanation
Memory Overhead	80-160 GB RAM for thread stacks alone	Each thread requires 8-16MB stack (default Linux pthread). 10K × 8MB = 80GB
Context Switch Overhead	Millions of switches per second, >90% CPU wasted	OS scheduler must cycle through all runnable threads. Each switch flushes CPU caches, TLB, and registers
File Descriptor Limits	Default system limits (1024) must be raised	Each connection consumes a file descriptor. 10K connections need adjusted `ulimit -n` and kernel parameters
Synchronization Complexity	Lock contention increases super-linearly with threads	Shared data structures require locks. Probability of contention grows with thread count, causing serialization
Connection Setup Teardown	Thread creation/destruction becomes bottleneck	`pthread_create()` and `pthread_join()` are expensive system calls at 10K scale

The critical insight is that **threads are expensive when waiting**. A thread blocked in `read()` or `write()` is still consuming resources (memory, kernel structures) while doing no useful work. The fundamental solution is to separate the **waiting** from the **processing**—instead of having thousands of threads wait individually, have a single entity efficiently monitor thousands of connections and only wake up when there's actual work to do.

Event-Driven Architecture: The Postal System Analogy

Consider a modern postal sorting facility with a single, highly efficient sorting machine (the event loop) and a team of specialized workers (callback handlers). Letters (I/O events) arrive continuously on conveyor belts (file descriptors). Rather than assigning one worker per letter to wait at each mailbox, the system works like this:

1. **The Sorting Machine (epoll)** monitors hundreds of mailboxes simultaneously
2. When a letter arrives in any mailbox, the machine detects it instantly
3. The machine routes that specific letter to an available specialized worker
4. The worker processes only that letter (reads address, applies postage)
5. If the worker needs to wait for something (like a scale to become available), they don't block—they return the letter to the machine with a note saying "notify me when the scale is free"
6. The worker immediately becomes available for other letters

This is the essence of the **Reactor pattern**—a single event loop dispatches events to appropriate handlers without blocking. The key components in our analogy map to system components:

Postal System Component	Event Loop Equivalent	Responsibility
Sorting Machine	`epoll` instance	Monitors multiple file descriptors for events
Mailboxes	File descriptors (sockets)	Sources of I/O events (data arriving, space available)
Letter with Data	Read/Write buffer	Application data to process or send
Specialized Workers	Callback functions	Process specific types of events (HTTP, echo, etc.)
"Notify Me Later" Notes	Event registration	Request to be notified when a specific condition occurs

The mental model shift is from **"one thread, one connection, linear execution"** to **"one loop, many connections, event-driven execution."** Instead of writing code that says "read from this socket until we get a full HTTP request," we write: "when data arrives on this socket, parse what you can; if you don't have a full request yet, ask to be notified when more data arrives."

> **Key Insight**: The event loop's power comes from never waiting idly. It either processes ready events or sleeps efficiently in the kernel until events occur. This eliminates the resource waste of blocked threads while maintaining concurrency.

This architecture enables our single-threaded server to handle 10K+ connections because:

- **Constant memory overhead**: ~1KB per connection for buffers/state vs. 8MB per thread
- **Constant context switching**: Exactly one thread, so no user-space context switches
- **Efficient waiting**: The kernel's `epoll_wait()` monitors all connections in one system call
- **Predictable latency**: No lock contention or scheduler interference between connections

Comparison of I/O Multiplexing Approaches

Linux offers several system calls for monitoring multiple file descriptors. Choosing the right one is critical for performance at scale. All these approaches follow the same basic pattern: tell the kernel which file descriptors to monitor and what events to watch for, then wait for something to happen.

> **Decision: Use epoll for I/O Multiplexing**
>
> - **Context**: Need to efficiently monitor 10,000+ file descriptors for read/write readiness on Linux
> - **Options Considered**:
>   1. `select()` - POSIX standard, available everywhere
>   2. `poll()` - Improved over select, but similar scalability issues
>   3. `epoll()` - Linux-specific, designed for massive scalability
>   4. `kqueue()` - BSD/macOS equivalent of epoll (not considered for Linux)
> - **Decision**: `epoll` with edge-triggered mode (EPOLLET)
> - **Rationale**: `select` / `poll` scan all file descriptors on every call (O(n)). With 10K FDs, this wastes CPU. `epoll` maintains kernel state and only returns changed FDs (O(1) for ready FDs). Edge-triggered mode reduces unnecessary wakeups
> - **Consequences**: Linux-only; must handle EPOLLET semantics (read/write until EAGAIN); slightly more complex setup than select/poll

The following table compares the technical characteristics of each approach:

Characteristic	`select()`	`poll()`	`epoll()` (Level-Triggered)	`epoll()` (Edge-Triggered)
Maximum FDs	FD_SETSIZE (1024)	Limited by `ulimit -n`	Limited by `ulimit -n`	Limited by `ulimit -n`
Time Complexity	O(n) scan all FDs each call	O(n) scan all FDs each call	O(1) for ready FDs	O(1) for ready FDs
Memory Usage	Fixed-size bitmask	Array of struct pollfd	Kernel hash table + user array	Kernel hash table + user array
Trigger Mode	Level-only (reports if ready)	Level-only (reports if ready)	Level (default)	Edge (EPOLLET flag)
FD Management	Rebuild set each call	Rebuild array each call	Kernel-managed set (add/mod/del)	Kernel-managed set (add/mod/del)
Portability	POSIX, everywhere	POSIX, most systems	Linux only	Linux only
Best For	< 1000 FDs, portability	< 1000 FDs, >1024 FDs needed	1000+ FDs, Linux only	1000+ FDs, Linux, max performance

Understanding Level vs. Edge Triggering:

- **Level-Triggered (default)**: "Notify me while the condition is true." If data is available to read, you'll keep getting notified every event loop iteration until you read all data
- **Edge-Triggered (EPOLLET)**: "Notify me when the condition changes." You get notified once when data arrives, then not again until you read and get EAGAIN

To illustrate the performance difference, consider checking 10,000 mailboxes:

- With `select` / `poll`: Walk past each of 10,000 mailboxes every time, checking if there's mail
- With `epoll`: Get a list saying "mailboxes 42, 317, and 5823 have new mail"

Trigger Mode Implications:

Trigger Mode	Pros	Cons	Required Handling Pattern
Level-Triggered	Simpler: won't miss events if you don't read all data immediately	More wakeups: get notified repeatedly while data remains	Can read partial data each time; loop continues while readable
Edge-Triggered	Fewer wakeups: only notified on state changes	Complex: must read until EAGAIN to reset trigger	Must read/write in loop until EAGAIN; risk starvation if other FDs have data

> **Architecture Principle**: For our 10K+ connections target, we choose edge-triggered epoll. While more complex to implement correctly, it minimizes unnecessary event loop iterations when connections have continuous data flow. The key is ensuring we always read/write until EAGAIN when notified.

Why Not Just Use Threads with epoll? Multi-threaded epoll designs (one epoll per thread) can scale beyond single-core limits, but they introduce synchronization complexity that's beyond our learning scope. Our single-threaded design establishes the foundational pattern—the reactor core—which can later be extended to multiple threads (one event loop per CPU core) in a production system.

Common Pitfalls in Conceptual Understanding:

⚠️ **Pitfall: Believing "Non-Blocking" Means "Zero Wait"**

- **Description**: Assuming non-blocking I/O eliminates all waiting time
- **Why It's Wrong**: Non-blocking calls return immediately regardless of data availability, but the event loop still sleeps in `epoll_wait()` when nothing is ready. The difference is fine-grained waiting (per FD) vs. coarse waiting (per thread)
- **Fix**: Understand that `epoll_wait()` is still a blocking call (with timeout), but it blocks on ALL monitored FDs simultaneously, not individually

⚠️ **Pitfall: Confusing Concurrency with Parallelism**

- **Description**: Expecting 10K connections to be processed simultaneously
- **Why It's Wrong**: Single-threaded event loops process one event at a time. Concurrency means making progress on many tasks interleaved, not simultaneously. While processing one connection's data, others wait in the queue
- **Fix**: Design handlers to be short and non-blocking. Defer expensive work to timer callbacks or task queues to avoid starving other connections

⚠️ **Pitfall: Overlooking the Cost of System Calls**

- **Description**: Thinking epoll eliminates all overhead
- **Why It's Wrong**: Each `epoll_ctl()` (add/modify/delete FD) is a system call with context switch cost. With 10K connections churning rapidly, this becomes significant
- **Fix**: Batch FD modifications where possible; reuse connections (HTTP keep-alive); design protocols to minimize connection churn

The transition from thread-per-connection to event-driven architecture represents a fundamental shift in how we think about server programming. Instead of "who does this work?" (which thread), we focus on "when does this work happen?" (in response to which event). This event-driven mindset, combined with Linux's `epoll` mechanism, enables our single-threaded server to handle massive concurrency within practical resource limits.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
I/O Multiplexing	`select()` with FD_SETSIZE=1024	`epoll()` with edge-triggered mode
Non-blocking Setup	`fcntl(fd, F_SETFL, O_NONBLOCK)`	`socket()` with SOCK_NONBLOCK flag (Linux 2.6.27+)
Buffer Management	Fixed-size per-connection buffers	Dynamic ring buffers with watermark thresholds
Timer Integration	Simple sorted list of timeouts	Hierarchical timer wheel (Milestone 2)

B. Recommended File/Module Structure

For the initial epoll basics implementation (Milestone 1):

```
event-loop-project/
├── include/
│   ├── event_loop.h      # Core event loop structures and API
│   └── connection.h      # Connection state management
├── src/
│   ├── event_loop.c      # epoll implementation, event dispatch
│   ├── connection.c      # Connection setup, teardown, buffer management
│   ├── server.c          # Listening socket setup, accept loop
│   └── main.c           # Entry point, signal handling, main loop
├── examples/
│   └── echo_server.c    # Example echo server using the event loop
└── tests/
    ├── test_event_loop.c
    └── test_concurrent.c
```

C. Infrastructure Starter Code

Complete, working code for non-blocking socket setup:

C

```c
/* include/nonblocking.h */
#ifndef NONBLOCKING_H
#define NONBLOCKING_H

#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>

/* Set a file descriptor to non-blocking mode */
static inline int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        return -1;
    }
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}

/* Safe read wrapper that handles EAGAIN/EWOULDBLOCK */
static inline ssize_t safe_read(int fd, void *buf, size_t count) {
    ssize_t n;
    do {
        n = read(fd, buf, count);
    } while (n == -1 && errno == EINTR);  // Retry if interrupted by signal
    return n;
}

/* Safe write wrapper that handles EAGAIN/EWOULDBLOCK */
static inline ssize_t safe_write(int fd, const void *buf, size_t count) {
    ssize_t n;
    do {
        n = write(fd, buf, count);
    } while (n == -1 && errno == EINTR);  // Retry if interrupted by signal
    return n;
}
```

```c
/* Check if errno indicates "would block" */
static inline int is_would_block(void) {
    return errno == EAGAIN || errno == EWOULDBLOCK;
}

#endif /* NONBLOCKING_H */
```

D. Core Logic Skeleton Code

Event loop initialization with epoll:

```c
/* src/event_loop.c - Core event loop implementation */

#include "event_loop.h"
#include <sys/epoll.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>

struct event_loop {
    int epoll_fd;                   // epoll file descriptor
    struct epoll_event *events;     // Array for epoll_wait results
    int max_events;                 // Size of events array
    int running;                    // Loop control flag
};

/* Create a new event loop instance */
struct event_loop *event_loop_create(int max_events) {
    // TODO 1: Allocate memory for event_loop struct
    // TODO 2: Create epoll instance using epoll_create1(EPOLL_CLOEXEC)
    // TODO 3: Check for epoll creation failure (return NULL if failed)
    // TODO 4: Allocate array for epoll_event structures
    // TODO 5: Initialize max_events and running flag (set to 1)
    // TODO 6: Return event_loop pointer or NULL on any failure
}

/* Register a file descriptor with the event loop */
int event_loop_register_fd(struct event_loop *loop, int fd,
                           uint32_t events, void *user_data) {
    // TODO 1: Create epoll_event struct and populate fields
    // TODO 2: Set events field (EPOLLIN, EPOLLOUT, EPOLLET, etc.)
    // TODO 3: Set user data pointer in epoll_event.data.ptr
    // TODO 4: Call epoll_ctl with EPOLL_CTL_ADD
    // TODO 5: Handle errors (EEXIST if already registered)
    // TODO 6: Return 0 on success, -1 on error
```

```c
}

/* Modify events for an already registered file descriptor */
int event_loop_modify_fd(struct event_loop *loop, int fd,
                         uint32_t events, void *user_data) {
    // TODO 1: Create epoll_event struct and populate fields
    // TODO 2: Call epoll_ctl with EPOLL_CTL_MOD
    // TODO 3: Handle errors (ENOENT if not registered)
    // TODO 4: Return 0 on success, -1 on error
}

/* Unregister a file descriptor from the event loop */
int event_loop_unregister_fd(struct event_loop *loop, int fd) {
    // TODO 1: Call epoll_ctl with EPOLL_CTL_DEL (ignore errors)
    // TODO 2: Note: epoll_ctl with EPOLL_CTL_DEL doesn't need epoll_event
    // TODO 3: Return 0 always (can't fail meaningfully)
}

/* Run the event loop (main dispatch function) */
int event_loop_run(struct event_loop *loop, int timeout_ms) {
    // TODO 1: Check that loop is not NULL and running flag is set
    // TODO 2: Call epoll_wait with timeout, store number of ready events
    // TODO 3: For each ready event (0 to n-1):
    // TODO 4:   Extract user data pointer from event.data.ptr
    // TODO 5:   Check event.events for EPOLLIN, EPOLLOUT, EPOLLERR, EPOLLHUP
    // TODO 6:   Dispatch to appropriate handler (callback function)
    // TODO 7: Handle epoll_wait errors (EINTR for signal interrupts)
    // TODO 8: Return number of events processed or -1 on fatal error
}

/* Stop the event loop */
void event_loop_stop(struct event_loop *loop) {
    // TODO 1: Set running flag to 0
    // TODO 2: Optional: wake up epoll_wait with an eventfd or pipe
}
```

E. Language-Specific Hints (C)

- **Error Handling**: Always check `errno` after system call failures. `EINTR` means interrupted by signal and usually should be retried
- **Memory Management**: Use `calloc()` instead of `malloc()` for structures to ensure zero initialization of padding bytes
- **Epoll Flags**: `EPOLL_CLOEXEC` ensures the epoll FD is closed on `exec()` calls. Always use it with `epoll_create1()`
- **Event Data**: Store connection context in `epoll_event.data.ptr`, not `data.fd`, to pass rich context to handlers
- **File Descriptor Limits**: Call `getrlimit(RLIMIT_NOFILE, ...)` and `setrlimit()` early to ensure enough FDs for 10K connections

F. Milestone Checkpoint (Milestone 1)

After implementing the basic epoll event loop:

1. **Compile and Run**:

```bash
gcc -o echo_server src/*.c examples/echo_server.c -I./include

./echo_server 8080
```

2. **Expected Behavior**:
   - Server starts on port 8080
   - No CPU usage when idle (event loop sleeps in `epoll_wait()`)
   - Multiple `telnet` connections can connect simultaneously
   - Each connection echoes back typed characters
   - Connections can be closed cleanly
3. **Verification Commands**:

```bash
# In one terminal - start server
./echo_server 8080



# In another terminal - test with telnet
telnet localhost 8080
Type "hello" → should echo "hello"



# Test multiple connections
for i in {1..5}; do
  (echo "test$i"; sleep 1) | telnet localhost 8080 &
done
```

4. **Signs of Problems**:
   - **CPU at 100%**: Likely not handling EAGAIN, causing busy loop
   - **Can't accept connections**: Forgot to set listening socket non-blocking
   - **Data loss**: Not reading until EAGAIN in edge-triggered mode
   - **Memory leak**: Not freeing connection contexts on close

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server accepts only one connection	Listening socket not set to non-blocking	`accept()` blocks entire loop	Call `set_nonblocking()` on listening socket
High CPU when idle	Not sleeping in `epoll_wait()`	Check if timeout is 0 or loop has no `epoll_wait` call	Ensure proper timeout to `epoll_wait()`
Connections timeout or hang	Edge-triggered mode not reading to completion	Add logging showing EAGAIN after reads	Read in loop until EAGAIN when EPOLLIN received
`epoll_ctl` fails with EEXIST	Double registration of same FD	Check registration logic	Use `EPOLL_CTL_MOD` for existing FDs
Memory grows unbounded	Connection contexts not freed	Use `valgrind --leak-check=full`	Ensure `close()` and free on EPOLLHUP

Goals and Non-Goals

> **Milestone(s):** All milestones

This section defines the **core capabilities** (Goals) and **explicit boundaries** (Non-Goals) of the Event Loop with epoll system. Establishing clear boundaries is critical for a learning-focused project—it prevents scope creep while ensuring the architecture remains focused on solving the fundamental C10K problem using the **Reactor pattern**. The goals directly map to the four project milestones, creating a cohesive learning path from basic event-driven I/O to a complete, scalable HTTP server.

Goals

The system must deliver four interconnected capabilities that collectively demonstrate mastery of high-concurrency server design. Each goal addresses a specific architectural challenge identified in the C10K problem.

Goal	Architectural Challenge Addressed	Key Capabilities	Primary Milestone
Handle 10K+ concurrent connections	Resource exhaustion from thread-per-connection models	Single-threaded **event loop** with `epoll`, non-blocking I/O, O(1) event dispatch	Milestone 1, 4
Support idle timeouts	Connection resource leakage from stale or slow clients	Hierarchical **timer wheel** with O(1) insertion/deletion, integrated timeout propagation	Milestone 2
Provide a callback API	Complexity of raw event loop programming	**Reactor pattern** abstraction with callback registration for I/O and timer events	Milestone 3
Include an HTTP server example	Practical application of event-driven architecture	Incremental HTTP/1.1 parser, write buffer management, keep-alive support	Milestone 4

Goal 1: Handle 10K+ Concurrent Connections

The primary objective is to demonstrate that a single-threaded, event-driven server can maintain **10,000+ simultaneous TCP connections** with predictable latency and minimal resource overhead. This directly tackles the C10K problem's core constraint: traditional operating systems cannot efficiently schedule 10,000 threads. The system achieves this through:

- **Linux `epoll` for I/O multiplexing**: Unlike `select()` or `poll()`, `epoll` uses O(1) event notification regardless of connection count, avoiding the linear scan overhead that breaks at scale.
- **Non-blocking I/O on all sockets**: Every file descriptor—including the listening socket—must be set to non-blocking mode using `fcntl(O_NONBLOCK)`. This ensures the single thread never blocks on `accept()`, `read()`, or `write()` operations.

- **Level-triggered event notification**: While edge-triggered (`EPOLLET`) can be more efficient, level-triggered mode simplifies correctness for learners by ensuring events re-fire if data remains unprocessed. The architecture must handle both models correctly.
- **Efficient memory per connection**: Each connection should maintain only necessary state (socket descriptor, read/write buffers, timer reference) in a compact `struct connection` to minimize memory footprint.

> **Architecture Decision: Single-threaded event loop over multi-threaded reactor**
>
> - **Context**: The C10K problem demonstrates that thread-per-connection models exhaust memory and CPU due to stack overhead and context switching. We need maximum connection density with minimal overhead.
> - **Options Considered**:
>   1. **Single-threaded event loop**: One thread handles all I/O, timers, and callbacks.
>   2. **Multi-threaded reactor**: One acceptor thread dispatches connections to worker threads, each with its own `epoll` instance.
>   3. **Thread pool with work stealing**: Fixed thread pool processes events from a shared queue.
> - **Decision**: Single-threaded event loop.
> - **Rationale**: For the learning objective—understanding the Reactor pattern fundamentals—a single-threaded model eliminates concurrency complexity (locking, race conditions) while demonstrating the raw efficiency possible with `epoll`. It achieves the highest connection density per CPU core, which is the essence of solving C10K. Multi-threaded variants add complexity without changing the fundamental event-driven paradigm.
> - **Consequences**: The server cannot utilize multiple CPU cores. All processing—including HTTP parsing—must be non-blocking and efficient to avoid stalling the event loop. This constraint reinforces careful design of incremental processing.

Goal 2: Support Idle Timeouts

Long-lived connections must be automatically cleaned up after periods of inactivity to prevent resource exhaustion from abandoned or idle clients. The system requires a timeout mechanism that:

- **Closes connections after configurable idle periods** (e.g., 30 seconds without read/write activity).
- **Integrates seamlessly with the event loop** without requiring separate threads or periodic signals.
- **Provides O(1) timer operations** to avoid degrading performance as connection count grows.
- **Allows timer cancellation** when connections become active again (resetting the idle timeout).

The **hierarchical timer wheel** data structure meets these requirements by organizing timers into multiple levels of increasing granularity (e.g., milliseconds, seconds, minutes), enabling constant-time insertion and expiration processing.

Goal 3: Provide a Callback API

Raw `epoll` programming requires manual management of file descriptor registration and event masks. The system must abstract these details behind a clean **Reactor pattern** API that:

- **Allows registration of callback functions** for `EPOLLIN` (read-ready) and `EPOLLOUT` (write-ready) events.
- **Supports timer callbacks** for one-shot and repeating operations.
- **Manages a deferred task queue** for work that should execute after I/O processing in the current event loop iteration.
- **Hides `epoll` system call details** behind intuitive functions like `event_loop_register_fd()` and `event_loop_run()`.

This abstraction transforms the event loop from a fragile, low-level construct into a reusable framework for building various protocol servers.

Goal 4: Include an HTTP Server Example

The theoretical architecture must be validated with a practical, benchmarkable implementation. An HTTP/1.1 server serves as the canonical demonstration because:

- **HTTP is a well-understood protocol** with clear request/response semantics.

- **It exercises all event loop components**: non-blocking reads/writes, incremental parsing, buffer management, timer-based keep-alive, and callback dispatch.
- **It provides measurable performance metrics** for the 10K goal via tools like `ab` (Apache Bench) or `wrk`.

The HTTP implementation must handle real-world complexities: requests split across multiple TCP packets, slow clients that can't consume responses quickly, and persistent connections with proper idle timeout management.

Non-Goals

Explicitly defining what the system does **not** do is equally important for maintaining focus and setting learner expectations. These boundaries acknowledge practical production considerations while keeping the project achievable within its educational scope.

Non-Goal	Reason for Exclusion	Alternative/Consideration
Multi-threading or multi-process architecture	Adds significant complexity (locking, synchronization) that distracts from core event-driven principles. The goal is to master single-threaded concurrency first.	For production, a multi-threaded reactor (one `epoll` instance per CPU core) is the logical next step after understanding this foundation.
Full HTTP/2 protocol support	HTTP/2 introduces binary framing, multiplexing, and flow control that require substantial additional implementation.	The event loop framework could support HTTP/2, but implementing the full protocol exceeds the scope of learning epoll fundamentals.
SSL/TLS encryption	TLS adds complex handshaking, renegotiation, and record layer processing that obscures the core I/O multiplexing concepts.	In production, TLS termination is often handled by a dedicated proxy (nginx) or library (OpenSSL with async mode).
Production-grade connection pooling	Connection pooling involves health checks, load balancing, and failover mechanisms that are orthogonal to the event loop itself.	The architecture focuses on handling connections, not managing pools between services.
Advanced load balancing or service discovery	These are distributed systems concerns that build upon, rather than comprise, the event loop pattern.	The event loop could be used within each node of a load-balanced cluster.
Windows compatibility or cross-platform I/O multiplexing	The project specifically targets Linux's `epoll` as the learning vehicle.	Production code might use abstraction libraries (libuv) but understanding the underlying OS mechanism is the educational goal.
Comprehensive security hardening	While basic error handling is required, full security audit (buffer overflow prevention, input validation) exceeds the scope.	Learners should apply security best practices but the focus remains on architecture and performance.
Database integration or business logic	The server is a platform for connection management, not a full application framework.	The callback API could dispatch to business logic, but implementing such logic is beyond the C10K focus.

Why These Boundaries Matter

Each non-goal represents a legitimate production concern that has been intentionally deferred:

> **Design Principle: Focus on the Pattern, Not the Production Features**
>
> The event loop is a **foundational pattern** that enables high-concurrency servers. By excluding multi-threading, TLS, and HTTP/2, we isolate and master this pattern in its purest form. Once understood, these extensions become incremental additions rather than overwhelming complexities. For example:
>
> - **Multi-threading** can be added by creating multiple event loop threads (one per CPU core) with load-balanced accept.
> - **TLS** can be integrated using OpenSSL's asynchronous mode with `SSL_read()` and `SSL_write()` returning `SSL_ERROR_WANT_READ/WRITE`.
> - **HTTP/2** could be implemented atop the same event loop by replacing the HTTP/1.1 parser with a state machine for binary frames.
>
> The architecture is deliberately **extensible at its boundaries**—the callback API, timer system, and buffer management all provide hooks for these future enhancements without redesign.

Common Pitfalls in Scope Interpretation

Learners often misunderstand these boundaries, leading to two counterproductive extremes:

⚠️ Pitfall: Over-engineering toward production readiness

- **Mistake**: Attempting to implement multi-threading, TLS, or connection pooling before mastering the single-threaded event loop.
- **Why it's wrong**: These features introduce complexity that obscures the core learning objectives. Debugging race conditions in a multi-threaded event loop is vastly harder than understanding a single-threaded one.
- **How to avoid**: Strictly follow the milestone sequence. Complete all four milestones with a single-threaded, plain TCP/HTTP server before considering extensions.

⚠️ Pitfall: Underestimating the HTTP server complexity

- **Mistake**: Treating the HTTP server as a trivial addition after the event loop works for echo servers.
- **Why it's wrong**: Real HTTP requires incremental parsing, header validation, chunked encoding, and proper response buffering—all of which exercise subtle aspects of non-blocking I/O and state management.
- **How to avoid**: Allocate significant time for Milestone 4. The HTTP implementation should be treated as a first-class component, not an afterthought.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Extension)
I/O Multiplexing	`epoll` with level-triggered mode	`epoll` with edge-triggered mode + `EPOLLONESHOT`
Timer Management	Hierarchical timer wheel (4-level)	Min-heap for exact timers + timerfd for kernel integration
HTTP Parsing	Handwritten incremental state machine	`http-parser` library (node.js parser in C)
Buffering	Per-connection read/write ring buffers	Scatter/gather I/O with `readv()` / `writev()`
Concurrency Model	Single-threaded reactor	Multi-threaded acceptor + workers (thread pool)

Recommended File/Module Structure:

```
event-loop-project/
├── include/
│   ├── event_loop.h      # Core event loop types and API
│   ├── timer_wheel.h     # Timer wheel data structure
│   ├── connection.h      # Connection state management
│   └── http_server.h     # HTTP protocol handler
├── src/
│   ├── main.c            # Server entry point
│   ├── event_loop.c      # epoll event loop implementation (Milestone 1, 3)
│   ├── timer_wheel.c     # Hierarchical timer wheel (Milestone 2)
│   ├── connection.c      # Connection lifecycle and buffers
│   ├── http_parser.c     # Incremental HTTP/1.1 parser (Milestone 4)
│   ├── http_response.c   # HTTP response generation
│   └── utils.c           # Non-blocking socket helpers
├── tests/
│   ├── test_timer_wheel.c
│   ├── test_http_parser.c
│   └── integration_test.c
└── benchmarks/
    └── c10k_benchmark.c  # 10K connection stress test
```

Infrastructure Starter Code:

Complete helper functions for non-blocking socket operations:

```c
// utils.c - COMPLETE implementation

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#include "utils.h"


int set_nonblocking(int fd) {

    int flags = fcntl(fd, F_GETFL, 0);

    if (flags == -1) {

        return -1;

    }

    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {

        return -1;

    }

    return 0;

}


int create_listening_socket(int port, int backlog) {

    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    if (listen_fd == -1) {

        return -1;

    }



    // Allow port reuse

    int reuse = 1;

    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));



    struct sockaddr_in addr = {0};

    addr.sin_family = AF_INET;

    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    addr.sin_port = htons(port);



    if (bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {

        close(listen_fd);
```

```
        return -1;
    }

    if (listen(listen_fd, backlog) == -1) {
        close(listen_fd);
        return -1;
    }

    if (set_nonblocking(listen_fd) == -1) {
        close(listen_fd);
        return -1;
    }

    return listen_fd;
}
```

Core Logic Skeleton Code:

Event loop creation function with TODOs mapping to Milestone 1:

```c
// event_loop.c - LEARNER IMPLEMENTS

#include <sys/epoll.h>

#include <stdlib.h>

#include "event_loop.h"


struct event_loop* event_loop_create(int max_events) {

    // TODO 1: Allocate memory for event_loop struct

    // TODO 2: Initialize fields: epoll_fd = epoll_create1(EPOLL_CLOEXEC)

    // TODO 3: Allocate events array of size max_events * sizeof(struct epoll_event)

    // TODO 4: Set max_events field and running = 1

    // TODO 5: Check for errors (epoll_create1 fails, malloc fails)

    // TODO 6: Return pointer to event_loop or NULL on error

}


int event_loop_register_fd(struct event_loop* loop, int fd,

                           uint32_t events, void* user_data) {

    // TODO 1: Create epoll_event struct and set .events = events

    // TODO 2: Set .data.ptr = user_data (or .data.fd = fd for simpler approach)

    // TODO 3: Call epoll_ctl(loop->epoll_fd, EPOLL_CTL_ADD, fd, &event)

    // TODO 4: Handle EEXIST error (fd already registered) with EPOLL_CTL_MOD

    // TODO 5: Return 0 on success, -1 on error

}


int event_loop_run(struct event_loop* loop, int timeout_ms) {

    // TODO 1: While loop->running is true

    // TODO 2: Call epoll_wait(loop->epoll_fd, loop->events, loop->max_events, timeout_ms)

    // TODO 3: Iterate through returned events (0 to n-1)

    // TODO 4: For each event, extract user_data pointer from event.data.ptr

    // TODO 5: Check event.events for EPOLLIN/EPOLLOUT/EPOLLERR/EPOLLHUP

    // TODO 6: Dispatch to appropriate callback function (to be implemented in Milestone 3)

    // TODO 7: Handle timeout_ms = -1 (wait forever) vs 0 (non-blocking poll)

    // TODO 8: Return number of events processed or -1 on error

}
```

Language-Specific Hints (C):

- Use `epoll_create1(EPOLL_CLOEXEC)` instead of `epoll_create()` to automatically close the epoll FD on `exec()`.

- Store `void* user_data` in `epoll_event.data.ptr` to pass connection context to callbacks.
- Always check `errno` after system calls fail: `EAGAIN` / `EWOULDBLOCK` for non-blocking I/O, `EINTR` for interrupted system calls.
- Compile with `-D_GNU_SOURCE` to get `EPOLL_CLOEXEC` and other Linux-specific extensions.
- Use `accept4()` with `SOCK_NONBLOCK` to accept and set non-blocking in one system call (Linux 2.6.28+).

Milestone Checkpoint for Goals Verification: After implementing each milestone, verify the corresponding goal:

1. **Milestone 1 - 10K Connections**: Run `./server` and use a load testing tool to establish 10,000 concurrent connections:

```bash
# In one terminal
./server --port 8080

# In another, using Python
python3 -c "import socket; [socket.socket().connect(('localhost', 8080)) for _ in range(10000)]"
```

Expected: Server should accept all connections without crashing, memory usage should be ~20-30MB (not gigabytes), and CPU should be idle after connections are established.

2. **Milestone 2 - Idle Timeouts**: Connect with `telnet localhost 8080`, don't send any data for 30 seconds:

```bash
telnet localhost 8080
# (wait 30 seconds without typing)
```

Expected: Connection should automatically close after the idle timeout. Check server logs for timeout messages.

3. **Milestone 3 - Callback API**: Write a test program that registers read/write callbacks:

```c
void on_read(int fd, void* data) { printf("FD %d is readable\n", fd); }
event_loop_register_callback(loop, fd, EPOLLIN, on_read, NULL);
```

Expected: Callback fires when data arrives on the socket. No raw `epoll` handling visible in user code.

4. **Milestone 4 - HTTP Server**: Benchmark with Apache Bench:

```bash
ab -n 10000 -c 100 http://localhost:8080/test
```

Expected: All requests complete successfully with latency < 100ms for 95th percentile. Server maintains 10K concurrent HTTP keep-alive connections.

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
CPU at 100% in event loop	`epoll_wait` returning immediately with no events	Check if timeout_ms = 0 (non-blocking mode) or events being constantly re-triggered	Use proper timeout, ensure level-triggered events are handled completely
Connections stuck at 1024	Hit default `epoll` instance limit	Check `cat /proc/sys/fs/epoll/max_user_watches`	Increase limit: `sysctl -w fs.epoll.max_user_watches=100000`
Timer not firing	Timer wheel not being advanced	Log timer wheel tick count; check if `epoll_wait` timeout is too large	Ensure `timeout_ms` in `event_loop_run` is min(next_timer_expiration, default)
Memory leak with 10K connections	Connection structures not freed on close	Use `valgrind --leak-check=full ./server`	Ensure `close(fd)` and free connection struct in all code paths (timeout, client close, error)
HTTP requests truncated	Read buffer too small or not handling partial reads	Log bytes read per `read()` call; check for `EAGAIN`	Increase buffer size, keep reading until `EAGAIN` in read callback

---

High-Level Architecture

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section provides the **bird's-eye view** of the event loop system's architecture—how all components fit together, their responsibilities, and the flow of data between them. Think of this as the **blueprint** for the entire project. Before diving into each component's internal details, we need to understand the overall structure and the **division of responsibilities** that enables handling 10,000+ connections in a single thread.

Component Diagram and Data Flow

Imagine the event loop system as a **high-efficiency post office** running with a single postal worker (the main thread). The postal worker doesn't wait at each mailbox for letters to arrive—instead, they monitor a centralized **notification board** (`epoll`) that lights up when any mailbox has incoming mail (read events) or is ready to accept outgoing mail (write events). The worker also wears a **multi-alarm wristwatch** (timer wheel) that buzzes at precise intervals to remind them to clean up abandoned mailboxes (idle timeouts). This mental model captures the essence of the Reactor pattern: a single thread efficiently dispatches work triggered by external events.

The system comprises four main components that work together:

Component	Primary Responsibility	Key Data Structures	Analogy
Event Loop (Reactor Core)	Monitors file descriptors for I/O readiness using `epoll`, dispatches events to registered callbacks	`event_loop`, `epoll_event`	Postal notification board
Timer Wheel	Manages timeout events with O(1) efficiency, integrated with the event loop's wait timeout	Hierarchical wheel slots, `timer` entries	Multi-alarm wristwatch
Async Task Queue	Schedules deferred callbacks for execution after I/O processing, enables clean API	Task queue, callback registry	Worker's todo list
Protocol Handlers	Implement application logic (e.g., HTTP) using non-blocking I/O and incremental parsing	`connection`, protocol-specific state	Mail sorting and processing rules

These components interact through a well-defined data flow, illustrated in the system overview:

Data Flow Through the System:

1. **Incoming Network Traffic** → The operating system's network stack receives packets and buffers data for sockets. When data arrives or a socket becomes writable, the kernel updates its internal readiness state.
2. **Event Detection (`epoll`)** → The `event_loop` calls `epoll_wait()` to query which file descriptors have pending I/O operations. The kernel returns an array of `epoll_event` structures, each containing the file descriptor and event type (`EPOLLIN`, `EPOLLOUT`, etc.).
3. **Event Dispatch** → For each ready file descriptor, the event loop looks up the associated `connection` context (via the `user_data` pointer in the `epoll_event`), then invokes the appropriate callback function registered for that event type.
4. **Protocol Processing** → The callback (typically part of a protocol handler like HTTP) performs non-blocking I/O operations (`read()`, `write()`). If an operation would block (`EAGAIN` / `EWOULDBLOCK`), it registers for the corresponding event and continues later. This enables **incremental processing**—handling partial data as it arrives.
5. **Timer Integration** → Before each `epoll_wait()` call, the event loop queries the timer wheel for the next expiration time and uses it as the timeout parameter. When `epoll_wait()` returns, the event loop "ticks" the timer wheel, executing any expired timer callbacks (e.g., closing idle connections).
6. **Deferred Task Execution** → After processing all I/O events and timers, the event loop executes any tasks in the async task queue. This ensures that callback registrations/modifications don't interfere with the current event iteration.

Decision: Component Separation and Single-Threaded Reactor

- **Context**: We need to handle 10K+ connections with minimal overhead while keeping the codebase maintainable and extensible. The system must support multiple protocols (echo, HTTP) and efficient timeout management.
- **Options Considered**:
  1. **Monolithic event loop**: All logic (I/O, timers, protocol) in one massive function.
  2. **Thread-per-component**: Separate threads for I/O multiplexing, timer management, and protocol processing.
  3. **Single-threaded Reactor with separated concerns**: One thread with cleanly separated components communicating through callbacks and shared state.
- **Decision**: Single-threaded Reactor with separated concerns.
- **Rationale**:
  - Monolithic code becomes unmaintainable and difficult to test.
  - Thread-per-component introduces synchronization overhead and complexity for 10K connections, negating the efficiency gains of event-driven architecture.
  - Separated concerns in a single thread maintain the performance benefits of no locking while enabling modular testing and clear responsibility boundaries.
- **Consequences**:
  - Enables clean API boundaries and unit testing of individual components.
  - Requires careful design to avoid blocking operations in any component (would stall entire system).
  - Callbacks must be non-blocking and execute quickly to maintain responsiveness.

Option	Pros	Cons	Why Not Chosen
Monolithic event loop	Simple initial implementation	Hard to extend, test, or debug; violates SRP	Becomes technical debt at scale
Thread-per-component	Parallel processing, natural isolation	Synchronization overhead, context switching costs, complex error handling	Defeats purpose of efficient single-threaded C10K solution
Single-threaded Reactor with separation	Clean architecture, testable, maintains performance	Requires disciplined non-blocking code	**CHOSEN**: Best balance of performance and maintainability

The **key architectural insight** is that all components share the same execution thread but have distinct responsibilities. The event loop owns the `epoll` file descriptor and dispatch logic. The timer wheel owns timeout tracking. Protocol handlers own connection state and parsing logic. They communicate through function pointers (callbacks) and shared context structures, avoiding the need for locks.

Recommended File/Module Structure

Organizing code for clarity and maintainability is crucial for a learning project. The recommended structure follows the **component separation** principle, with each major component living in its own pair of header and implementation files. This modular approach allows you to work on one milestone at a time while maintaining a clean architecture.

Project Directory Layout:

```
event-loop-epoll-project/
├── include/                    # Public header files
│   ├── event_loop.h           # Event loop API and types
│   ├── timer_wheel.h          # Timer wheel API and types
│   ├── connection.h           # Connection context structure
│   └── http_server.h          # HTTP server API
├── src/                       # Implementation source files
│   ├── core/
│   │   ├── event_loop.c       # Event loop implementation (Milestone 1)
│   │   ├── timer_wheel.c      # Timer wheel implementation (Milestone 2)
│   │   └── connection.c       # Connection buffer management
│   ├── networking/
│   │   ├── socket_utils.c     # Non-blocking socket helpers
│   │   └── listening_socket.c # Listening socket setup
│   ├── protocols/
│   │   ├── echo_handler.c     # Simple echo protocol (Milestone 1)
│   │   └── http_handler.c     # HTTP/1.1 server (Milestone 4)
│   └── utils/
│       ├── task_queue.c       # Async task queue (Milestone 3)
│       └── logging.c          # Debug logging utilities
├── tests/                     # Unit and integration tests
│   ├── test_timer_wheel.c
│   ├── test_event_loop.c
│   └── test_http_parser.c
├── examples/                  # Example programs
│   ├── echo_server.c          # Complete echo server using the framework
│   └── http_server.c          # Complete HTTP server (Milestone 4)
├── benchmarks/                # Performance tests
│   └── c10k_benchmark.c       # 10K connection benchmark
└── Makefile                   # Build system
```

File-to-Component Mapping:

File	Component	Key Functions	Milestone
`src/core/event_loop.c`	Event Loop	`event_loop_create`, `event_loop_register_fd`, `event_loop_run`	1, 3
`src/core/timer_wheel.c`	Timer Wheel	`timer_wheel_create`, `timer_wheel_add`, `timer_wheel_tick`	2
`src/core/connection.c`	Connection Management	`connection_create`, `connection_read`, `connection_write`	1, 4
`src/utils/task_queue.c`	Async Task Queue	`task_queue_push`, `task_queue_execute_all`	3
`src/protocols/http_handler.c`	HTTP Protocol Handler	`http_handle_read`, `http_handle_write`, `http_parse_incremental`	4

> **Decision: Directory Structure by Concern**
>
> - **Context**: We need to organize C code for a multi-component system that learners will implement incrementally across four milestones.
> - **Options Considered**:
>   1. **Flat structure**: All `.c` and `.h` files in one directory.
>   2. **Structure by milestone**: Separate directories for each milestone's code.
>   3. **Structure by component/concern**: Group files by architectural component (core, networking, protocols, utils).
> - **Decision**: Structure by component/concern.
> - **Rationale**:
>   - Flat structure becomes chaotic with 10+ files and doesn't reflect architectural boundaries.
>   - Structure by milestone creates artificial barriers and makes code reuse between milestones difficult.
>   - Structure by component mirrors the architectural design, makes dependencies clear, and facilitates independent testing of each component.
> - **Consequences**:
>   - Learners must understand the build system (Makefile) to compile across directories.
>   - Clear separation encourages thinking in terms of APIs between components.
>   - Natural organization for growing the project with additional protocols (WebSocket, custom protocols).

Header File Dependencies:

The header files create a clean API boundary between components. Here's how they depend on each other:

```
event_loop.h  ← (includes) → connection.h, timer_wheel.h (for integration)
timer_wheel.h ← standalone, no external dependencies
connection.h  ← standalone, defines core data structure
http_server.h ← (includes) → event_loop.h, connection.h (uses the framework)
```

This dependency structure means:

- The **event loop** knows about connections and timer wheel for integration.
- **Protocol handlers** depend on the event loop API but not vice versa (loose coupling).
- The **timer wheel** is independent and can be tested in isolation.

Common Pitfalls in Project Organization:

⚠️ Pitfall: Circular Header Dependencies

- **Description**: Creating header files that include each other (e.g., `event_loop.h` includes `http_server.h`, which includes `event_loop.h`).
- **Why it's wrong**: Causes compilation failures, unclear ownership, and tight coupling between components.
- **How to fix**: Design a clear dependency hierarchy. Use forward declarations (`struct connection;`) when types are referenced but not fully defined. The event loop framework should not include protocol-specific headers.

⚠️ Pitfall: Global Variables for Shared State

- **Description**: Using global variables for event loop instance, timer wheel, or connection table.
- **Why it's wrong**: Makes testing difficult, prevents multiple instances, and creates hidden dependencies.
- **How to fix**: Pass component instances as parameters. The `event_loop_run()` function should take a `struct event_loop*` parameter, not rely on a global.

⚠️ Pitfall: Mixing Platform-Specific Code with Core Logic

- **Description**: Putting `#ifdef LINUX` or `#ifdef MACOS` directly in the event loop core logic.

- **Why it's wrong**: Reduces readability and makes porting harder.
- **How to fix**: Isolate platform-specific code in wrapper functions (e.g., `socket_utils.c` for `set_nonblocking()`). The event loop should call `epoll_wait()` directly (it's Linux-specific by design), but other platform concerns are isolated.

Implementation Guidance

This implementation guidance provides the **starter code** and **skeleton structure** to begin implementing the architecture. Since the primary language is C, we'll focus on establishing the foundational files and types.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Build System	GNU Make with explicit rules	CMake with cross-platform support
Memory Management	Manual `malloc` / `free` with careful tracking	Reference-counted objects or memory pools
Data Structures	Simple linked lists and arrays	Hash tables for connection lookup, ring buffers for I/O
Testing	Simple test programs with `assert()`	Unity or Check testing frameworks
Logging	`fprintf(stderr, ...)` with log levels	Syslog integration or structured logging

B. Starter Header Files:

Create these files first to establish the type definitions and APIs:

`include/event_loop.h`:

```c
#ifndef EVENT_LOOP_H
#define EVENT_LOOP_H

#include <stdint.h>

struct connection;  // Forward declaration

struct event_loop {
    int epoll_fd;
    struct epoll_event* events;
    int max_events;
    int running;
    // TODO: Add fields for timer wheel integration
    // TODO: Add fields for task queue integration
};

// Callback types for I/O events
typedef void (*read_callback_t)(struct connection* conn);
typedef void (*write_callback_t)(struct connection* conn);
typedef void (*timer_callback_t)(void* user_data);

// Event loop API
struct event_loop* event_loop_create(int max_events);
int event_loop_register_fd(struct event_loop* loop, int fd,
                           uint32_t events, void* user_data);
int event_loop_run(struct event_loop* loop, int timeout_ms);
void event_loop_destroy(struct event_loop* loop);

#endif
```

`include/connection.h`:

```c
#ifndef CONNECTION_H

#define CONNECTION_H


#define READ_BUFFER_SIZE 4096

#define WRITE_BUFFER_SIZE 4096


struct connection {

    int fd;                     // File descriptor

    char* read_buffer;          // Data read from socket

    char* write_buffer;         // Data to write to socket

    void* timer_ref;            // Reference to associated timer

    int state;                  // Connection state (e.g., READING, WRITING)

    int read_buffer_used;       // Bytes currently in read buffer

    int write_buffer_used;      // Bytes pending in write buffer

    int write_buffer_sent;      // Bytes already sent from write buffer

    // TODO: Add protocol-specific state (e.g., HTTP parser state)

};


struct connection* connection_create(int fd);

void connection_destroy(struct connection* conn);

int connection_read(struct connection* conn);   // Returns bytes read

int connection_write(struct connection* conn);  // Returns bytes written


#endif
```

`include/timer_wheel.h`:

```c
#ifndef TIMER_WHEEL_H
#define TIMER_WHEEL_H

#include <stdint.h>

typedef void (*timer_callback_t)(void* user_data);

struct timer {
    uint64_t expires_at;        // Absolute expiration time (ms)
    timer_callback_t callback;
    void* user_data;
    struct timer* next;         // For chaining in wheel slot
};

struct timer_wheel {
    // TODO: Define hierarchical wheel structure
    // Suggested: 64ms slots, 64 slots per level, 4 levels
    // This gives ~4.6 hours maximum timeout
};

struct timer_wheel* timer_wheel_create(void);
void* timer_wheel_add(struct timer_wheel* wheel, uint64_t timeout_ms,
                      timer_callback_t callback, void* user_data);
void timer_wheel_remove(struct timer_wheel* wheel, void* timer_ref);
uint64_t timer_wheel_next_expiration(struct timer_wheel* wheel);
void timer_wheel_tick(struct timer_wheel* wheel);

#endif
```

C. Build System Starter (`Makefile`):

```makefile
CC = gcc
CFLAGS = -Wall -Wextra -g -I./include
LDFLAGS = -lm

# Source directories
SRC_DIR = src
CORE_DIR = $(SRC_DIR)/core
NETWORKING_DIR = $(SRC_DIR)/networking
PROTOCOLS_DIR = $(SRC_DIR)/protocols
UTILS_DIR = $(SRC_DIR)/utils

# Object files for core components
CORE_OBJS = $(CORE_DIR)/event_loop.o \
            $(CORE_DIR)/timer_wheel.o \
            $(CORE_DIR)/connection.o

NETWORKING_OBJS = $(NETWORKING_DIR)/socket_utils.o \
                  $(NETWORKING_DIR)/listening_socket.o

UTILS_OBJS = $(UTILS_DIR)/task_queue.o \
             $(UTILS_DIR)/logging.o

# Example servers
ECHO_SERVER_OBJS = examples/echo_server.o $(PROTOCOLS_DIR)/echo_handler.o
HTTP_SERVER_OBJS = examples/http_server.o $(PROTOCOLS_DIR)/http_handler.o

all: echo_server http_server

# Core components
$(CORE_DIR)/%.o: $(CORE_DIR)/%.c include/%.h
        $(CC) $(CFLAGS) -c $< -o $@

# Networking utilities
$(NETWORKING_DIR)/%.o: $(NETWORKING_DIR)/%.c
        $(CC) $(CFLAGS) -c $< -o $@

# Echo server example
echo_server: $(ECHO_SERVER_OBJS) $(CORE_OBJS) $(NETWORKING_OBJS) $(UTILS_OBJS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

# HTTP server example
http_server: $(HTTP_SERVER_OBJS) $(CORE_OBJS) $(NETWORKING_OBJS) $(UTILS_OBJS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

# Clean build
clean:
        rm -f $(CORE_DIR)/*.o $(NETWORKING_DIR)/*.o $(PROTOCOLS_DIR)/*.o \
              $(UTILS_DIR)/*.o examples/*.o echo_server http_server

.PHONY: all clean
```

D. Core Logic Skeleton for Event Loop Creation:

`src/core/event_loop.c` (partial skeleton):

```c
#include "event_loop.h"
#include "connection.h"
#include <sys/epoll.h>
#include <stdlib.h>
#include <errno.h>

struct event_loop* event_loop_create(int max_events) {
    struct event_loop* loop = malloc(sizeof(struct event_loop));
    if (!loop) return NULL;

    // TODO 1: Create epoll instance using epoll_create1(EPOLL_CLOEXEC)
    // TODO 2: Check for error (-1) and handle appropriately
    // TODO 3: Allocate memory for events array (max_events * sizeof(struct epoll_event))
    // TODO 4: Initialize loop fields: epoll_fd, events, max_events, running = 1
    // TODO 5: Return loop pointer (or NULL on failure)

    return loop;
}

int event_loop_register_fd(struct event_loop* loop, int fd,
                           uint32_t events, void* user_data) {
    struct epoll_event ev;
    ev.events = events;
    ev.data.ptr = user_data;  // Store connection pointer

    // TODO 1: Call epoll_ctl(EPOLL_CTL_ADD, ...) to register fd
    // TODO 2: Handle error cases (e.g., EEXIST if already registered)
    // TODO 3: Return 0 on success, -1 on error
    return 0;
}

int event_loop_run(struct event_loop* loop, int timeout_ms) {
    while (loop->running) {
        // TODO 1: Calculate actual timeout:
        //   - If timeout_ms == -1, wait indefinitely
```

```
        //   - Otherwise, use min(timeout_ms, next_timer_expiration)
        // TODO 2: Call epoll_wait() to get ready events
        // TODO 3: Handle EINTR (signal interruption) by continuing loop
        // TODO 4: For each ready event:
        //   a. Extract connection pointer from ev.data.ptr
        //   b. If (ev.events & EPOLLIN), call read callback
        //   c. If (ev.events & EPOLLOUT), call write callback
        //   d. If (ev.events & EPOLLERR or EPOLLHUP), handle error/close
        // TODO 5: After processing events, tick timer wheel
        // TODO 6: Execute any pending tasks from async task queue
    }
    return 0;
}
```

E. Language-Specific Hints for C:

1. **Memory Management**: Always check `malloc()` return values. Use `valgrind` to detect memory leaks during development.
2. **Error Handling**: Check all system call returns. Use `perror()` for debugging, but implement proper error recovery in production code.
3. **Non-blocking I/O**: After `read()` or `write()` returns `-1`, check `errno == EAGAIN || errno == EWOULDBLOCK` to distinguish actual errors from "would block" conditions.
4. **Build Dependencies**: Use `-D_GNU_SOURCE` in `CFLAGS` to enable Linux-specific features like `EPOLL_CLOEXEC` and `accept4()`.
5. **Type Safety**: Use `void*` for user data in callbacks, but cast back to specific types (e.g., `struct connection*`) inside callbacks with clear documentation.

F. Milestone Checkpoint for High-Level Structure:

After setting up the project structure and creating the skeleton files:

1. **Build the project**: Run `make` in the project root directory.
   - **Expected**: Successful compilation with warnings but no errors.
   - **Signs of trouble**: Missing header files, undefined references, or syntax errors.
2. **Verify header dependencies**: Create a simple `test.c` that includes all headers:

```c
#include "event_loop.h"
#include "timer_wheel.h"
#include "connection.h"
int main() { return 0; }
```

Compile with `gcc -I./include -c test.c`.

- **Expected**: Clean compilation.
- **Signs of trouble**: Circular dependencies or missing forward declarations.

3. **Run basic example**: Implement a minimal `event_loop_create()` and compile an empty echo server skeleton.
    - **Command**: `make echo_server` (should link successfully even with stub implementations).
    - **Next step**: The echo server won't work yet, but the build system should be functional.

This foundation establishes the architectural boundaries and file structure. Each subsequent milestone will flesh out the components within this organized framework, ensuring clean separation of concerns and maintainable code growth.

Data Model

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section defines the **core data structures** that make the event loop work efficiently at scale. Think of these structures as the DNA of your server—they encode how connections are tracked, how timeouts are managed, and how events trigger callbacks. The design must balance memory efficiency against fast access, since at 10,000+ connections, even small per-connection overhead compounds dramatically.

Core Types: Connection, Timer, Callback

At the heart of any event-driven server are three fundamental concepts: **connections** representing active client sockets, **timers** for scheduling future events, and **callbacks** as the glue that connects I/O events to your application logic. The data structures for these must support all the operations the event loop needs while minimizing memory fragmentation and pointer chasing.

Connection Structure

Imagine each connection as a **work-in-progress envelope** at a postal sorting facility. The envelope has designated slots for incoming data (read buffer) and outgoing data (write buffer), a tracking number (file descriptor), a status indicator (state), and a timer reference that says when this envelope will be discarded if no activity occurs. This self-contained design allows the event loop to process thousands of connections without maintaining separate lookup tables.

> **Decision: Unified Connection Context Object**
>
> - **Context**: We need to associate application state with each file descriptor registered with epoll. When `epoll_wait` returns an event, we must quickly access all relevant state for that connection.
> - **Options Considered**:
>   1. **Separate Hash Table**: Store connection state in a hash table keyed by file descriptor, with epoll events only carrying the fd.
>   2. **Embedded Pointer**: Store connection pointer directly in epoll's `data.ptr` field.
>   3. **File Descriptor as Index**: Use fd as an index into a pre-allocated array of connection objects.
> - **Decision**: Use embedded pointer (option 2) with a unified `connection` structure.
> - **Rationale**: The `epoll_event.data.ptr` field provides O(1) access without additional lookups. This eliminates hash collisions and extra memory accesses. The unified structure keeps all related data in one cache-friendly allocation.
> - **Consequences**: Connection objects must be allocated/freed carefully to avoid dangling pointers. The pointer remains valid until explicitly removed from epoll and freed.

Field	Type	Description
`fd`	`int`	The file descriptor for the socket. This is the primary identifier for the connection at the OS level.
`read_buffer`	`char*`	Pointer to dynamically allocated buffer for incoming data. Sized by `READ_BUFFER_SIZE` (typically 4KB).
`write_buffer`	`char*`	Pointer to dynamically allocated buffer for outgoing data. Sized by `WRITE_BUFFER_SIZE` (typically 4KB).
`timer_ref`	`void*`	Opaque reference to a timer in the timer wheel, used for cancelling idle timeouts when activity occurs.
`state`	`int`	Current protocol state (e.g., `STATE_READING`, `STATE_WRITING`, `STATE_CLOSING`). Helps manage connection lifecycle.
`read_buffer_used`	`int`	Number of valid bytes currently in `read_buffer`. Tracks how much data has arrived but not yet processed.
`write_buffer_used`	`int`	Number of bytes to send currently in `write_buffer`. Total data queued for writing.
`write_buffer_sent`	`int`	Number of bytes from `write_buffer` already written to socket. Used for partial writes with `EPOLLOUT`.

> **Critical Insight**: The `timer_ref` field creates a **bidirectional link** between connections and timers. When data arrives, we can cancel the old idle timeout and schedule a new one without searching the timer wheel. This is essential for O(1) timeout reset operations.

Timer Structure

Timers are like **kitchen timer dials** in a large restaurant kitchen. Each dial has a specific expiration time and is linked to a particular dish (connection). The hierarchical timer wheel organizes these dials into slots based on when they'll ring. The timer structure itself just needs to know when it expires, what function to call, and what connection it belongs to.

Field	Type	Description
`expires_at`	`uint64_t`	Absolute expiration time in milliseconds since an arbitrary epoch (typically server start). Used for ordering and comparison.
`callback`	`timer_callback_t`	Function pointer to call when timer expires. Signature: `void (*)(void* user_data)`.
`user_data`	`void*`	Argument passed to callback, usually a pointer to the associated `connection`.
`next`	`struct timer*`	Pointer to next timer in the same slot (linked list for collision resolution).

Design Rationale for Linked List Chaining: The timer wheel uses a hash table with separate chaining. Each wheel slot holds a linked list of timers that expire in that time range. This approach handles hash collisions gracefully and maintains O(1) average insertion/deletion when combined with the hierarchical wheel algorithm.

Callback Registration Pattern

While there's no single `callback` structure, the system uses a **registration pattern** where callbacks are associated with file descriptors through the `connection` object and with timers through the `timer` structure. Think of this as **event subscriptions**—you're telling the event loop: "When this socket becomes readable, call my read handler; when this timer expires, call my timeout handler."

Callback Types in the System:

1. **I/O Callbacks**: Functions registered via `event_loop_register_fd()` that get called when `EPOLLIN` or `EPOLLOUT` events occur.
2. **Timer Callbacks**: Functions attached to `timer` objects that execute when `expires_at` is reached.
3. **Deferred Tasks**: Functions placed in a queue to execute in the next event loop iteration (not associated with specific FDs or timers).

⚠️ **Pitfall: Callback Lifetime Management Mistake**: Storing direct pointers to stack-allocated data in `user_data` fields. **Why it's wrong**: When a connection closes and its `connection` object is freed, any pending timer or callback that references it will cause a use-after-free crash. **How to fix**: Use **reference counting** or ensure all timers are cancelled before freeing connection objects. The `timer_ref` field allows cancellation before `connection_destroy()`.

Event Loop State and Timer Wheel Slots

Event Loop State Structure

The `event_loop` is the **central dispatcher**—like an air traffic control tower that monitors all runways (file descriptors) and directs planes (connections) when they're ready for takeoff or landing. This structure holds the epoll instance, the array for returned events, and control flags.

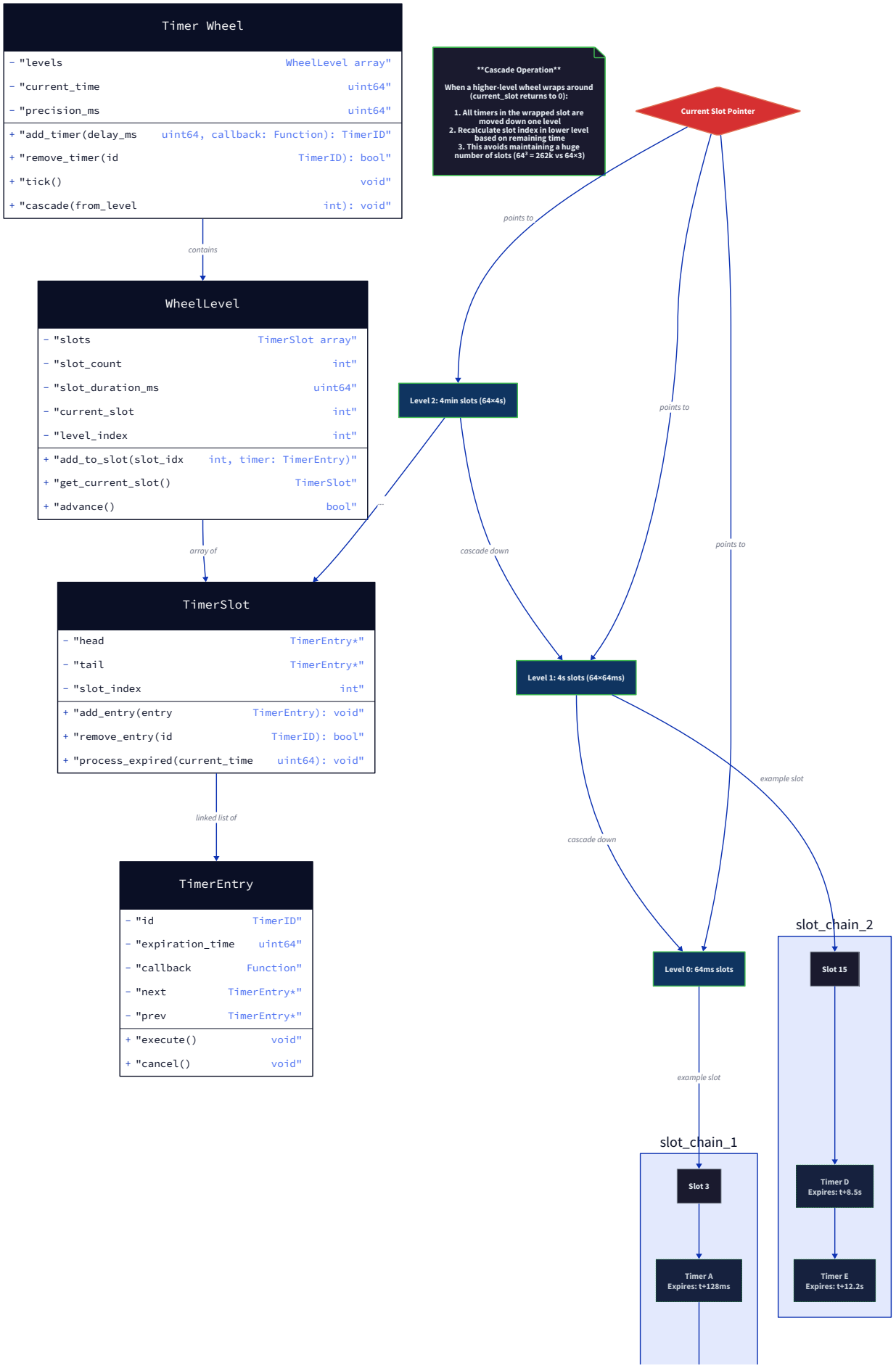
Field	Type	Description
`epoll_fd`	`int`	File descriptor for the epoll instance created with `epoll_create1()`. All monitored FDs are registered here.
`events`	`struct epoll_event*`	Dynamically allocated array of epoll events returned by `epoll_wait()`. Sized by `max_events`.
`max_events`	`int`	Maximum number of events `epoll_wait()` can return per call. Typically set to expected maximum concurrent connections.
`running`	`int`	Boolean flag (0/1) indicating whether the event loop should continue running. Set to 0 to initiate graceful shutdown.
`timer_wheel`	`struct timer_wheel*`	Pointer to the hierarchical timer wheel instance. Integrated directly for timeout management.
`task_queue`	`struct deferred_task*`	Head pointer to a linked list of deferred tasks scheduled for next loop iteration.

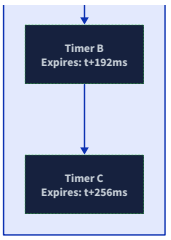
Decision: Integration of Timer Wheel into Event Loop

- **Context**: The event loop needs to check for timer expirations efficiently without adding complexity to the main loop.
- **Options Considered**:
  1. **Separate Timer Thread**: Run timer management in a separate thread that signals the main loop via pipe or eventfd.
  2. **epoll_wait Timeout**: Use the `timeout` parameter of `epoll_wait()` to sleep until the next timer expires.
  3. **Separate Tick Calls**: Call `timer_wheel_tick()` unconditionally in every loop iteration.
- **Decision**: Combine options 2 and 3—store timer wheel in event loop and use `epoll_wait()` timeout.
- **Rationale**: This avoids thread synchronization complexity while ensuring timers fire with reasonable accuracy. The `epoll_wait` timeout prevents busy-waiting when no I/O events occur.
- **Consequences**: Timer resolution is limited by `epoll_wait`'s millisecond timeout granularity. Long timeouts may cause delayed firing if calculated incorrectly.

Hierarchical Timer Wheel Data Layout

The hierarchical timer wheel is like a **multi-level gear system** where each gear represents a different time scale. Imagine a clock with seconds, minutes, and hours hands—timers scheduled within the next minute go to the seconds wheel, those within the hour go to the minutes wheel, and so on. This allows O(1) operations for most timers while bounding cascade operations.


Timer Wheel
- "levels WheelLevel array"
- "current_time uint64"
- "precision_ms uint64"
+ "add_timer(delay_ms uint64, callback: Function): TimerID"
+ "remove_timer(id TimerID): bool"
+ "tick() void"
+ "cascade(from_level int): void"
contains
WheelLevel
- "slots TimerSlot array"
- "slot_count int"
- "slot_duration_ms uint64"
- "current_slot int"
- "level_index int"
+ "add_to_slot(slot_idx int, timer: TimerEntry)"
+ "get_current_slot() TimerSlot"
+ "advance() bool"
array of
TimerSlot
- "head TimerEntry*"
- "tail TimerEntry*"
- "slot_index int"
+ "add_entry(entry TimerEntry): void"
+ "remove_entry(id TimerID): bool"
+ "process_expired(current_time uint64): void"
linked list of
TimerEntry
- "id TimerID"
- "expiration_time uint64"
- "callback Function"
- "next TimerEntry*"
- "prev TimerEntry*"
+ "execute() void"
+ "cancel() void"
Cascade Operation
When a higher-level wheel wraps around (current_slot returns to 0):
1. All timers in the wrapped slot are moved down one level
2. Recalculate slot index in lower level based on remaining time
3. This avoids maintaining a huge number of slots (64² = 262k vs 64×3)
Current Slot Pointer
points to
points to
points to
Level 2: 4min slots (64×4s)
Level 1: 4s slots (64×64ms)
Level 0: 64ms slots
cascade down
cascade down
example slot
example slot
slot_chain_1
Slot 3
Timer A
Expires: t+128ms
slot_chain_2
Slot 15
Timer D
Expires: t+8.5s
Timer E
Expires: t+12.2s



Wheel Levels and Their Purposes:

1. **Level 0 (Milliseconds)**: 64 slots, each representing 1ms. Covers timers up to 64ms out.
2. **Level 1 (Seconds)**: 64 slots, each representing 64ms (1 << 6). Covers timers up to ~4 seconds.
3. **Level 2 (Minutes)**: 64 slots, each representing 4.096s (64 * 64ms). Covers timers up to ~4 minutes.
4. **Level 3 (Hours)**: 64 slots, each representing 4.37min (64 * 4.096s). Covers timers up to ~4.6 hours.
5. **Level 4 (Days)**: 64 slots, each representing ~4.66h (64 * 4.37min). Covers timers up to ~12 days.

Timer Wheel Structure:

Field	Type	Description
`levels`	`int`	Number of levels in the hierarchy (typically 5 for the ranges above).
`slots_per_level`	`int`	Number of slots per level (power of 2, typically 64 for efficient modulo).
`resolution_ms`	`uint64_t`	Base timer resolution in milliseconds (typically 1ms for Level 0).
`current_time`	`uint64_t`	Current time in ms since epoch, updated on each `timer_wheel_tick()`.
`wheel`	`struct timer***`	2D array: `wheel[level][slot]` points to head of linked list of timers in that slot.
`count`	`uint64_t`	Total number of active timers in the wheel (for debugging and monitoring).

Slot Index Calculation Algorithm:

1. Determine time difference: `delta = expires_at - current_time`
2. For each level from 0 to levels-1:
   - If `delta < slots_per_level * (resolution_ms << (6 * level))`:
     - `slot = (current_slot[level] + (delta >> (6 * level))) & (slots_per_level - 1)`
     - Insert timer into `wheel[level][slot]` linked list
     - Return

> ⚠️ **Pitfall: Timer Cascading Overhead Mistake**: Placing a timer in a high-level slot when it actually belongs in a lower level. **Why it's wrong**: When the wheel advances, timers must "cascade" down through levels, requiring multiple re-insertions and degrading to O(n) performance. **How to fix**: Always compute the **correct level** using the precise formula above. During `timer_wheel_tick()`, when a slot at level > 0 expires, re-insert all its timers into appropriate lower-level slots.

Cascade Operation Example: When the current time advances from 63ms to 64ms:

1. Level 0 slot index increments from 63 to 0 (wraps around)
2. Level 1 slot index increments by 1 (since we've completed a full Level 0 rotation)
3. All timers in the now-expired Level 1 slot (slot that just became current) are removed and re-inserted into appropriate Level 0 slots based on their exact expiration times.

Connection State Machine

While not a separate data structure, the `state` field in `connection` follows a specific state machine that governs connection lifecycle:

Current State	Event	Next State	Actions
`STATE_ACCEPTED`	`EPOLLIN` with data	`STATE_READING`	Read available data, start parsing
`STATE_READING`	Complete request read	`STATE_WRITING`	Generate response, register for `EPOLLOUT` if buffer full
`STATE_WRITING`	`EPOLLOUT` ready	`STATE_WRITING`	Write buffered data, deregister `EPOLLOUT` when empty
`STATE_WRITING`	All data written	`STATE_READING` (keep-alive) or `STATE_CLOSING`	Reset buffers for next request or initiate close
Any state	Timer expires	`STATE_CLOSING`	Close socket, free resources
Any state	Read returns 0 (EOF)	`STATE_CLOSING`	Client closed connection
Any state	Error on socket	`STATE_CLOSING`	Clean up immediately

> **Critical Insight**: The state machine is **non-blocking at every transition**. Each state change corresponds to an event loop iteration, never waiting for I/O. This ensures single-threaded concurrency.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Memory Allocation	`malloc/free` per connection	Object pool with pre-allocation
Timer Resolution	1ms base resolution	Configurable resolution per level
Buffer Management	Fixed-size buffers (4KB)	Dynamic growth with watermarks
Time Source	`clock_gettime(CLOCK_MONOTONIC)`	`clock_gettime(CLOCK_MONOTONIC_RAW)`

B. Recommended File/Module Structure:

```
event-loop-project/
├── include/
│   ├── event_loop.h      # Event loop structures and API
│   ├── connection.h      # Connection structure and operations
│   ├── timer_wheel.h     # Timer wheel structures and API
│   └── http_parser.h     # HTTP parsing structures
├── src/
│   ├── event_loop.c      # Event loop implementation
│   ├── connection.c      # Connection management
│   ├── timer_wheel.c     # Timer wheel implementation
│   ├── http_parser.c     # Incremental HTTP parser
│   └── main.c           # Server entry point with example HTTP handler
└── tests/
    ├── test_timer.c      # Timer wheel unit tests
    └── test_http.c       # HTTP parser tests
```

C. Infrastructure Starter Code:

```c
/* include/event_loop.h */

#ifndef EVENT_LOOP_H

#define EVENT_LOOP_H


#include <stdint.h>

#include <sys/epoll.h>


#define MAX_EVENTS 1024

#define READ_BUFFER_SIZE 4096

#define WRITE_BUFFER_SIZE 4096


/* Connection states */

enum conn_state {

    STATE_ACCEPTED,

    STATE_READING,

    STATE_WRITING,

    STATE_CLOSING

};


/* Connection structure */

struct connection {

    int fd;

    char *read_buffer;

    char *write_buffer;

    void *timer_ref;

    int state;

    int read_buffer_used;

    int write_buffer_used;

    int write_buffer_sent;

    /* Protocol-specific data can be added here */

    void *proto_data;

};


/* Timer callback type */

typedef void (*timer_callback_t)(void *user_data);
```

```c
/* Event loop structure */
struct event_loop {
    int epoll_fd;
    struct epoll_event *events;
    int max_events;
    int running;
    struct timer_wheel *timer_wheel;
    struct deferred_task *task_queue;
};

/* API declarations */
struct event_loop *event_loop_create(int max_events);
int event_loop_register_fd(struct event_loop *loop, int fd,
                           uint32_t events, void *user_data);
int event_loop_run(struct event_loop *loop, int timeout_ms);
void event_loop_destroy(struct event_loop *loop);

/* Helper functions */
int set_nonblocking(int fd);
int create_listening_socket(int port, int backlog);

#endif /* EVENT_LOOP_H */
```

D. Core Logic Skeleton Code:

```c
/* src/connection.c */

#include "connection.h"

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>


struct connection *connection_create(int fd) {

    /* TODO 1: Allocate memory for connection struct using malloc() */

    /* TODO 2: Initialize fd field with provided file descriptor */

    /* TODO 3: Allocate read_buffer of size READ_BUFFER_SIZE */

    /* TODO 4: Allocate write_buffer of size WRITE_BUFFER_SIZE */

    /* TODO 5: Set timer_ref to NULL (no timer yet) */

    /* TODO 6: Set state to STATE_ACCEPTED */

    /* TODO 7: Initialize all buffer usage counters to 0 */

    /* TODO 8: Return the connection pointer, or NULL on failure */

    return NULL;

}


void connection_destroy(struct connection *conn) {

    if (!conn) return;

    /* TODO 1: Close the file descriptor if it's not -1 */

    /* TODO 2: Free read_buffer if not NULL */

    /* TODO 3: Free write_buffer if not NULL */

    /* TODO 4: Note: timer cancellation should happen BEFORE destroy */

    /* TODO 5: Free any protocol-specific data (proto_data) */

    /* TODO 6: Free the connection struct itself */

}


int connection_read(struct connection *conn) {

    /* TODO 1: Calculate available space: READ_BUFFER_SIZE - read_buffer_used */

    /* TODO 2: Use read() system call with non-blocking fd */

    /* TODO 3: If read returns >0, add to read_buffer_used */

    /* TODO 4: If read returns 0, client closed connection - return -1 */

    /* TODO 5: If read returns -1 with EAGAIN/EWOULDBLOCK, return 0 */
```

```
    /* TODO 6: If read returns -1 with other error, return -1 */

    /* TODO 7: Return number of bytes read on success */

    return 0;

}


int connection_write(struct connection *conn) {

    /* TODO 1: Calculate unsent data: write_buffer_used - write_buffer_sent */

    /* TODO 2: Use write() system call with pointer to unsent portion */

    /* TODO 3: If write returns >0, update write_buffer_sent */

    /* TODO 4: If write returns -1 with EAGAIN/EWOULDBLOCK, return 0 */

    /* TODO 5: If write returns -1 with other error, return -1 */

    /* TODO 6: If all data sent (write_buffer_sent == write_buffer_used):

          - Reset counters for next message

          - Return 0 to indicate completion */

    /* TODO 7: Return number of bytes written on success */

    return 0;

}
```

E. Language-Specific Hints:

- Use `epoll_create1(EPOLL_CLOEXEC)` to ensure the epoll fd is closed on `exec()`.
- Always check `errno` after system calls—`EAGAIN` and `EWOULDBLOCK` are expected for non-blocking I/O.
- For timer wheel slots, use `calloc()` to ensure all pointers are initially NULL.
- Store timestamps as `uint64_t` milliseconds using `clock_gettime(CLOCK_MONOTONIC)` to avoid wrap-around issues.
- When freeing connection buffers, use `free()` and immediately set pointers to NULL to prevent double-free.

F. Milestone Checkpoint: After implementing the data structures:

1. Run a simple test: `gcc -o test_connection src/connection.c -Iinclude && ./test_connection`
2. Expected: No crashes, memory usage should be stable (check with `valgrind`).
3. Verify structure sizes: `printf("connection: %zu bytes\n", sizeof(struct connection))` should show ~48 bytes on 64-bit systems.
4. Create a mock event loop that allocates 10,000 connection objects—memory should be ~500KB plus buffers.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Memory grows unbounded	Connections not freed on close	Add logging to `connection_destroy()`	Ensure `connection_destroy()` called for every closed fd
Timers fire too early	Wrong slot calculation	Log `expires_at`, `current_time`, and calculated slot	Debug timer wheel insertion logic
Partial data lost	Buffer counters incorrect	Log `read_buffer_used` before/after each read	Ensure counters updated atomically
High CPU usage	Busy loop in `epoll_wait`	Check `timeout_ms` calculation	Ensure `timer_wheel_next_expiration()` returns correct value

Component Design

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section details each core component of the event loop system, following the project milestones. Think of the event loop as the central nervous system—it senses stimuli (I/O events), processes them through reflex arcs (callbacks), and maintains homeostasis with timers. The timer wheel is the circadian rhythm, the task queue is short-term memory, and the HTTP handler is a specialized skill built on these foundations.

Event Loop (epoll Basics) - Milestone 1

The event loop is the **reactor**—the single-threaded coordinator that monitors all I/O sources and dispatches events to handlers. Imagine a single air traffic controller watching hundreds of radar screens (file descriptors) who directs each plane (connection) when it's clear for takeoff (write) or has arrived (read).

> **Decision: Level-Triggered vs Edge-Triggered epoll**
>
> - **Context**: epoll supports two notification modes: level-triggered (notifies while condition is true) and edge-triggered (notifies only on state changes). We need to choose for our educational server handling 10K+ connections.
> - **Options Considered**:
>   - Level-triggered (default): Simpler for learners, matches select/poll behavior, but may cause unnecessary wakeups if data isn't fully drained.
>   - Edge-triggered with EPOLLONESHOT: Most efficient, avoids thundering herd, but requires precise buffer management and re-arm operations.
>   - Hybrid: Start with level-triggered, optionally support edge-triggered.
> - **Decision**: Use **level-triggered** for Milestone 1-3, with edge-triggered as an advanced extension.
> - **Rationale**: Level-triggered is more forgiving—if a callback doesn't consume all data, epoll will notify again on the next iteration. This matches the incremental processing model and reduces complex re-registration logic for learners. The performance difference is negligible for educational scale.
> - **Consequences**: Slightly more epoll_wait returns when data remains, but simpler code and easier debugging. Learners can graduate to edge-triggered optimization later.

Option	Pros	Cons	Chosen?
Level-triggered (default)	Simple, self-correcting, matches select/poll mental model	May cause extra epoll_wait returns if data not fully read	✅ Yes for learning
Edge-triggered with EPOLLONESHOT	Maximum performance, minimizes syscalls	Complex buffer management, must re-arm after each event	No for core
Hybrid configurable	Best of both worlds	Increased complexity, two code paths	No

The event loop component owns the epoll instance, manages file descriptor registration, and executes the main dispatch loop. Its responsibilities include:

- Creating and configuring the epoll instance
- Registering and deregistering file descriptors for I/O events
- Executing the infinite dispatch loop that waits for events and processes them
- Managing the lifecycle of associated connection objects

Mental Model: The event loop is like a busy receptionist with an intercom system. Each office (file descriptor) has a button that lights up when someone enters (EPOLLIN) or when the outgoing tray is empty (EPOLLOUT). The receptionist watches all lights, and when one illuminates, they announce over the intercom: "Office 42, you have a visitor!" The office worker (callback) then handles the visitor at their own pace.

Core Data Structures:

Name	Type	Description
`event_loop`	struct	Main event loop state container
`epoll_event`	struct epoll_event	Linux kernel event structure

`event_loop` Structure Fields:

Field	Type	Description
`epoll_fd`	int	The epoll file descriptor returned by `epoll_create1()`
`events`	struct epoll_event*	Dynamically allocated array for epoll_wait results
`max_events`	int	Capacity of the events array (configurable)
`running`	int	Boolean flag: 1 while loop should continue, 0 to stop
`timer_wheel`	struct timer_wheel*	Pointer to associated timer wheel (Milestone 2)
`task_queue`	struct deferred_task*	Head pointer to deferred task queue (Milestone 3)

`epoll_event` Structure Fields (Linux standard):

Field	Type	Description
`events`	uint32_t	Bitmask of epoll events (EPOLLIN, EPOLLOUT, etc.)
`data`	union	User data attached to event (we store `connection*` in `ptr`)

Event Loop Interface:

Method	Parameters	Returns	Description
`event_loop_create`	`max_events int`	`struct event_loop*`	Allocates and initializes event loop with given epoll_wait capacity
`event_loop_register_fd`	`loop struct event_loop*`, `fd int`, `events uint32_t`, `user_data void*`	`int` (0 success, -1 error)	Registers FD with epoll for specified events, attaches user_data
`event_loop_modify_fd`	`loop struct event_loop*`, `fd int`, `events uint32_t`, `user_data void*`	`int`	Modifies existing FD registration (e.g., add EPOLLOUT when write buffer full)
`event_loop_unregister_fd`	`loop struct event_loop*`, `fd int`	`int`	Removes FD from epoll monitoring
`event_loop_run`	`loop struct event_loop*`, `timeout_ms int`	`int`	Main dispatch loop; runs until `running` becomes 0
`event_loop_stop`	`loop struct event_loop*`	`void`	Sets `running` to 0 to break out of loop

Algorithm: Event Loop Dispatch (single iteration):

1. Calculate timeout for `epoll_wait`: minimum of user-provided timeout and next timer expiration (from timer wheel)
2. Call `epoll_wait(epoll_fd, events, max_events, timeout_ms)` to wait for I/O events
3. For each returned event (0 to n events):
    1. Extract `connection*` from `event.data.ptr`
    2. If `event.events & EPOLLIN`: call connection's read callback
    3. If `event.events & EPOLLOUT`: call connection's write callback
    4. If `event.events & (EPOLLHUP | EPOLLERR)`: initiate connection cleanup
4. Call `timer_wheel_tick()` to process expired timers
5. Process all deferred tasks in the task queue
6. Return to step 1 if `running` is still true

Connection State Machine (simplified for Milestone 1):

Current State	Event	Next State	Action Taken
`STATE_ACCEPTED`	`EPOLLIN` received	`STATE_READING`	Call `connection_read()`, if data arrives, transition to reading
`STATE_READING`	Read returns data	`STATE_WRITING`	Process data, prepare response, register for `EPOLLOUT`
`STATE_READING`	Read returns 0 (EOF)	`STATE_CLOSING`	Initiate connection closure
`STATE_WRITING`	`EPOLLOUT` received	`STATE_READING` or `STATE_CLOSING`	Call `connection_write()`, if all data sent, deregister `EPOLLOUT`, return to reading (keep-alive) or close
Any state	`EPOLLERR` or `EPOLLHUP`	`STATE_CLOSING`	Clean up connection resources

Common Pitfalls for Milestone 1:

⚠️ Pitfall: Blocking accept() on listening socket

- **Description**: Forgetting to set `O_NONBLOCK` on the listening socket causes `accept()` to block the entire event loop when no connections are pending.
- **Why it's wrong**: Event loop must never block; a single blocking call stalls all other connections.
- **Fix**: Always call `set_nonblocking()` on the listening socket after creation.

⚠️ **Pitfall: Ignoring EAGAIN/EWOULDBLOCK**

- **Description**: Treating `read()` / `write()` returning -1 with `errno == EAGAIN` as an error and closing the connection.
- **Why it's wrong**: These are normal non-blocking I/O conditions meaning "try later," not errors.
- **Fix**: Check `errno` after -1 return: if `EAGAIN` or `EWOULDBLOCK`, simply return 0 bytes processed and continue.

⚠️ **Pitfall: Starving writes while processing reads**

- **Description**: Processing all `EPOLLIN` events before any `EPOLLOUT` events in the same loop iteration.
- **Why it's wrong**: A connection with both read and write ready might have its write delayed, causing buffer buildup.
- **Fix**: Process both events for each connection before moving to the next, or use round-robin.

> **Key Insight**: The event loop's efficiency comes from **never waiting** for any single operation. It always has work to do—either process ready I/O, handle timers, or execute deferred tasks. This is why non-blocking operations are non-negotiable.

Timer Wheel (Timer Wheel and Timeouts) - Milestone 2

The timer wheel manages time-based events with **O(1) complexity** for insertion and deletion. Imagine a multi-tiered revolving door system in a large hotel: the innermost door rotates every minute (64ms slots), the middle every hour (4s slots), and the outer every day (4min slots). Guests (timers) enter at the appropriate door based on their departure time, cascading inward as time progresses.

> **Decision: Hierarchical Timer Wheel vs Min-Heap**
>
> - **Context**: We need efficient timeout management for 10K+ connections with frequent insertions (new timeouts) and deletions (cancellations when activity occurs).
> - **Options Considered**:
> - **Min-heap (priority queue)**: O(log n) insert/delete, O(1) min retrieval. Simple implementation.
> - **Simple wheel (single-level)**: O(1) operations but limited to fixed future range (wheel size × resolution).
> - **Hierarchical (multilevel) wheel**: O(1) operations with arbitrary future range via cascading.
> - **Time-sorted list**: O(n) operations, unacceptable at scale.
> - **Decision**: **Hierarchical timer wheel** with 4 levels (64ms, 4s, 4min, 4hr).
> - **Rationale**: For connection timeouts, insertions and cancellations vastly outnumber expirations. O(1) operations matter most. The hierarchical design supports the full 32-bit timeout range (≈49 days) while maintaining constant time. The cascading overhead is amortized over many ticks.
> - **Consequences**: More complex implementation than min-heap, but predictable performance regardless of timer count.

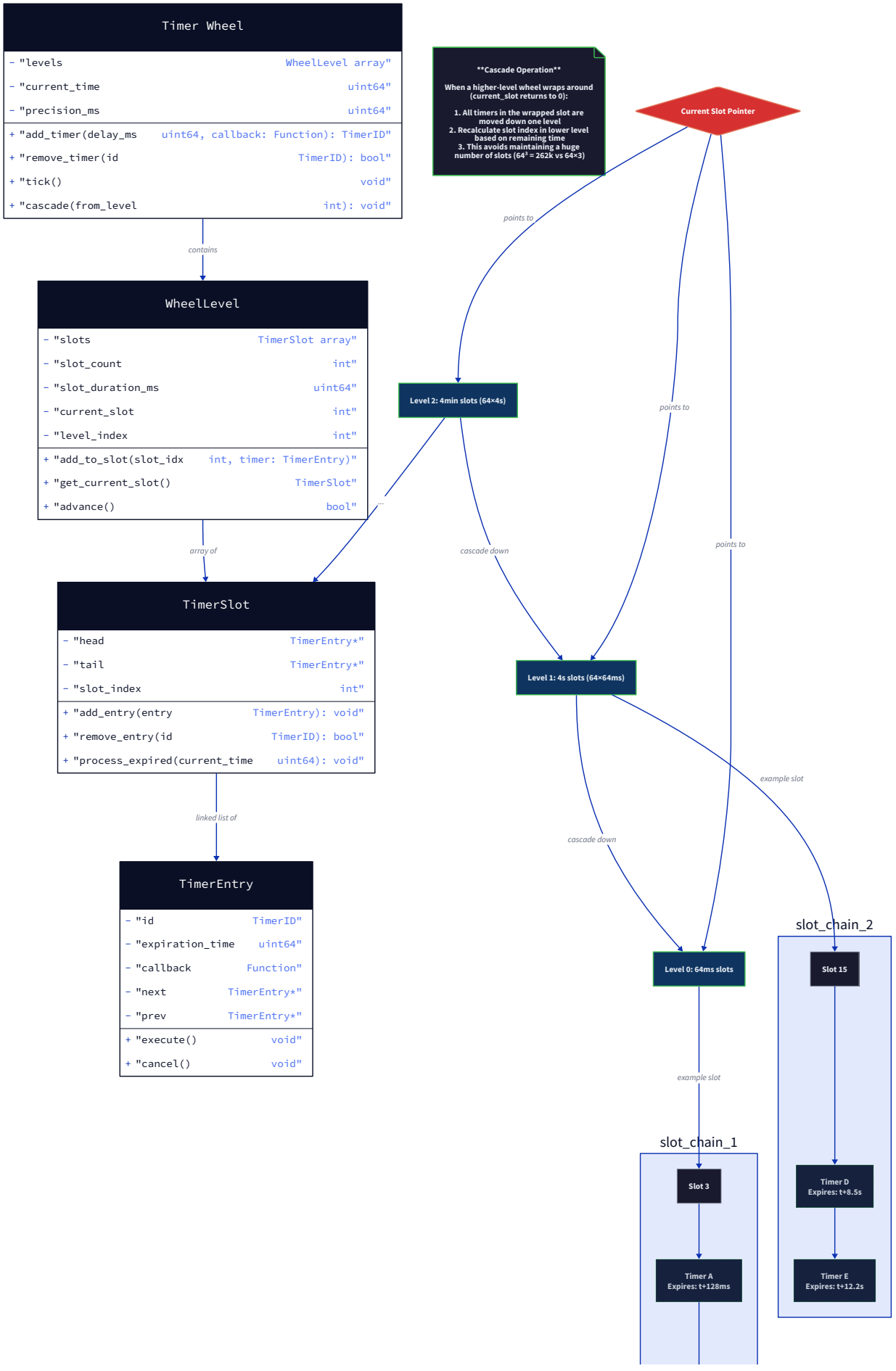
Option	Pros	Cons	Chosen?
Min-heap	Simple, standard, supports arbitrary times	O(log n) operations, heapify overhead	No
Simple wheel	True O(1), very simple	Limited to wheel span (e.g., 2^8 × 64ms = 16s)	No
Hierarchical wheel	O(1), arbitrary range, predictable	Complex cascading logic, fixed overhead	✅ Yes
Hashed timing wheel	O(1) average, handles large ranges	Hash collision chains, less predictable	No

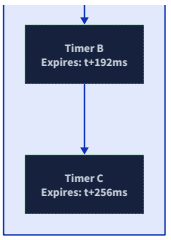
The timer wheel component manages the lifecycle of all timed events. Its responsibilities include:

- Inserting new timers with arbitrary timeout values

- Efficiently advancing the current time and expiring due timers
- Supporting timer cancellation (e.g., when a connection receives data before its idle timeout)
- Calculating the next expiration time for `epoll_wait` timeout optimization

Mental Model: The timer wheel is like a multi-layered egg timer. You set a timer by placing an egg in the appropriate slot of the appropriate wheel. Every tick (64ms), the innermost wheel rotates one slot. If that slot has eggs, they hatch (fire). When a wheel completes a full rotation, it rotates the next outer wheel by one slot, moving all eggs from that outer slot to the appropriate inner wheel slots (cascading).


Timer Wheel
- "levels WheelLevel array"
- "current_time uint64"
- "precision_ms uint64"
+ "add_timer(delay_ms uint64, callback: Function): TimerID"
+ "remove_timer(id TimerID): bool"
+ "tick() void"
+ "cascade(from_level int): void"
contains
WheelLevel
- "slots TimerSlot array"
- "slot_count int"
- "slot_duration_ms uint64"
- "current_slot int"
- "level_index int"
+ "add_to_slot(slot_idx int, timer: TimerEntry)"
+ "get_current_slot() TimerSlot"
+ "advance() bool"
array of
TimerSlot
- "head TimerEntry*"
- "tail TimerEntry*"
- "slot_index int"
+ "add_entry(entry TimerEntry): void"
+ "remove_entry(id TimerID): bool"
+ "process_expired(current_time uint64): void"
linked list of
TimerEntry
- "id TimerID"
- "expiration_time uint64"
- "callback Function"
- "next TimerEntry*"
- "prev TimerEntry*"
+ "execute() void"
+ "cancel() void"
Cascade Operation
When a higher-level wheel wraps around (current_slot returns to 0):
1. All timers in the wrapped slot are moved down one level
2. Recalculate slot index in lower level based on remaining time
3. This avoids maintaining a huge number of slots (64² = 262k vs 64×3)
Current Slot Pointer
points to
points to
points to
Level 2: 4min slots (64×4s)
Level 1: 4s slots (64×64ms)
Level 0: 64ms slots
cascade down
cascade down
example slot
example slot
slot_chain_1
Slot 3
Timer A
Expires: t+128ms
slot_chain_2
Slot 15
Timer D
Expires: t+8.5s
Timer E
Expires: t+12.2s



Core Data Structures:

Name	Type	Description
`timer_wheel`	struct	Main timer wheel container
`timer`	struct	Individual timer entry stored in wheel slots

`timer_wheel` Structure Fields:

Field	Type	Description
`levels`	int	Number of wheel levels (e.g., 4)
`slots_per_level`	int	Slots per level (power of 2, e.g., 64)
`resolution_ms`	uint64_t	Milliseconds per tick of innermost wheel (e.g., 64)
`current_time`	uint64_t	Current time in wheel ticks (monotonically increasing)
`wheel`	struct timer***	3D array: levels × slots × timer linked list heads
`count`	uint64_t	Total active timers for statistics

`timer` Structure Fields:

Field	Type	Description
`expires_at`	uint64_t	Absolute expiration time in wheel ticks
`callback`	timer_callback_t	Function pointer to call on expiration
`user_data`	void*	Context data passed to callback (e.g., connection*)
`next`	struct timer*	Next timer in same slot (singly linked list)

Timer Wheel Interface:

Method	Parameters	Returns	Description
`timer_wheel_create`	(none)	`struct timer_wheel*`	Creates wheel with default levels (4) and slots (64)
`timer_wheel_add`	`wheel struct timer_wheel*`, `timeout_ms uint64_t`, `callback timer_callback_t`, `user_data void*`	`void*` (timer reference)	Schedules callback after `timeout_ms`, returns opaque ref for cancellation
`timer_wheel_remove`	`wheel struct timer_wheel*`, `timer_ref void*`	`void`	Cancels pending timer using reference returned by add
`timer_wheel_next_expiration`	`wheel struct timer_wheel*`	`uint64_t`	Returns milliseconds until next timer expires (for epoll_wait timeout)
`timer_wheel_tick`	`wheel struct timer_wheel*`	`void`	Advances time by one tick (called from event loop), expires due timers
`timer_wheel_current_time`	`wheel struct timer_wheel*`	`uint64_t`	Returns current time in milliseconds for relative calculations

Algorithm: Timer Wheel Insertion:

1. Calculate absolute expiration: `current_time + (timeout_ms / resolution_ms)`
2. Determine level and slot:
    1. For each level i from 0 to levels-1:
        1. Calculate range of this level: `range = (slots_per_level ^ (i+1)) * resolution_ms`
        2. If `timeout_ms < range`:
            1. Calculate slot: `slot = (current_time + timeout_ms) >> (i * log2(slots_per_level)) % slots_per_level`
            2. Insert timer into list at `wheel[i][slot]`
            3. Store level and slot in timer reference for O(1) removal
            4. Break
3. If timeout exceeds maximum range of wheel, cap to maximum (rare, handled by highest level)

Algorithm: Timer Wheel Tick and Cascading:

1. Increment `current_time` by 1 tick
2. For level 0 (innermost):
    1. Calculate current slot: `slot = current_time % slots_per_level`
    2. Remove all timers from `wheel[0][slot]`
    3. For each removed timer, execute its callback
3. For each level i where `current_time % (slots_per_level ^ (i+1)) == 0`:
    1. Calculate previous slot in level i+1
    2. Remove all timers from that slot in level i+1
    3. For each removed timer, reinsert into appropriate lower-level slot (cascade down)
    4. Continue to next level if cascading triggered further wheel rotation

Connection Timeout Integration:

- When a connection becomes active (read/write), cancel its existing idle timer
- After processing, schedule new idle timer (e.g., 30 seconds)

- On timeout expiration, callback receives `connection*` as user_data and initiates closure

Common Pitfalls for Milestone 2:

⚠️ **Pitfall: Timer drift from inaccurate epoll_wait timeout**

- **Description**: Using fixed `epoll_wait` timeout (e.g., 100ms) instead of minimum until next timer expiration.
- **Why it's wrong**: Timers fire late, causing idle timeouts to extend beyond configured values.
- **Fix**: Call `timer_wheel_next_expiration()` before each `epoll_wait` and use that as timeout.

⚠️ **Pitfall: Not handling timer cancellation races**

- **Description**: Timer fires and callback begins executing while connection simultaneously receives data and attempts cancellation.
- **Why it's wrong**: Connection could be closed while processing data, causing use-after-free.
- **Fix**: Use timer reference invalidation: set a cancelled flag in timer or connection, check in callback before acting.

⚠️ **Pitfall: Wheel slots not power of two**

- **Description**: Using 100 slots per level for "nice decimal numbers."
- **Why it's wrong**: Modulo operations become expensive divisions instead of bitwise AND.
- **Fix**: Always use power of two slots (64, 128, 256) for fast `slot = time & (slots-1)`.

> **Key Insight**: The timer wheel's efficiency comes from **batching time computations**. Instead of checking each timer individually ("is it time yet?"), we bucket timers by expiration slot and only examine buckets whose time has come. This transforms O(n) scan into O(1) bucket removal.

Async Task Scheduling - Milestone 3

Async task scheduling provides the **callback API** that makes the reactor pattern usable. Imagine a restaurant with a single waiter (event loop) who takes orders (I/O events) and gives them to the kitchen (callbacks). The kitchen can also give the waiter future tasks ("check table 5 in 10 minutes") that go into the waiter's pocket notebook (deferred queue).

> **Decision: Immediate vs Deferred Callback Execution**
>
> - **Context**: When an I/O event occurs or timer fires, we need to decide when to execute the associated callback relative to other events.
> - **Options Considered**:
> - **Immediate execution**: Callback runs directly in the event processing loop.
> - **Deferred to queue**: Callback added to task queue, executed after all I/O events in current iteration.
> - **Hybrid**: I/O callbacks immediate, timer callbacks deferred.
> - **Decision**: **I/O callbacks immediate, timer and deferred tasks queued**.
> - **Rationale**: I/O callbacks should run immediately to drain buffers and maintain throughput. Timer callbacks (like connection timeout) often involve cleanup that might modify event registrations, which is safer deferred. This follows the Reactor pattern's "don't call us, we'll call you" semantics.
> - **Consequences**: Slightly more complex dispatch logic, but avoids re-entrancy issues during I/O processing.

Option	Pros	Cons	Chosen?
All immediate	Simple, predictable order	Callbacks can modify event set during iteration (dangerous)	No
All deferred	Safe, deterministic execution order	I/O delays by one tick, reduces responsiveness	No
Hybrid (I/O immediate, others deferred)	Responsive I/O, safe modifications	Two execution paths, more complex	✅ Yes
Configurable per callback	Maximum flexibility	Very complex, hard to reason about	No

The async scheduling component manages the registration and execution of all callbacks. Its responsibilities include:

- Providing clean API for registering interest in file descriptor events
- Managing callback context (user data) lifecycle
- Maintaining deferred task queue for non-I/O work
- Supporting one-shot and repeating timers via timer wheel integration

Mental Model: The async scheduler is like a theater stage manager. Actors (file descriptors) signal when they're ready (I/O events). The stage manager immediately whispers their lines (I/O callbacks). Meanwhile, stagehands (timers) place notes in the manager's "future cues" box (timer wheel). During scene breaks, the manager checks this box and processes any due notes (timer callbacks). Any actor requesting a costume change (deferred task) gets queued for the next break.

Core Data Structures:

Name	Type	Description
`deferred_task`	struct	Entry in deferred task queue
`callback_registry`	struct	Maps file descriptors to their callbacks (simplified)

`deferred_task` Structure Fields:

Field	Type	Description
`callback`	task_callback_t	Function to execute
`user_data`	void*	Context data
`next`	struct deferred_task*	Next task in queue

Callback Types (function pointer signatures):

Type	Signature	Purpose
`read_callback_t`	`void (*)(int fd, void* user_data)`	Called when FD has data ready to read
`write_callback_t`	`void (*)(int fd, void* user_data)`	Called when FD is ready for writing
`timer_callback_t`	`void (*)(void* user_data)`	Called when timer expires
`task_callback_t`	`void (*)(void* user_data)`	Called for deferred execution

Async Scheduling Interface:

Method	Parameters	Returns	Description
`event_loop_register_read`	`loop struct event_loop*`, `fd int`, `callback read_callback_t`, `user_data void*`	`int`	Registers read interest, internal wrapper for `event_loop_register_fd`
`event_loop_register_write`	`loop struct event_loop*`, `fd int`, `callback write_callback_t`, `user_data void*`	`int`	Registers write interest (often done dynamically when buffer full)
`event_loop_deregister`	`loop struct event_loop*`, `fd int`, `events uint32_t`	`int`	Removes specific event interests
`event_loop_defer_task`	`loop struct event_loop*`, `callback task_callback_t`, `user_data void*`	`void`	Adds task to deferred queue for next loop iteration
`timer_wheel_add_repeating`	`wheel struct timer_wheel*`, `interval_ms uint64_t`, `callback timer_callback_t`, `user_data void*`	`void*`	Adds timer that automatically reschedules after firing

Algorithm: Event Loop with Async Scheduling (enhanced from Milestone 1):

1. Calculate `epoll_wait` timeout from `timer_wheel_next_expiration()`
2. Call `epoll_wait`
3. For each returned event:
    1. Extract `connection*` from `event.data.ptr`
    2. If `event.events & EPOLLIN`:
        1. Look up read callback for this FD
        2. **Execute immediately**: call `read_callback(fd, connection*)`
    3. If `event.events & EPOLLOUT`:
        1. Look up write callback for this FD
        2. **Execute immediately**: call `write_callback(fd, connection*)`
4. Call `timer_wheel_tick()`: each expired timer's callback is **added to deferred queue**
5. Process all tasks in deferred queue (including timer callbacks):
    1. While queue not empty:
        2. Pop task from head
        3. Execute `task_callback(user_data)`
        4. Free task node
6. Repeat

Repeating Timer Implementation:

- When a repeating timer fires, its callback executes
- Before returning, the callback (or timer infrastructure) calls `timer_wheel_add()` again with same interval
- Alternatively, `timer_wheel_add_repeating()` creates a special timer that auto-reschedules

Common Pitfalls for Milestone 3:

⚠️ **Pitfall: Callback modifies event registration during iteration**

- **Description**: A read callback calls `event_loop_deregister(fd, EPOLLIN)` while event loop is still iterating through the epoll events array.
- **Why it's wrong**: May cause invalid memory access if epoll modifies internal structures during iteration.
- **Fix**: Either defer registration changes (add to modification queue) or use immediate I/O callbacks but defer timer callbacks (our chosen approach).

⚠️ **Pitfall: Unbounded deferred task queue growth**

- **Description**: Tasks are added to deferred queue faster than they're processed.
- **Why it's wrong**: Memory exhaustion, increased latency.
- **Fix**: Implement queue size limit, or better, ensure tasks don't spawn infinite sub-tasks. Use circular buffer instead of linked list for bounded size.

⚠️ **Pitfall: Not cleaning up user_data on connection close**

- **Description**: Timer or deferred task holds `connection*` after connection freed.
- **Why it's wrong**: Use-after-free when callback eventually executes.
- **Fix**: When destroying connection, cancel all associated timers and scan deferred queue removing tasks with that user_data.

> **Key Insight**: The async scheduling layer **decouples event detection from event processing**. The event loop detects "what happened," while callbacks decide "what to do about it." This separation allows protocol handlers (like HTTP) to be built independently of the reactor machinery.

HTTP Server on Event Loop - Milestone 4

The HTTP server is a **protocol handler** built atop the event loop framework. Imagine a postal worker (event loop) who normally just forwards letters (bytes). Now they learn to recognize birthday invitations (HTTP requests): they check if the envelope has all required fields (complete headers), prepare a standard RSVP (HTTP response), and mail it back, all while continuing to handle other letters.

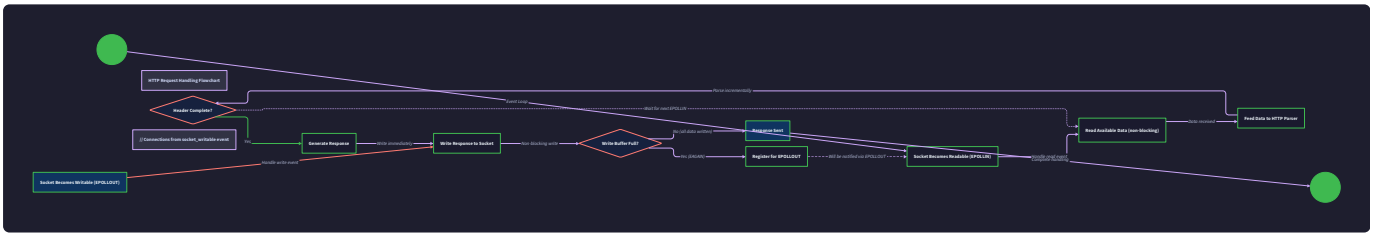
> **Decision: Incremental Parsing vs Buffering Whole Request**
>
> - **Context**: With non-blocking reads, an HTTP request may arrive in multiple fragments across different events. We need to parse efficiently.
> - **Options Considered**:
> - **Buffer whole request then parse**: Simple, but wastes memory on partial buffering and delays processing.
> - **Incremental/streaming parser**: Parse as data arrives, immediate processing of complete requests.
> - **State machine without full parsing**: Just find CRLF delimiters, minimal validation.
> - **Decision**: **Incremental parser with state machine**.
> - **Rationale**: Handles slow clients efficiently, minimizes memory usage, starts processing as soon as request is complete. Follows HTTP/1.1's line-oriented protocol naturally.
> - **Consequences**: More complex parser logic, must handle all partial states (mid-header, mid-chunk, etc.).

Option	Pros	Cons	Chosen?
Buffer whole request	Simple, robust, easy to validate	Memory overhead, latency on slow networks	No
Incremental parser	Efficient, immediate processing, scalable	Complex state machine, many edge cases	✅ Yes
Hybrid (buffer up to limit)	Balance of simplicity and efficiency	Still wastes memory on large requests	No

The HTTP server component implements the application logic. Its responsibilities include:

- Incremental parsing of HTTP/1.1 requests from non-blocking sockets
- Managing per-connection read/write buffers with flow control
- Generating appropriate HTTP responses
- Supporting keep-alive connections with configurable timeouts
- Handling backpressure when clients can't receive data quickly

Mental Model: The HTTP handler is like a coffee shop barista working during rush hour. Customers (clients) shout orders (request bytes) as they think of them. The barista writes each fragment on a notepad (read buffer). When they detect a complete order (double newline), they make the coffee (generate response) and pour it into a cup (write buffer). If the cup fills before the customer takes it, they set it aside and wait for the customer to empty their hands (EPOLLOUT).



Core Data Structures:

Name	Type	Description
`http_parser`	struct	Stateful incremental HTTP request parser
`http_request`	struct	Parsed request representation
`http_response`	struct	Response being built

`http_parser` Structure Fields:

Field	Type	Description
`state`	enum	Current parsing state (e.g., `START_LINE`, `HEADERS`, `BODY`, `COMPLETE`)
`buffer`	char*	Accumulated data for current parsing stage
`buffer_used`	int	Bytes currently in buffer
`content_length`	int	From Content-Length header, or -1 if chunked/unknown
`headers_complete`	int	Boolean flag indicating headers fully parsed
`current_header`	struct	Temporary storage for header being parsed

`connection` Structure Fields (HTTP extensions):

Field	Type	Description
`proto_data`	void*	Pointer to protocol-specific data (`http_parser*` for HTTP connections)
`state`	int	Extended with HTTP states: `HTTP_READING` , `HTTP_WRITING` , `HTTP_KEEPALIVE`

HTTP Server Interface:

Method	Parameters	Returns	Description
`http_connection_create`	`fd int`	`struct connection*`	Creates connection with HTTP parser initialized
`http_handle_read`	`conn struct connection*`	`int`	Read callback: reads data, feeds parser, generates response if request complete
`http_handle_write`	`conn struct connection*`	`int`	Write callback: sends response bytes from write buffer
`http_generate_response`	`conn struct connection*` , `request struct http_request*`	`void`	Generates appropriate HTTP response into connection's write buffer
`http_parser_feed`	`parser struct http_parser*` , `data char*` , `len int`	`int`	Incrementally parses data, returns bytes consumed

Algorithm: HTTP Read Callback:

1. Call `connection_read()` to read available data into read buffer
2. If bytes read > 0:
    1. Call `http_parser_feed()` with new data
    2. While parser indicates progress:
        1. If parser state becomes `HEADERS_COMPLETE` :
            1. Extract `http_request` from parser
            2. Call `http_generate_response()` to fill write buffer
            3. If write buffer has data, register for `EPOLLOUT` (if not already)
        2. If parser state becomes `ERROR` :
            1. Send 400 Bad Response, mark connection for closure
3. If bytes read == 0 (EOF):
    1. Initiate graceful closure
4. If bytes read == -1 with `EAGAIN` :
    1. Simply return (normal non-blocking condition)
5. Reset connection's idle timer (keep-alive)

Algorithm: HTTP Write Callback:

1. Call `connection_write()` to send data from write buffer
2. If all bytes sent:
    1. If keep-alive supported and requested:
        1. Reset parser for next request
        2. Deregister `EPOLLOUT` (if registered)
        3. Return to reading state
    2. Else:
        1. Initiate graceful closure

3. If partial bytes sent (write buffer still has data):
    1. Update `write_buffer_sent` offset
    2. Keep `EPOLLOUT` registered for next write opportunity
4. If write returns `EAGAIN`:
    1. Keep `EPOLLOUT` registered, return

Keep-Alive Implementation:

- Parser detects `Connection: keep-alive` header
- After response sent, instead of closing, reset parser state and buffer
- Maintain idle timer (e.g., 5 seconds); close connection if no new request arrives
- Limit number of requests per connection to prevent starvation

Buffer Management Strategy:

- Read buffer: circular buffer of `READ_BUFFER_SIZE` (4096). When full and more data arrives, either grow buffer or close connection (simplified: close).
- Write buffer: linear buffer with `write_buffer_sent` offset. When response exceeds buffer size, use write slicing: send first chunk, register `EPOLLOUT`, send next chunk when ready.
- Backpressure: When `connection_write()` returns `EAGAIN`, stop reading from that connection to prevent buffer buildup.

Common Pitfalls for Milestone 4:

⚠️ **Pitfall: HTTP request line split across reads**

- **Description**: "GET /index.html HTTP/1.1\r\n" arrives as "GET /index" then ".html HTTP/1.1\r\n".
- **Why it's wrong**: Naive parser expecting complete lines fails.
- **Fix**: Incremental parser stores partial data in buffer, continues when more arrives.

⚠️ **Pitfall: Write buffer exhaustion without EPOLLOUT registration**

- **Description**: Response too large for buffer, but code doesn't register for `EPOLLOUT` to continue later.
- **Why it's wrong**: Response truncated, connection hangs.
- **Fix**: After filling write buffer, if more data remains, register `EPOLLOUT`. In write callback, send next chunk.

⚠️ **Pitfall: Keep-alive connection starvation**

- **Description**: Single fast client sends infinite requests on keep-alive, blocking others.
- **Why it's wrong**: Fairness issue, denial of service vector.
- **Fix**: Limit requests per connection (e.g., 100), then close. Or implement weighted fair queuing.

> **Key Insight**: The HTTP server demonstrates **composability** of the event loop framework. By implementing just three callbacks (create, read, write) and using the provided buffer management and timer facilities, we get a production-capable server. This pattern extends to any protocol (WebSocket, Redis, custom).

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Event Loop	Level-triggered epoll	Edge-triggered epoll with EPOLLONESHOT
Timer Wheel	4-level hierarchical (64ms, 4s, 4min, 4hr)	Hashed timing wheel with drift correction
Async Scheduling	Single deferred task queue	Priority queue with multiple urgency levels
HTTP Parser	State machine scanning for `\r\n`	Ragel state machine generator or `llhttp` library

B. Recommended File/Module Structure:

```
event_loop_project/
├── src/
│   ├── main.c                    # Entry point, server initialization
│   ├── event_loop/
│   │   ├── event_loop.c          # Core event loop (Milestone 1)
│   │   ├── event_loop.h
│   │   ├── connection.c          # Connection management
│   │   └── connection.h
│   ├── timer/
│   │   ├── timer_wheel.c         # Hierarchical timer wheel (Milestone 2)
│   │   └── timer_wheel.h
│   ├── async/
│   │   ├── scheduler.c           # Callback registration (Milestone 3)
│   │   └── scheduler.h
│   └── protocols/
│       ├── http/
│       │   ├── http_server.c     # HTTP server implementation (Milestone 4)
│       │   ├── http_server.h
│       │   ├── http_parser.c     # Incremental HTTP parser
│       │   └── http_parser.h
│       └── echo.c                # Simple echo protocol for testing
├── tests/
│   ├── test_timer_wheel.c
│   └── test_http_parser.c
└── benchmarks/
    └── c10k_benchmark.c          # 10K concurrent connection test
```

C. Infrastructure Starter Code (non-core components):

`src/util/nonblocking.c`:

```c
#include <fcntl.h>

#include <unistd.h>

#include <errno.h>


int set_nonblocking(int fd) {

    int flags = fcntl(fd, F_GETFL, 0);

    if (flags == -1) return -1;

    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);

}


int create_listening_socket(int port, int backlog) {

    int fd = socket(AF_INET6, SOCK_STREAM | SOCK_NONBLOCK, 0);

    if (fd == -1) return -1;



    // Allow both IPv4 and IPv6

    int opt = 1;

    setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));



    struct sockaddr_in6 addr = {0};

    addr.sin6_family = AF_INET6;

    addr.sin6_port = htons(port);

    addr.sin6_addr = in6addr_any;  // Listen on all interfaces



    if (bind(fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {

        close(fd);

        return -1;

    }



    if (listen(fd, backlog) == -1) {

        close(fd);

        return -1;

    }



    return fd;
```

```
}
```

D. Core Logic Skeleton Code:

`src/event_loop/event_loop.c` (Milestone 1):

```c
struct event_loop* event_loop_create(int max_events) {
    // TODO 1: Allocate memory for event_loop struct
    // TODO 2: Call epoll_create1(EPOLL_CLOEXEC) for epoll_fd
    // TODO 3: Allocate events array of size max_events
    // TODO 4: Initialize running = 1, timer_wheel = NULL, task_queue = NULL
    // TODO 5: Return pointer or NULL on error
}

int event_loop_register_fd(struct event_loop* loop, int fd,
                           uint32_t events, void* user_data) {
    // TODO 1: Create epoll_event struct, set .events and .data.ptr = user_data
    // TODO 2: Call epoll_ctl(EPOLL_CTL_ADD, fd, &event)
    // TODO 3: Return 0 on success, -1 on error (check errno)
}

int event_loop_run(struct event_loop* loop, int timeout_ms) {
    // TODO 1: While loop->running is true:
    // TODO 2:   Calculate actual timeout: min(timeout_ms, timer_wheel_next_expiration())
    // TODO 3:   Call epoll_wait(loop->epoll_fd, loop->events, loop->max_events, actual_timeout)
    // TODO 4:   For i from 0 to nfds-1:
    // TODO 5:     Extract connection* from loop->events[i].data.ptr
    // TODO 6:     If loop->events[i].events & EPOLLIN: call connection_read(conn)
    // TODO 7:     If loop->events[i].events & EPOLLOUT: call connection_write(conn)
    // TODO 8:   If loop->timer_wheel: call timer_wheel_tick(loop->timer_wheel)
    // TODO 9:   Process all tasks in loop->task_queue
    // TODO 10: Return 0 when loop exits
}
```

`src/timer/timer_wheel.c` (Milestone 2):

```c
void* timer_wheel_add(struct timer_wheel* wheel, uint64_t timeout_ms,
                      timer_callback_t callback, void* user_data) {
    // TODO 1: Calculate absolute expiry: wheel->current_time + (timeout_ms / wheel->resolution_ms)
    // TODO 2: Determine level: find smallest level where timeout_ms < (slots_per_level^(level+1) * resolution_ms)
    // TODO 3: Calculate slot index: (current_time + timeout_ms) >> (level * log2(slots_per_level)) % slots_per_level
    // TODO 4: Allocate timer struct, set fields
    // TODO 5: Insert timer into linked list at wheel->wheel[level][slot]
    // TODO 6: Store (level, slot, list_position) in opaque reference for O(1) removal
    // TODO 7: Increment wheel->count
    // TODO 8: Return opaque reference
}

uint64_t timer_wheel_next_expiration(struct timer_wheel* wheel) {
    // TODO 1: If wheel->count == 0, return UINT64_MAX (no timers)
    // TODO 2: For each level from 0 to levels-1:
    // TODO 3:   For each slot from current_slot to end (circular):
    // TODO 4:     If wheel->wheel[level][slot] is non-empty:
    // TODO 5:       Calculate absolute time of this slot
    // TODO 6:       Return max(0, (slot_time - current_time) * resolution_ms)
    // TODO 7: Return UINT64_MAX (should not reach here if count > 0)
}
```

`src/async/scheduler.c` (Milestone 3):

```c
void event_loop_defer_task(struct event_loop* loop,
                           task_callback_t callback, void* user_data) {
    // TODO 1: Allocate deferred_task struct
    // TODO 2: Set callback and user_data fields
    // TODO 3: Insert at tail of loop->task_queue (maintain head/tail pointers)
    // TODO 4: If this is first task, no special handling needed
}

static void process_deferred_tasks(struct event_loop* loop) {
    // TODO 1: While loop->task_queue not empty:
    // TODO 2:   Take task from head of queue
    // TODO 3:   Execute task->callback(task->user_data)
    // TODO 4:   Free task struct
    // TODO 5:   Continue to next task
    // TODO 6: Reset queue head/tail to NULL
}
```

`src/protocols/http/http_parser.c` (Milestone 4):

```c
int http_parser_feed(struct http_parser* parser, char* data, int len) {
    // TODO 1: Based on parser->state:
    // TODO 2:   STATE_START_LINE: Look for '\r\n', parse method, URI, version
    // TODO 3:   STATE_HEADERS: Look for '\r\n', parse "Key: Value", empty line ends headers
    // TODO 4:   STATE_BODY: If content_length > 0, accumulate exactly that many bytes
    // TODO 5:   STATE_CHUNKED: Parse chunked encoding (advanced)
    // TODO 6: Update parser->buffer with consumed data
    // TODO 7: Return number of bytes consumed (0..len)
    // TODO 8: Transition states when appropriate
    // TODO 9: Set parser->headers_complete when empty line found
}
```

E. Language-Specific Hints (C):

- Use `epoll_create1(EPOLL_CLOEXEC)` instead of `epoll_create()` for proper FD cleanup on exec
- Always check `errno` after system call failures; `EINTR` handling is optional for this project
- For timer wheel modulo operations: use `slot = time & (slots_per_level - 1)` when slots is power of two
- Store `connection*` in `epoll_event.data.ptr`, not `epoll_event.data.fd`, to avoid separate lookup
- Use `accept4(fd, NULL, NULL, SOCK_NONBLOCK)` if available to accept with non-blocking already set

F. Milestone Checkpoint Verification:

Milestone 1:

```
$ ./server -p 8080 &
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello
^]
telnet> quit
Connection closed.
```

- **Expected**: Server echoes each line. Multiple concurrent telnet sessions work.
- **Failure signs**: Single connection blocks others; server hangs on `accept()`.

Milestone 2:

```
$ ./server -p 8080 --timeout 5 &
$ time (echo "test"; sleep 6; echo "test2") | nc localhost 8080
```

- **Expected**: First "test" echoed, connection closed after ~5 seconds, second "test2" not delivered.
- **Failure signs**: Connection stays open indefinitely; timeout fires immediately.

Milestone 3:

```
$ ./server -p 8080 &
$ curl -v http://localhost:8080/
```

- **Expected**: Returns simple HTTP response (maybe 404). Check logs for callback execution order.
- **Failure signs**: No response; server crashes; callbacks not firing.

Milestone 4:

```
$ ./server -p 8080 &
$ ab -c 100 -n 10000 http://localhost:8080/
```

- **Expected**: Completes 10K requests with < 1% failures, memory stable.
- **Failure signs**: Connection resets; request timeouts; memory grows unbounded.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
CPU at 100% with no connections	`epoll_wait` timeout = 0 (non-blocking)	Check timeout calculation logic	Use proper timeout from timer wheel
Connections stuck in `TIME_WAIT`	Not closing sockets properly on timeout	Use `netstat -tn` or `ss -t`	Ensure `close(fd)` called, check all code paths
Timer fires 64ms late	Wheel resolution too coarse	Log timer add vs fire times	Reduce resolution to 10ms or use higher-precision `clock_gettime()`
HTTP response truncated	Write buffer full, no EPOLLOUT registration	Log write buffer sizes	Register for EPOLLOUT when buffer has pending data
Memory leak with keep-alive	Parser state not reset between requests	Check `http_parser_feed()` state transitions	Fully reset parser after each request
Some clients starved	No round-robin in event processing	Log which FDs get serviced each iteration	Process only limited events per FD per tick, or use fairness queue

Interactions and Data Flow

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

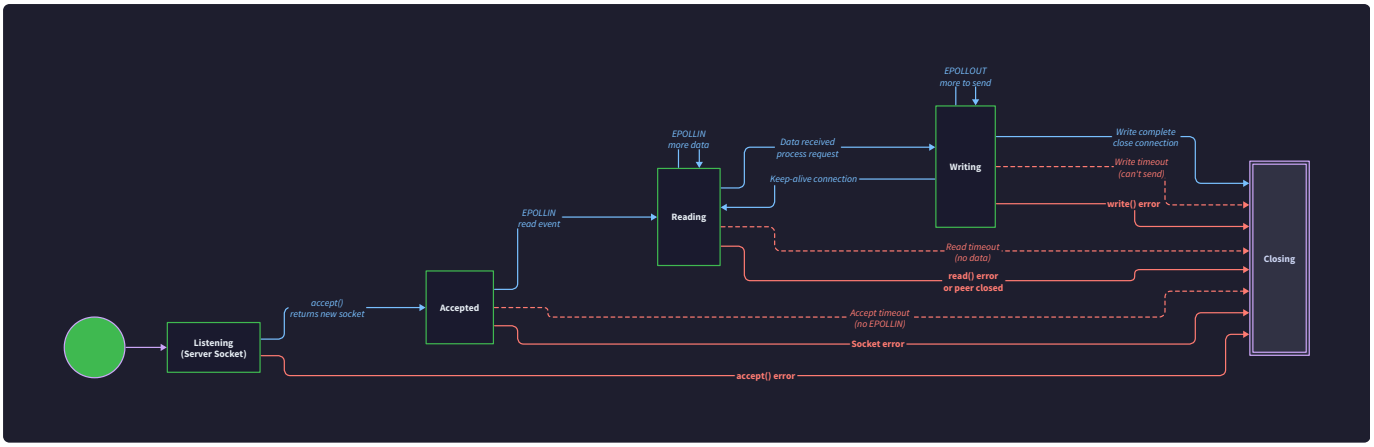
This section details the **dynamic behavior** of the event loop system. While the previous sections described the static components and data structures, here we examine how these parts interact over time to handle real-world scenarios. Think of this as watching the system in motion—like observing a complex clock mechanism where gears (components) turn in precise sequence to produce the desired outcome (handling 10K+ connections).

Understanding these flows is critical because the reactor pattern's power emerges from the choreography between I/O events, timer ticks, and deferred tasks. A single misstep—like processing timers before I/O events—can cause subtle bugs like delayed responses or missed timeouts.

Sequence: Connection Lifecycle

Imagine each TCP connection as a **visitor to a high-traffic museum**. The event loop is the single tour guide managing all visitors simultaneously. The guide doesn't wait for one visitor to finish examining an exhibit before attending to another. Instead, they constantly scan the room (epoll_wait), notice when visitors are ready (read/write events), provide information in small chunks (non-blocking I/O), and politely escort out visitors who've been idle too long (timer timeouts). This analogy captures the essence of the connection lifecycle in our event-driven architecture.

The complete journey of a connection follows a **finite state machine** (refer to



). Below, we walk through each state transition with concrete steps, data structure updates, and the API calls involved.

Architecture Decision Record: Connection State Management

> **Decision: State Machine Embedded in Connection Structure**
>
> - **Context**: We need to track where each connection is in its lifecycle (accepting, reading, writing, closing) to apply the correct handling logic when events occur. The state determines which callbacks to invoke and what cleanup to perform.
> - **Options Considered**:
>   1. **Separate state machines per protocol**: Each protocol handler (HTTP, echo) maintains its own state enumeration, duplicating common states like `READING` and `WRITING`.
>   2. **Generic state with protocol-specific extensions**: A base connection state machine (`STATE_ACCEPTED`, `STATE_READING`, etc.) with a `proto_data` field for protocol-specific state (like `HTTP_READING`).
>   3. **Fully protocol-agnostic event callbacks**: No explicit state tracking; callbacks decide what to do based on buffered data and previous actions, requiring complex logic within each callback.
> - **Decision**: Option 2—generic state machine with protocol extension. The `connection` struct contains a `state` field with common states, and protocol handlers can store additional state in `proto_data`.
> - **Rationale**: This provides a clean separation of concerns. The event loop manages the generic lifecycle (allocation, registration, timeout, cleanup) while protocol handlers manage protocol-specific parsing and logic. It avoids duplication while maintaining clear state transitions that are easy to debug.
> - **Consequences**: Protocol handlers must coordinate with the base state (e.g., transitioning from `STATE_READING` to `STATE_WRITING` when a request is fully parsed). The `proto_data` field requires careful memory management (allocated/freed with the connection).

Option	Pros	Cons	Chosen?
Separate state machines per protocol	Clean protocol separation; no shared state assumptions	Duplication of common states; harder to enforce consistent lifecycle management	❌
Generic state with protocol extensions	**Unified lifecycle management; extensible; clear debugging**	**Slight complexity in protocol handler coordination**	✅
Fully protocol-agnostic callbacks	Maximum flexibility; no state tracking overhead	Callbacks become complex; hard to reason about connection progress	❌

The connection lifecycle proceeds through the following stages, each triggered by specific events and resulting in specific actions:

Table: Connection State Transitions and Actions

Current State	Event Trigger	Next State	Actions Taken (in order)
LISTENING (listening socket)	`EPOLLIN` event on listening socket	`STATE_ACCEPTED`	1. `accept4()` with `SOCK_NONBLOCK` 2. Create `connection` struct via `connection_create()` 3. Register new socket with `EPOLLIN` via `event_loop_register_fd()` 4. Add idle timer via `timer_wheel_add()` (reference stored in `connection.timer_ref`)
`STATE_ACCEPTED`	`EPOLLIN` event on connection socket	`STATE_READING` (or `HTTP_READING`)	1. Cancel previous idle timer (if any) via `timer_wheel_remove()` 2. Invoke protocol-specific read callback (e.g., `http_handle_read()`) 3. Callback performs `connection_read()`, processes data, may change state
`STATE_READING`	Read callback completes request processing	`STATE_WRITING` (or `HTTP_WRITING`)	1. Generate response (e.g., `http_generate_response()`) into `connection.write_buffer` 2. If `write_buffer` has data, register for `EPOLLOUT` via `event_loop_modify_fd()` 3. Add new idle timer for write phase (optional, different timeout)
`STATE_WRITING`	`EPOLLOUT` event on connection socket	`STATE_WRITING` (remain until buffer empty) or `STATE_READING` (keep-alive)	1. Invoke write callback (e.g., `http_handle_write()`) 2. Callback performs `connection_write()`, sends buffered data 3. If all data sent: - For HTTP with keep-alive: deregister `EPOLLOUT`, return to `STATE_READING`, restart idle timer - Otherwise: transition to `STATE_CLOSING`
`STATE_WRITING`	Write buffer full (cannot send all data)	`STATE_WRITING` (stays)	1. `connection_write()` returns `EAGAIN` 2. Write callback leaves data in `write_buffer`, ensures `EPOLLOUT` remains registered 3. Next `EPOLLOUT` event will resume sending
Any active state	Timer expiration (idle timeout)	`STATE_CLOSING`	1. Timer callback receives `connection` as `user_data` 2. Calls `event_loop_unregister_fd()` to remove from epoll 3. Calls `connection_destroy()` which closes socket and frees buffers
Any active state	Read returns 0 bytes (client closed)	`STATE_CLOSING`	1. Read callback detects EOF, transitions to `STATE_CLOSING` 2. `event_loop_unregister_fd()` and `connection_destroy()`
Any active state	Write error (e.g., `EPIPE`, `ECONNRESET`)	`STATE_CLOSING`	1. Write callback checks error code, transitions to `STATE_CLOSING` 2. Cleanup as above

Current State	Event Trigger	Next State	Actions Taken (in order)
`STATE_CLOSING`	Cleanup completed	(destroyed)	1. All resources freed, connection removed from all data structures

Let's walk through a concrete example of an HTTP connection with keep-alive:

Concrete Walk-Through: HTTP Keep-Alive Connection

1. **Client connects**: The listening socket (port 8080) receives an `EPOLLIN` event. The event loop's dispatch logic calls the accept callback.
   - `accept4()` returns a new socket fd (e.g., 42) with `SOCK_NONBLOCK`.
   - `connection_create(42)` allocates a `connection` struct, initializes buffers, sets `state = STATE_ACCEPTED`.
   - `event_loop_register_fd(loop, 42, EPOLLIN, connection_ptr)` adds the socket to epoll monitoring.
   - `timer_wheel_add(wheel, 30000, idle_timeout_callback, connection_ptr)` sets a 30-second idle timeout, storing the timer reference in `connection.timer_ref`.
2. **Client sends HTTP request**: Socket 42 becomes readable (`EPOLLIN`). The event loop invokes the registered read callback, which is `http_handle_read()`.
   - The callback first cancels the idle timer: `timer_wheel_remove(wheel, connection->timer_ref)`.
   - It calls `connection_read(connection)` which reads up to `READ_BUFFER_SIZE` bytes into `read_buffer`. Suppose the request is partial (only "GET /index.html HT").
   - `http_parser_feed()` consumes the available bytes but doesn't complete the request (headers not fully received).
   - The callback updates `connection->state = HTTP_READING` (a protocol-specific state stored in `proto_data`).
   - It adds a new idle timer (maybe 5 seconds for reading phase) via `timer_wheel_add()`.
3. **More data arrives**: Another `EPOLLIN` event occurs on socket 42. `http_handle_read()` is called again.
   - Timer is cancelled again (idle timeout reset).
   - `connection_read()` reads the remainder of the request ("TP/1.1\r\nHost: ...").
   - `http_parser_feed()` now completes header parsing, sets `headers_complete = 1`, and extracts `Content-Length`.
   - The parser may need more data for body (if `Content-Length > 0`). In this example, it's a GET request with no body, so parsing completes.
   - `http_generate_response()` generates an HTTP 200 response with some HTML content, copying it into `connection.write_buffer`.
   - The callback changes `connection->state = HTTP_WRITING`.
   - It registers for `EPOLLOUT` via `event_loop_modify_fd(loop, 42, EPOLLOUT, connection_ptr)` to be notified when writable.
   - A new idle timer for the writing phase is added (maybe 10 seconds).
4. **Socket becomes writable**: `EPOLLOUT` event fires on socket 42. The event loop invokes the write callback `http_handle_write()`.
   - `connection_write(connection)` sends as much of `write_buffer` as possible. Suppose the entire response fits in one write (common for small responses).
   - All bytes are sent, so `write_buffer` is cleared.
   - Since HTTP keep-alive is supported and requested, the connection should wait for the next request.
   - The callback deregisters `EPOLLOUT` (only `EPOLLIN` needed now) via `event_loop_modify_fd(loop, 42, EPOLLIN, connection_ptr)`.
   - `connection->state` is set back to `HTTP_READING` (or `HTTP_KEEPALIVE`).
   - The idle timer is reset to the keep-alive timeout (maybe 30 seconds).

5. **Client sends second request**: The cycle repeats from step 2.
6. **Client idle timeout**: Suppose after the response, the client disappears and sends no more data. After 30 seconds, the timer added in step 4 expires.
    - The timer callback (`idle_timeout_callback`) receives the `connection` pointer.
    - It calls `event_loop_unregister_fd(loop, connection->fd)` to remove the socket from epoll.
    - It calls `connection_destroy(connection)` , which `close()` s the socket and frees all buffers and the `connection` struct itself.

Common Pitfalls in Connection Lifecycle Management

⚠️ Pitfall: Timer reference dangling after connection destruction

- **Description**: A connection is destroyed (due to error or normal close) but its timer still exists in the timer wheel. When the timer later expires, it calls the callback with a stale `user_data` pointer, causing a use-after-free crash.
- **Why it's wrong**: Timer callbacks must never access freed memory. This leads to undefined behavior and security vulnerabilities.
- **Fix**: Always cancel the timer (`timer_wheel_remove()`) **before** freeing the connection. In the destruction sequence: 1) remove from epoll, 2) cancel timer, 3) close socket and free memory. Alternatively, store a weak reference in the timer that's validated before use.

⚠️ Pitfall: State inconsistency during partial writes

- **Description**: When a write cannot send all buffered data (`EAGAIN`), the code transitions to `STATE_WRITING` and registers `EPOLLOUT` . However, if an error occurs before the next `EPOLLOUT` event (like client disconnect), the connection might be stuck in `STATE_WRITING` with no way to progress.
- **Why it's wrong**: The connection becomes a zombie—consuming resources but unable to complete.
- **Fix**: Always check for errors on the socket (via `getsockopt(SO_ERROR)` or checking `errno` after a subsequent operation) even when waiting for `EPOLLOUT` . Additionally, implement a write timeout timer that forces closure if data cannot be sent within a reasonable period.

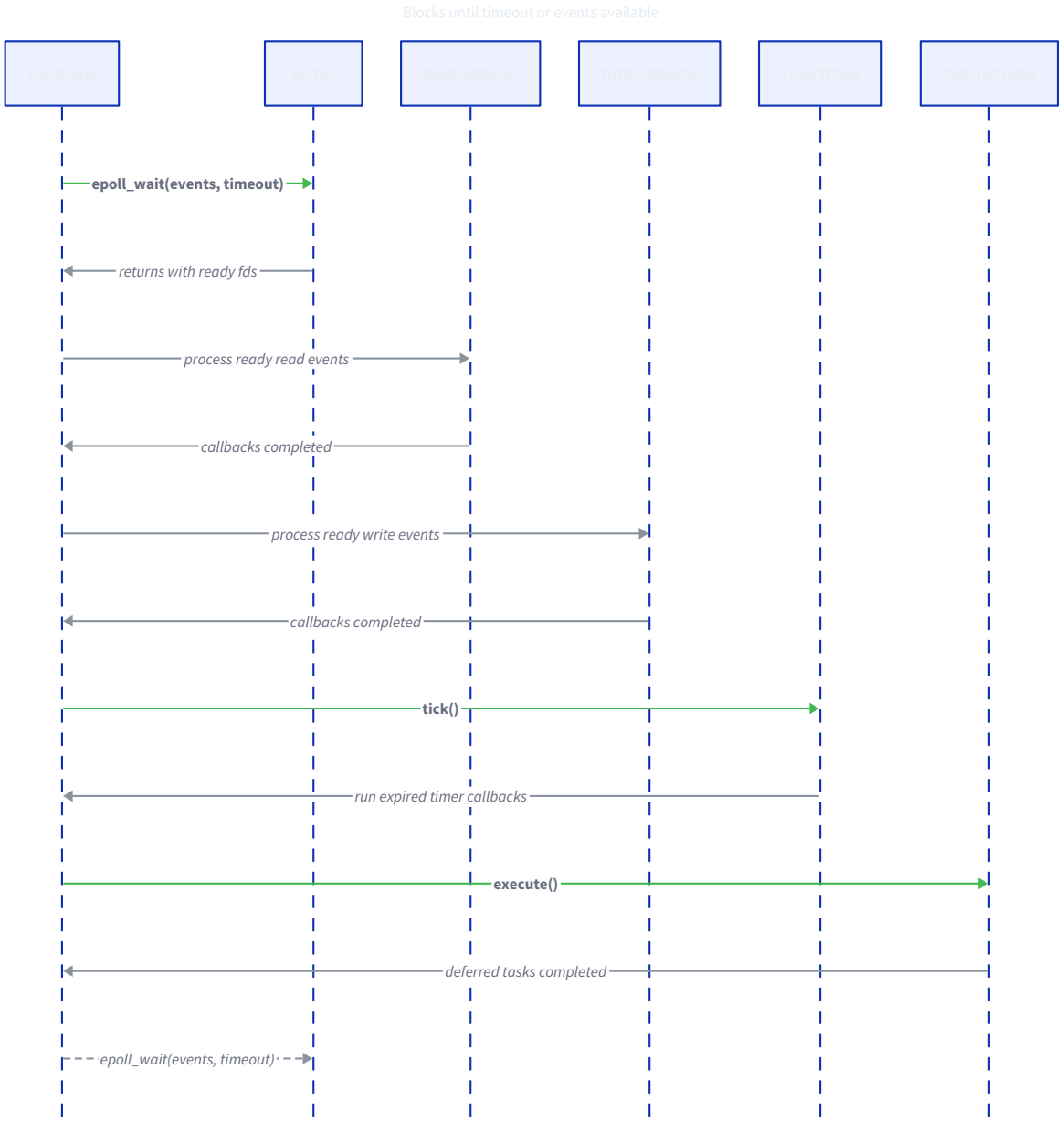
⚠️ Pitfall: Busy-loop on `EPOLLOUT` with empty buffer

- **Description**: After all buffered data is sent, the code forgets to deregister `EPOLLOUT` . The socket remains continuously writable (level-triggered), causing the event loop to spin constantly invoking the write callback.
- **Why it's wrong**: Wastes CPU cycles, reduces performance, and may cause unexpected behavior if the write callback is called with empty buffer.
- **Fix**: Immediately after `connection_write()` sends all data, check if `write_buffer_used == 0` . If so, modify the event registration to remove `EPOLLOUT` (using `event_loop_modify_fd()` with only `EPOLLIN`). Alternatively, use edge-triggered mode (`EPOLLET`) which only notifies on transition to writable, but this requires different handling.

Flow: Event Loop Tick

The event loop tick is the **heartbeat** of the entire system. Imagine a master chef in a busy kitchen who repeatedly follows a strict routine: 1) Check which orders are ready (epoll_wait), 2) Prepare dishes that have ingredients available (process I/O events), 3) Check the oven timer (timer wheel tick), 4) Clean up and prep for next round (deferred tasks). This disciplined, repetitive cycle ensures no order is forgotten and everything progresses efficiently.

Each iteration of the event loop—called a "tick"—processes all pending I/O events, expires due timers, and executes deferred tasks. The order of these operations is crucial for correctness and performance. Refer to



for a visual sequence.

Architecture Decision Record: Processing Order Within a Tick

Decision: I/O → Timers → Deferred Tasks

- **Context**: We must decide the sequence in which to handle different event sources within a single loop iteration. The order affects latency, fairness, and potential starvation.
- **Options Considered**:
  1. **Timers → I/O → Tasks**: Process timer callbacks first, then I/O events, then deferred tasks.
  2. **I/O → Timers → Tasks**: Process all I/O events (reads then writes), then timer expirations, then deferred tasks.
  3. **Mixed priority queue**: All events (I/O, timer, task) placed in a single priority queue sorted by urgency.
- **Decision**: Option 2—I/O first, then timers, then deferred tasks.
- **Rationale**: I/O events are typically the highest priority because they represent data already arrived that needs immediate processing to free kernel buffers. Timers are checked after I/O because their expiration time has already passed slightly (we check after epoll_wait returns). Deferred tasks run last because they're explicitly deferred to the next iteration and shouldn't block I/O processing. This order minimizes latency for incoming data while ensuring timers are checked at least every `epoll_wait` timeout.
- **Consequences**: Timer callbacks may experience a small additional latency (up to the `epoll_wait` timeout) beyond their scheduled time. However, this is acceptable for idle timeouts and heartbeat timers. Critical timing applications would need a different design.

Option	Pros	Cons	Chosen?
Timers → I/O → Tasks	Timers fire more precisely at expiration	I/O data may sit in kernel buffers longer, increasing memory pressure and latency	❌
I/O → Timers → Tasks	**Minimizes latency for incoming data; logical flow**	**Timer callbacks slightly delayed**	✅
Mixed priority queue	Most theoretically fair; all events handled by urgency	Complex implementation; overhead of sorting; may starve I/O if many timers expire	❌

The event loop tick follows a precise algorithm executed by `event_loop_run()`. Below is the step-by-step procedure for a single iteration:

Algorithm: Event Loop Tick (single iteration of `event_loop_run`)

1. **Calculate timeout for epoll_wait**:
   - Call `timer_wheel_next_expiration(timer_wheel)` to get milliseconds until the next timer expires.
   - If no timers pending, set `timeout_ms = -1` (infinite wait).
   - If timers pending, set `timeout_ms = min(next_expiration, MAX_EPOLL_TIMEOUT)` where `MAX_EPOLL_TIMEOUT` is a safety cap (e.g., 1000ms) to prevent hanging indefinitely.
2. **Wait for I/O events**:
   - Call `epoll_wait(loop->epoll_fd, loop->events, loop->max_events, timeout_ms)`.
   - This blocks the thread until either:
     - At least one file descriptor has a pending event (readable, writable, error).
     - The timeout expires (no I/O activity before next timer deadline).
     - A signal interrupts the call (handled by checking `errno == EINTR`).
   - Returns number of ready events `n_events`.
3. **Process I/O events**:
   - For `i = 0` to `n_events - 1`:
     - Retrieve `epoll_event` from `loop->events[i]`.

    - Extract `user_data` pointer (typically a `connection*`) from `epoll_event.data.ptr`.
    - Check `epoll_event.events` for error flags (`EPOLLERR`, `EPOLLHUP`). If present, transition connection to `STATE_CLOSING` and continue to next event.
    - If `epoll_event.events & EPOLLIN`: Invoke the connection's read callback (e.g., `http_handle_read`). The callback handles non-blocking read and state transitions.
    - If `epoll_event.events & EPOLLOUT`: Invoke the connection's write callback (e.g., `http_handle_write`). The callback handles non-blocking write and may deregister `EPOLLOUT` when done.
    - Note: A single event can have both `EPOLLIN` and `EPOLLOUT` set (socket both readable and writable). Process both flags.
4. **Tick the timer wheel**:
    - Call `timer_wheel_tick(timer_wheel)`. This function:
        - Increments `current_time` by the elapsed time (may use a monotonic clock or simply assume one tick occurred).
        - Scans the appropriate wheel slots for expired timers.
        - For each expired timer, invokes its callback with `user_data`.
        - Handles cascading of future timers from higher to lower wheel levels.
5. **Process deferred task queue**:
    - While `loop->task_queue` is not empty:
        - Dequeue the first `deferred_task`.
        - Execute its `callback(task->user_data)`.
        - Free the task node.
    - Note: Tasks enqueued during this processing (by callbacks) are executed on the **next** iteration to avoid infinite loops.
6. **Housekeeping and loop condition**:
    - Check if `loop->running == 0` (set by `event_loop_stop()`). If so, break out of the loop.
    - Optionally, perform periodic maintenance (e.g., log statistics, resize internal buffers if needed) every N iterations.
    - Return to step 1 for the next iteration.

Timing and Latency Considerations

The choice of `timeout_ms` in step 1 is critical for balancing responsiveness and CPU usage:

- If `timeout_ms` is too large (e.g., always `-1`), timer expirations will be delayed until the next I/O event occurs, causing idle connections to linger beyond their timeout.
- If `timeout_ms` is too small (e.g., always `1`), the loop will busy-wait, consuming 100% CPU even when idle.
- The correct approach: dynamically set `timeout_ms` to the time until the next timer expiration, capped at a reasonable maximum (e.g., 100ms) to allow periodic housekeeping even when no timers are scheduled.

Concrete Example: Tick with Mixed Events

Let's walk through a concrete tick where multiple event types occur:

1. **Before tick**:
    - Timer wheel has a timer expiring in 50ms (idle timeout for connection A).
    - Connection B has pending data in its kernel receive buffer.
    - Deferred task queue has one task (cleanup of a recently closed connection).
2. **Step 1**: Calculate timeout. `timer_wheel_next_expiration()` returns 50. `timeout_ms = 50`.
3. **Step 2**: `epoll_wait` with 50ms timeout. Suppose after 30ms, connection B becomes readable. `epoll_wait` returns with `n_events = 1`.

4. **Step 3**: Process I/O events. The event is for connection B with `EPOLLIN`. Invoke `http_handle_read(connection_B)`. It reads data, parses a complete HTTP request, generates response, and registers `EPOLLOUT`. No errors.
5. **Step 4**: Tick timer wheel. `timer_wheel_tick()` increments time by 30ms (the actual wait). The timer that was 50ms away is now 20ms from expiration, so it does NOT fire yet. No callbacks invoked.
6. **Step 5**: Process deferred tasks. The cleanup task callback frees memory from a connection closed in the previous tick.
7. **Step 6**: Loop continues (`running == 1`).

Next tick:

- `timeout_ms` now 20ms (timer for connection A).
- `epoll_wait` waits up to 20ms. No I/O events occur.
- After 20ms, `epoll_wait` times out (returns 0 events).
- Process I/O events: none.
- `timer_wheel_tick()` increments time by 20ms, timer for connection A expires. Timer callback calls `connection_destroy(connection_A)`.
- Process deferred tasks: none.
- Loop continues.

This example shows how the event loop interleaves different event types, ensuring timely processing of both I/O and timers.

Data Flow During a Tick

The following table summarizes the data transformations that occur during each phase of the tick:

Phase	Input Data	Transformation	Output Data
`epoll_wait`	Kernel I/O readiness state	System call blocks until timeout or readiness	Array of `epoll_event` structures with `fd` and event flags
I/O Processing	`epoll_event` array + `connection` pointers	Callbacks read/write socket buffers, update connection state, modify buffers	Updated `connection` buffers, possibly new event registrations (`EPOLLOUT`), state transitions
Timer Tick	`timer_wheel` state + current time	Scan for expired timers, invoke callbacks, cascade future timers	Timer callbacks executed (may close connections, send heartbeats), `timer_wheel` slots updated
Task Processing	`deferred_task` linked list	Execute callbacks in FIFO order	Side effects (cleanup, logging), queue emptied

Common Pitfalls in Event Loop Tick Implementation

⚠️ Pitfall: Starving timers or tasks due to infinite I/O

- **Description**: If a continuous stream of I/O events occurs (e.g., a fast client sending data), the loop may never progress beyond step 3 because `epoll_wait` always returns immediately with events. Timers never get checked, and deferred tasks never run.
- **Why it's wrong**: Timers don't fire, causing idle connections to never timeout. Deferred tasks (like cleanup) accumulate, causing memory leaks.
- **Fix**: Ensure `epoll_wait` has a non-zero timeout even when I/O is busy. Cap the number of I/O events processed per tick (e.g., process at most `2 * max_events` per tick before breaking to timer/task phases). Alternatively, use a separate mechanism to force timer checks, like a periodic signal or checking monotonic clock between iterations.

⚠️ Pitfall: Callback modifying event set during iteration

- **Description**: While iterating through the `epoll_event` array, a callback calls `event_loop_modify_fd()` or `event_loop_unregister_fd()` for the same fd being processed. This may corrupt the event array or cause undefined behavior.

- **Why it's wrong**: Modifying the underlying epoll set while iterating over the result of `epoll_wait` is safe in Linux (the returned array is a snapshot), but modifying the **same fd's events** can lead to confusion about which events were actually present.
- **Fix**: Defer modifications until after the I/O processing loop. For example, when a read callback completes a request and needs to register `EPOLLOUT`, set a flag in the connection and perform the modification after all events are processed. Alternatively, use the `EPOLLONESHOT` flag to automatically deregister after an event, then re-register with desired events.

⚠️ **Pitfall: Incorrect timeout calculation leading to drift**

- **Description**: Using a simple `sleep` or assuming each tick takes exactly `timeout_ms` milliseconds causes timer drift over time. Real-world `epoll_wait` may return early due to signals or I/O, and processing takes variable time.
- **Why it's wrong**: Timers become increasingly inaccurate; a 30-second idle timeout might fire after 35 seconds.
- **Fix**: Base time on a monotonic clock (`clock_gettime(CLOCK_MONOTONIC)`) rather than assuming elapsed time. Store the clock value before `epoll_wait` and after, compute actual elapsed milliseconds, and advance the timer wheel by that amount. This ensures timers expire at correct absolute times regardless of processing overhead.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Time Measurement	`gettimeofday()` (microsecond resolution)	`clock_gettime(CLOCK_MONOTONIC)` (nanosecond, not affected by system time changes)
Epoll Event Loop	Level-triggered (default) with manual `EPOLLOUT` registration	Edge-triggered (`EPOLLET`) with careful buffer exhaustion checks
Timer Wheel Resolution	Fixed 1ms ticks (requires high-frequency `epoll_wait` timeout)	Configurable resolution (e.g., 10ms) with cascading levels
Buffer Management	Fixed-size per-connection buffers (4KB)	Dynamic growth with upper limit; buffer pooling for reuse

Recommended File/Module Structure

```
event-loop-epoll/
├── include/
│   ├── event_loop.h      # Core event loop structures and API
│   ├── timer_wheel.h     # Timer wheel API
│   ├── connection.h      # Connection structure and utilities
│   └── http_handler.h    # HTTP-specific callbacks and parser
├── src/
│   ├── main.c            # Entry point, creates listening socket, starts loop
│   ├── event_loop.c      # Implementation of event_loop_* functions
│   ├── timer_wheel.c     # Hierarchical timer wheel implementation
│   ├── connection.c      # Connection creation, destruction, I/O helpers
│   └── http_handler.c    # HTTP parser, response generation
├── tests/
│   ├── test_timer_wheel.c
│   └── test_http_parser.c
└── Makefile
```

Infrastructure Starter Code

Here is complete, ready-to-use code for a monotonic time utility (place in `src/time_util.c` and `include/time_util.h`):

```c
/* include/time_util.h */
#ifndef TIME_UTIL_H
#define TIME_UTIL_H

#include <stdint.h>

/* Returns monotonic time in milliseconds since an arbitrary point.
 * Not affected by system clock changes. */
uint64_t monotonic_time_ms(void);

#endif
```

```c
/* src/time_util.c */
#include "time_util.h"
#include <time.h>
#include <stdint.h>

uint64_t monotonic_time_ms(void) {
    struct timespec ts;
    /* CLOCK_MONOTONIC is not affected by system time changes */
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (uint64_t)ts.tv_sec * 1000 + (uint64_t)ts.tv_nsec / 1000000;
}
```

Core Logic Skeleton Code

The main event loop dispatch function with detailed TODO comments mapping to the algorithm steps:

```c
/* src/event_loop.c */

#include "event_loop.h"

#include "timer_wheel.h"

#include "connection.h"

#include <sys/epoll.h>

#include <errno.h>

#include <stdlib.h>


int event_loop_run(struct event_loop* loop, int timeout_ms) {

    if (!loop || !loop->running) return -1;



    while (loop->running) {

        int n_events;

        int processed_events = 0;



        /* TODO 1: Calculate timeout for epoll_wait based on next timer expiration */

        /* - Call timer_wheel_next_expiration(loop->timer_wheel) to get ms until next timer

         * - If returns 0 (timers already expired), set timeout = 0 (non-blocking poll)

         * - If returns >0, set timeout = min(return_value, timeout_ms)

         * - If no timers (returns UINT64_MAX), set timeout = timeout_ms (or -1 for infinite) */



        /* TODO 2: Wait for I/O events with calculated timeout */

        /* - Call epoll_wait(loop->epoll_fd, loop->events, loop->max_events, timeout)

         * - If interrupted by signal (errno == EINTR), continue to next iteration

         * - Store number of returned events in n_events */



        /* TODO 3: Process I/O events */

        /* - For i from 0 to n_events-1:

         *   - Get epoll_event* ev = &loop->events[i]

         *   - Get connection* conn = (connection*)ev->data.ptr

         *   - Check ev->events for EPOLLERR or EPOLLHUP: if present, transition conn to STATE_CLOSING

         *   - If ev->events & EPOLLIN: invoke conn->read_callback(conn) or default read handler

         *   - If ev->events & EPOLLOUT: invoke conn->write_callback(conn) or default write handler

         *   - processed_events++ */
```

```
        /* TODO 4: Tick the timer wheel */
        /* - Calculate actual elapsed time (optional, using monotonic clock)
         *   - Store time before epoll_wait and after, compute difference
         *   - Or assume elapsed = timeout (less accurate)
         * - Call timer_wheel_tick(loop->timer_wheel) to process expired timers */

        /* TODO 5: Process deferred task queue */
        /* - While loop->task_queue is not NULL:
         *   - Get task = loop->task_queue
         *   - Update loop->task_queue = task->next
         *   - Execute task->callback(task->user_data)
         *   - Free(task) */

        /* TODO 6: Housekeeping */
        /* - Optionally, every 1000 iterations, log statistics
         * - Check loop->running flag (may be set to 0 by a callback via event_loop_stop) */
    }

    return 0;
}
```

Language-Specific Hints (C)

- **Epoll events array**: Allocate `loop->events` as `malloc(sizeof(struct epoll_event) * max_events)`. Ensure proper cleanup in destructor.
- **Non-blocking I/O**: After `accept()`, set socket non-blocking with `fcntl(fd, F_SETFL, O_NONBLOCK)`. Handle `EAGAIN` / `EWOULDBLOCK` in read/write loops.
- **Timer wheel time**: Use `monotonic_time_ms()` for current time in timer wheel to avoid issues when system time changes.
- **Connection cleanup**: Always `close(fd)` and free buffers in `connection_destroy()`. Set pointers to NULL after freeing to catch use-after-free.
- **Error handling**: Check all system call returns. Use `perror()` for logging during development.

Milestone Checkpoint: Verifying the Event Loop Tick

After implementing the event loop core (Milestone 1), verify the tick flow:

1. **Start the echo server**: Compile and run `./server 8080`.
2. **Connect multiple clients**: In separate terminals, run `telnet localhost 8080`.
3. **Observe I/O processing**: Type a line in one telnet session. It should be echoed back immediately.

4. **Verify timer integration**: After implementing timer wheel (Milestone 2), add a 5-second idle timeout. Connect a client but don't send data. Wait 5 seconds; the connection should close automatically.
5. **Check deferred tasks**: Add a deferred task that logs "tick" every 100 iterations. Run server and observe log output.

Expected behavior: The server handles multiple concurrent connections without blocking. CPU usage should be near 0% when idle, spike briefly when processing data. No connections should hang or leak.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
CPU at 100% when idle	`epoll_wait` timeout set to 0 or too small	Print calculated timeout before `epoll_wait`; ensure it's -1 when no timers	Properly compute `timer_wheel_next_expiration()`; cap minimum timeout to e.g., 1ms
Timer timeouts inaccurate (fire late)	Not using monotonic clock; assuming each tick takes exactly `timeout_ms`	Log actual elapsed time using `monotonic_time_ms()` before and after `epoll_wait`	Implement precise time measurement; advance timer wheel by actual elapsed milliseconds
Some connections never get data	Forgot to set non-blocking on accepted sockets	Check `fcntl` return value; add logging when `accept4` fails	Use `accept4` with `SOCK_NONBLOCK` or call `set_nonblocking()` after `accept`
Write buffer never empties	`EPOLLOUT` not registered when write returns `EAGAIN`	Log when `connection_write()` returns `EAGAIN`; check event registration	After partial write, call `event_loop_modify_fd()` to add `EPOLLOUT`
Memory leak over time	Connections not destroyed on close; timer references not freed	Use `valgrind --leak-check=full ./server`; add counters for connections created/destroyed	Ensure `connection_destroy()` frees all resources; cancel timer before destroying connection

Error Handling and Edge Cases

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

In an event-driven system handling 10,000+ concurrent connections, robust error handling isn't just about correctness—it's about survival. Each unhandled error can gradually leak resources until the system becomes unresponsive. Unlike traditional blocking I/O where errors appear as obvious failures in a single thread, non-blocking event loops require anticipating failures across thousands of independent connections while maintaining system stability.

Think of error handling in an event loop like air traffic control at a busy airport. The controller (event loop) must handle routine operations (normal I/O) while constantly monitoring for emergencies (errors), managing fuel shortages (resource constraints), and dealing with unresponsive pilots (slow clients). Missing a single distress signal or allowing one plane to occupy the runway too long can cascade into system-wide failure.

Non-Blocking I/O Error Patterns

Non-blocking I/O fundamentally changes how errors manifest. Instead of operations failing with hard errors, they often return "soft" conditions that indicate "try again later" or "the connection state changed." Successful event loop implementation requires treating these not as exceptions but as expected, routine conditions.

Mental Model: The Restaurant with a Ticket System
Imagine a busy restaurant where waitstaff (the event loop) take orders (read requests) and deliver food (write responses). When the

kitchen is temporarily backed up, the waitstaff don't stop working—they check back later (`EAGAIN`). If a customer leaves without paying (connection reset), the waitstaff don't panic—they simply clean the table (`ECONNRESET`). If a customer's mouth is full when trying to deliver more food (pipe broken), they wait until the customer is ready (`EPIPE`). Each condition has a standard operating procedure.

Common Error Conditions and Their Handling

The following table catalogs the critical error patterns and their proper handling in the event loop context:

Error Condition	Detection Method	Typical Cause	Recovery Action	Notes
`EAGAIN` / `EWOULDBLOCK`	`errno` after `read()` / `write()` / `accept()`	Non-blocking operation cannot complete immediately	Stop current operation, maintain current state, wait for next `EPOLLIN` / `EPOLLOUT` notification	Not actually an error—expected behavior in non-blocking mode
`ECONNRESET`	`errno` after `read()` / `write()`	Remote peer abruptly closed connection (TCP RST)	Immediately close local socket, free connection resources, deregister from epoll	May occur on either read or write operations
`EPIPE`	`errno` after `write()`	Writing to socket whose read side is closed (broken pipe)	Close socket, deregister from epoll	Often accompanied by `SIGPIPE` signal; disable signal with `MSG_NOSIGNAL` or `signal(SIGPIPE, SIG_IGN)`
`EINTR`	`errno` after system call	System call interrupted by signal	Restart the same system call (with same parameters)	Most system calls in event loop should be retried
`EBADF`	`errno` after system call	File descriptor is invalid (already closed)	Log error, skip processing for this fd	Indicates programming error in fd lifecycle management
`ENOMEM`	`errno` after `malloc()` / `accept()`	System out of memory	Close least important connections, continue serving	Apply backpressure by rejecting new connections
`EMFILE`	`errno` after `accept()`	Process file descriptor limit reached	Stop accepting temporarily, close idle connections	Use `epoll` to monitor listening socket only when below threshold
`EFAULT`	`errno` after system call	Invalid buffer pointer (programming bug)	Log and terminate program	Cannot recover from memory corruption

Architecture Decision: Treating EAGAIN as Expected Flow, Not Error

- **Context**: In non-blocking I/O, operations frequently cannot complete immediately. The system must distinguish between temporary unavailability and permanent failure.
- **Options Considered**:
  1. Treat all non-zero return values as errors requiring connection closure
  2. Special-case `EAGAIN` / `EWOULDBLOCK` but treat other errors as fatal
  3. Categorize errors into retryable, connection-fatal, and system-fatal
- **Decision**: Implement option 3 with detailed categorization
- **Rationale**: Option 1 would close connections prematurely under load. Option 2 misses important distinctions (e.g., `EINTR` should retry, `ENOMEM` requires different handling than `ECONNRESET`). Detailed categorization maximizes connection longevity while ensuring resource recovery.
- **Consequences**: More complex error handling logic but higher connection stability under varying network conditions.

Error Category	Example Errors	Handling Strategy	Connection Fate
Retryable	`EAGAIN`, `EWOULDBLOCK`, `EINTR`	Retry later (same event loop tick or next notification)	Preserved
Connection-Fatal	`ECONNRESET`, `EPIPE`, `ETIMEDOUT`	Close connection, free resources	Terminated
System-Fatal	`ENOMEM`, `EMFILE`, `EFAULT`	Apply backpressure, log, possible graceful degradation	May terminate some connections

Implementation Strategy for Error Handling in `connection_read()` and `connection_write()`

Each I/O operation in the event loop follows a consistent error handling pattern:

1. **Attempt Operation**: Call `read()` or `write()` on non-blocking socket
2. **Check Return Value**:
   - `> 0`: Success, process data
   - `= 0`: EOF (remote closed gracefully), initiate clean closure
   - `-1`: Check `errno` for specific condition
3. **Handle `errno` Cases**:
   - `EAGAIN` / `EWOULDBLOCK`: Update epoll registration if needed, return "incomplete"
   - `ECONNRESET` / `EPIPE`: Mark connection for closure
   - Others: Log, classify, take appropriate action
4. **State Machine Transition**: Update `connection` state based on outcome

Example Walkthrough: Handling Partial Read with EAGAIN

Consider a connection in `STATE_READING` with `read_buffer` size 4096 bytes. The event loop receives `EPOLLIN` notification and calls `connection_read()`:

1. `read()` returns 2048 bytes (half the buffer filled)
2. The HTTP parser processes these bytes but doesn't find a complete request
3. Next `read()` attempt returns -1 with `errno = EAGAIN`
4. Instead of closing the connection, we:
   - Update `read_buffer_used` to 2048
   - Keep connection in `STATE_READING`
   - Ensure epoll still has `EPOLLIN` interest (level-triggered will re-notify)

    - Return to event loop to process other connections
5. When more data arrives, epoll will notify again, and we resume reading

Common Pitfalls in Non-Blocking Error Handling

⚠️ Pitfall: Treating EAGAIN as Fatal Error
Description: Closing connections when `read()` / `write()` returns -1 with `EAGAIN`
Why It's Wrong: This eliminates the benefit of non-blocking I/O—connections would close under normal load when buffers are temporarily full
Fix: Always check `errno` after -1 return; handle `EAGAIN` / `EWOULDBLOCK` as "operation incomplete"

⚠️ Pitfall: Ignoring ECONNRESET on Write
Description: Assuming connection errors only appear on read operations
Why It's Wrong: A peer may send RST while we're writing, causing `ECONNRESET` on `write()`
Fix: Check for connection-fatal errors after both `read()` and `write()` operations

⚠️ Pitfall: Forgetting to Handle EPIPE Signal
Description: Process crashes when writing to closed connection due to `SIGPIPE`
Why It's Wrong: Default Unix behavior sends `SIGPIPE` to process writing to broken pipe
Fix: Set `signal(SIGPIPE, SIG_IGN)` at program start or use `send(fd, buf, len, MSG_NOSIGNAL)`

Edge Cases: Slow Clients and Backpressure

In high-concurrency systems, a small number of slow clients can consume disproportionate resources, eventually causing denial of service to well-behaved clients. This "slow client problem" requires deliberate backpressure mechanisms and careful buffer management.

Mental Model: The Highway On-Ramp Metering System
Imagine a highway (server resources) with many on-ramps (client connections). Without metering, a few slow-moving vehicles (slow clients) entering the highway can cause traffic jams for everyone. The metering system (backpressure) regulates how quickly vehicles enter based on highway capacity. Each on-ramp has a limited queue space (write buffer). When the queue fills, new vehicles must wait (application pauses writes).

Managing Write Buffers with EPOLLOUT Backpressure

The core challenge with slow clients is managing the write path. When a client cannot accept data as fast as we can send it, we must avoid unbounded buffer growth while maintaining efficient resource utilization.

Backpressure Strategy Using epoll Edge-Triggered Mode
Edge-triggered epoll (`EPOLLET`) provides natural backpressure by only notifying when the socket transitions from unwritable to writable:

1. **Initial State**: After generating response data, attempt `write()`
2. **Partial Write**: If `write()` returns fewer bytes than buffer content (or `EAGAIN`), store remaining data in `write_buffer`
3. **Register EPOLLOUT**: Call `event_loop_modify_fd()` to add `EPOLLOUT` interest
4. **Wait for Notification**: Event loop won't notify again until socket becomes writable
5. **EPOLLOUT Notification**: When client can accept more data, epoll notifies
6. **Resume Write**: Attempt another `write()` from remaining buffer
7. **Completion**: When buffer empty, remove `EPOLLOUT` interest with `event_loop_modify_fd()`

Write Buffer State Machine
The `connection` 's write buffer management follows this state machine:

Current State	Event	Next State	Action
`WRITE_IDLE`	Response data generated	`WRITE_PENDING`	Copy data to `write_buffer`, attempt `write()`
`WRITE_PENDING`	`write()` succeeds fully	`WRITE_IDLE`	Clear `write_buffer`, keep only `EPOLLIN` interest
`WRITE_PENDING`	`write()` partial/ `EAGAIN`	`WRITE_BACKPRESSURED`	Keep unsent data in buffer, add `EPOLLOUT` interest
`WRITE_BACKPRESSURED`	`EPOLLOUT` notification	`WRITE_PENDING`	Attempt `write()` on remaining buffer
`WRITE_BACKPRESSURED`	Timeout expires	`STATE_CLOSING`	Close connection (slow client)
Any write state	`EPOLLHUP` / `EPOLLERR`	`STATE_CLOSING`	Close connection immediately

> **Architecture Decision: Fixed-Size vs. Dynamic Write Buffers**
>
> - **Context**: Each connection needs buffer space for pending writes when clients are slow. Buffer sizing impacts memory usage and fairness.
> - **Options Considered**:
>   1. Fixed-size buffers per connection (`WRITE_BUFFER_SIZE` bytes)
>   2. Dynamically growing buffers (realloc as needed)
>   3. Per-connection buffer pools with chunked allocation
> - **Decision**: Fixed-size buffers with fair queuing
> - **Rationale**: Fixed buffers provide predictable memory usage critical for 10K connections (e.g., 4KB × 10K = 40MB). Dynamic buffers could exhaust memory from few slow clients. Buffer pools add complexity without solving the fundamental problem: slow clients shouldn't consume unbounded resources.
> - **Consequences**: When write buffer fills, we must stop generating data for that connection, potentially requiring flow control at application protocol level.

Handling Half-Closed Connections (TCP FIN)

TCP allows independent closure of each direction of communication (half-close), creating subtle edge cases:

Scenario: Client sends `FIN` (reads closed) but keeps write side open

- **Detection**: `read()` returns 0 (EOF) but `write()` may still succeed
- **Proper Handling**:
  1. When `read()` returns 0, mark read side closed
  2. If write buffer has pending data, continue attempting writes
  3. After write buffer empties, send `FIN` with `shutdown(fd, SHUT_WR)` or close
  4. Monitor for client's `FIN` to fully close

Implementation Table for Half-Close States:

Socket State	Read Behavior	Write Behavior	epoll Events	Action
Fully open	Normal	Normal	`EPOLLIN` | `EPOLLOUT`	Normal processing
Read-side closed	`read()` returns 0	May succeed	`EPOLLIN` (hup)	Complete writes, then close
Write-side closed	May succeed	`write()` fails with `EPIPE`	`EPOLLOUT` (hup)	Close immediately
Both sides closed	`read()` returns 0	`write()` fails	`EPOLLHUP`	Close immediately

Slow Loris Attack Mitigation

A Slow Loris attack opens many connections and sends data extremely slowly, exhausting server resources. Our architecture provides natural defenses:

1. **Idle Timeouts**: Timer wheel closes connections inactive for configurable period
2. **Read Timeouts**: Separate timeout for incomplete requests (e.g., 10 seconds for HTTP headers)
3. **Connection Limits**: Maximum concurrent connections prevents exhaustion
4. **Buffer Limits**: Fixed read buffer prevents slow clients from consuming unbounded memory

Timeout Hierarchy for HTTP Connections:

Timeout Type	Typical Value	Reset Condition	Expiration Action
Keep-alive idle	5 seconds	Any data received	Close connection
Header read	10 seconds	Complete header received	Close connection
Request body	30 seconds	Body bytes received	Close connection
Write backpressure	30 seconds	Write completes	Close connection

Example Walkthrough: Handling Slow Client with Partial Writes
A client connects and requests a 100KB file. The server generates response but client has poor connectivity:

1. Server writes 8KB successfully, but next `write()` returns `EAGAIN`
2. Server registers `EPOLLOUT`, moves to `WRITE_BACKPRESSURED` state
3. Timer wheel adds write timeout (30 seconds)
4. Client receives data slowly over 15 seconds, triggering occasional `EPOLLOUT` notifications
5. Each notification writes more data until buffer empty
6. Server removes `EPOLLOUT` interest, cancels write timeout
7. If timeout had expired first, server would close connection

Common Pitfalls with Backpressure and Slow Clients

⚠️ Pitfall: Unbounded Write Buffer Growth
Description: Continuously appending to write buffer when client can't keep up
Why It's Wrong: Single slow client can exhaust all server memory, causing denial of service
Fix: Use fixed-size buffer; when full, stop generating data and apply backpressure to application layer

⚠️ Pitfall: Busy Loop on EPOLLOUT
Description: In level-triggered mode, continuously receiving `EPOLLOUT` when socket writable
Why It's Wrong: Wastes CPU when client is slow but socket buffer has some space
Fix: Use edge-triggered (`EPOLLET`) or deregister `EPOLLOUT` after write attempt, re-register when `EAGAIN`

⚠️ Pitfall: Ignoring FIN While Write Buffer Pending
Description: Closing connection immediately when `read()` returns 0, even with pending writes
Why It's Wrong: Client may still be reading our response; immediate close loses data
Fix: Implement proper half-close handling: complete pending writes before full close

⚠️ Pitfall: No Per-Connection Write Timeout
Description: Allowing slow client to hold buffer indefinitely
Why It's Wrong: Enables Slow Loris attacks and resource exhaustion
Fix: Timer wheel timeout for each connection in `WRITE_BACKPRESSURED` state

Resource Exhaustion Strategies

At 10,000+ connections, resource exhaustion becomes a real concern. The system must gracefully degrade rather than catastrophically fail.

Memory Pressure Response:

1. **Monitor Allocation Failures**: When `malloc()` returns NULL, close idle connections first
2. **Connection Prioritization**: Under memory pressure, close connections with:
   - No pending requests
   - No write buffer data
   - Longest idle time
3. **Emergency Mode**: Stop accepting new connections when below free memory threshold

File Descriptor Exhaustion Response:

1. **Monitor `accept()` Failures**: When `EMFILE` occurs, temporarily disable listening socket monitoring
2. **LRU Connection Closure**: Close least recently used connections to free file descriptors
3. **Gradual Recovery**: Re-enable acceptance when FD count drops below threshold (e.g., 90% of limit)

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Error Detection	Check `errno` after each system call	Use `getsockopt(fd, SOL_SOCKET, SO_ERROR)` for deferred error checking
Backpressure Signaling	Fixed-size buffers with `EAGAIN` handling	Dynamic buffer sizing with watermarks and application-level flow control
Timeout Management	Single idle timeout per connection	Multiple hierarchical timeouts (header, body, keep-alive, write)

B. File/Module Structure

```
event_loop_project/
├── src/
│   ├── error_handling.c          # Error classification and handling utilities
│   ├── error_handling.h
│   ├── connection.c              # Connection error handling methods
│   ├── connection.h
│   ├── buffer.c                  # Fixed-size buffer management with backpressure
│   ├── buffer.h
│   ├── event_loop.c              # Main loop with error recovery
│   └── event_loop.h
└── examples/
    └── slow_client_test.c        # Test program for backpressure scenarios
```

C. Infrastructure Starter Code

Complete Error Handling Utilities (error_handling.c):

```c
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include "error_handling.h"

// Categorize errno value into error type
error_category_t categorize_error(int errno_value) {
    switch (errno_value) {
        case EAGAIN:
        case EWOULDBLOCK:
        case EINTR:
            return ERROR_RETRYABLE;

        case ECONNRESET:
        case EPIPE:
        case ETIMEDOUT:
        case ECONNABORTED:
            return ERROR_CONNECTION_FATAL;

        case ENOMEM:
        case EMFILE:
        case ENFILE:
            return ERROR_SYSTEM_FATAL;

        case EBADF:
        case EINVAL:
        case EFAULT:
            return ERROR_PROGRAMMING;

        default:
            return ERROR_UNKNOWN;
    }
}
```

```c
// Check socket for pending errors using getsockopt
int check_socket_error(int fd) {
    int error = 0;
    socklen_t len = sizeof(error);

    if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &len) < 0) {
        return errno;
    }
    return error;
}

// Disable SIGPIPE signal for entire process
void disable_sigpipe(void) {
    #ifdef SO_NOSIGPIPE
        // Some systems have socket-level option
        // (Set on per-socket basis)
    #else
        signal(SIGPIPE, SIG_IGN);
    #endif
}

// Safe write with SIGPIPE prevention
ssize_t safe_write(int fd, const void *buf, size_t count) {
    #ifdef MSG_NOSIGNAL
        return send(fd, buf, count, MSG_NOSIGNAL);
    #else
        return write(fd, buf, count);
    #endif
}
```

D. Core Logic Skeleton Code

Connection Read with Comprehensive Error Handling (connection.c):

```c
// Performs non-blocking read with full error handling
// Returns: >0 bytes read, 0 for EOF, -1 for error (check errno), -2 for would block
int connection_read(struct connection *conn) {
    if (!conn || conn->fd < 0) {
        return -1;  // Programming error
    }

    // Ensure buffer has space
    int available = READ_BUFFER_SIZE - conn->read_buffer_used;
    if (available <= 0) {
        // Buffer full - protocol should consume data
        return -1;  // Protocol error
    }

    ssize_t n = read(conn->fd,
                     conn->read_buffer + conn->read_buffer_used,
                     available);

    if (n > 0) {
        // Successful read
        conn->read_buffer_used += n;

        // Reset idle timeout since we received data
        if (conn->timer_ref) {
            timer_wheel_remove(conn->loop->timer_wheel, conn->timer_ref);
            conn->timer_ref = timer_wheel_add(conn->loop->timer_wheel,
                                              IDLE_TIMEOUT_MS,
                                              idle_timeout_callback,
                                              conn);
        }
        return n;
    }
    else if (n == 0) {
```

```
        // EOF - remote closed connection gracefully
        return 0;
    }
    else {  // n < 0
        int err = errno;
        error_category_t category = categorize_error(err);

        switch (category) {
            case ERROR_RETRYABLE:
                // EAGAIN/EWOULDBLOCK/EINTR - try again later
                return -2;  // Special code for "would block"

            case ERROR_CONNECTION_FATAL:
                // Connection broken - schedule cleanup
                conn->state = STATE_CLOSING;
                return -1;

            case ERROR_SYSTEM_FATAL:
                // System-wide issue - apply backpressure
                log_system_error(err, "read");
                // Might close some connections under memory pressure
                return -1;

            default:
                // Unknown or programming error
                conn->state = STATE_CLOSING;
                return -1;
        }
    }
}

// Handles write backpressure with EPOLLOUT registration
int connection_write(struct connection *conn) {
    if (!conn || conn->fd < 0 || conn->write_buffer_used == 0) {
```

```
    return 0;  // Nothing to write or invalid state
}

size_t remaining = conn->write_buffer_used - conn->write_buffer_sent;
ssize_t n = safe_write(conn->fd,
                       conn->write_buffer + conn->write_buffer_sent,
                       remaining);

if (n > 0) {
    // Partial or complete write successful
    conn->write_buffer_sent += n;

    if (conn->write_buffer_sent == conn->write_buffer_used) {
        // All data written - clear buffer
        conn->write_buffer_used = 0;
        conn->write_buffer_sent = 0;

        // Remove EPOLLOUT interest if we had it
        event_loop_modify_fd(conn->loop, conn->fd, EPOLLIN, conn);

        // Cancel write timeout if active
        if (conn->write_timer_ref) {
            timer_wheel_remove(conn->loop->timer_wheel, conn->write_timer_ref);
            conn->write_timer_ref = NULL;
        }

        // For HTTP keep-alive, transition to waiting for next request
        if (conn->proto_state == HTTP_WRITING) {
            conn->proto_state = HTTP_KEEPALIVE;
        }
    }
    return n;
}
```

```
    else if (n < 0) {
        int err = errno;

        if (err == EAGAIN || err == EWOULDBLOCK) {
            // Socket buffer full - register for EPOLLOUT notification

            // TODO 1: Add EPOLLOUT interest using event_loop_modify_fd
            // TODO 2: Add write timeout timer (e.g., 30 seconds)
            // TODO 3: Store the timer reference in conn->write_timer_ref
            // TODO 4: Return 0 to indicate write paused

            return 0;
        }
        else {
            // Real error - close connection
            conn->state = STATE_CLOSING;
            return -1;
        }
    }

    return 0;  // Should not reach here
}

// Callback for write timeout (client too slow)
void write_timeout_callback(void *user_data) {
    struct connection *conn = (struct connection *)user_data;

    // TODO 1: Check if write buffer still has pending data
    // TODO 2: If yes, log slow client warning
    // TODO 3: Set connection state to STATE_CLOSING
    // TODO 4: Clear write_timer_ref to avoid double-free
}
```

E. Language-Specific Hints for C

1. **`errno` Preservation**: `errno` is thread-local but can be overwritten by library calls. Save it immediately after system call failure:

```c
ssize_t n = write(fd, buf, len);
if (n < 0) {
    int saved_errno = errno;  // Save immediately
    // ... use saved_errno
}
```

2. **Socket Error Checking**: Use `getsockopt(fd, SOL_SOCKET, SO_ERROR, ...)` to check for asynchronous errors that epoll might report via `EPOLLERR`.
3. **Non-Blocking Accept**: Even with non-blocking listening socket, `accept()` can return `EAGAIN` if no pending connections between epoll notification and accept call.
4. **Portability**: `EAGAIN` and `EWOULDBLOCK` may have the same value on some systems but different on others. Check for both.

F. Milestone Checkpoint: Error Handling Verification

After implementing error handling, verify with these tests:

1. **Slow Client Test**:

```
$ nc localhost 8080
(Type very slowly, one character every 10 seconds)
```

Expected: Server should timeout connection after idle timeout (e.g., 30 seconds) instead of waiting forever.

2. **Connection Reset Test**:

```
$ python3 -c "import socket; s=socket.socket(); s.connect(('localhost',8080)); s.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER, struct.pack('ii',1,0)); s.close()"
```

Expected: Server should handle `ECONNRESET` gracefully without crashing, logging appropriate message.

3. **Memory Pressure Test**:

```
$ ulimit -n 1000  # Limit file descriptors
$ ./server  # Start server
$ ./test_client --connections 2000  # Attempt to exceed limits
```

Expected: Server should reject new connections when near limit rather than crashing.

4. **Write Backpressure Verification**:

```
$ ./slow_receiver.py  # Client that reads 1 byte/second
$ curl http://localhost:8080/large-file.bin
```

Expected: Server should pause writes when client buffer fills, resume when client reads.

G. Debugging Tips for Error Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
CPU at 100%	Busy loop on `EPOLLOUT` (level-triggered)	Check if `EPOLLOUT` handler runs continuously	Use edge-triggered or deregister `EPOLLOUT` after write attempt
Connections stuck in `WRITING`	Missing `EPOLLOUT` registration after `EAGAIN`	Log when write returns `EAGAIN` and verify `event_loop_modify_fd()` called	Ensure `EPOLLOUT` added when write blocks
Memory exhaustion with few clients	Unbounded buffer growth	Monitor `write_buffer_used` per connection; set maximum	Implement fixed buffer size and backpressure
Random connection resets	Missing `EAGAIN` handling	Add logging to `connection_read()` / `connection_write()` errno handling	Check for all retryable errors (`EAGAIN` , `EWOULDBLOCK` , `EINTR`)
`accept()` returning `EMFILE`	File descriptor leak	Use `lsof -p <pid>` to see open files; check connection cleanup	Ensure all closed sockets have resources freed and epoll deregistered

---

Testing Strategy

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Testing a high-concurrency event loop server requires a multi-layered approach that verifies correctness at each architectural level while maintaining the performance characteristics that make the system valuable. Think of testing this system like validating a high-performance race car: you need individual component tests (engine, brakes, transmission), integration tests (how the powertrain works together), and finally track testing under realistic load conditions to verify it actually achieves 10,000+ concurrent connections with stable performance.

The testing strategy must address three critical dimensions: **functional correctness** (does it handle connections correctly?), **concurrency robustness** (does it handle 10K connections without failure?), and **performance characteristics** (does it maintain acceptable latency under load?). Each milestone introduces new complexity that requires specific verification approaches, from basic echo functionality in Milestone 1 to full HTTP protocol compliance in Milestone 4.

Unit and Integration Tests

Unit testing focuses on individual components in isolation, while integration testing verifies that components work together correctly. Given the event-driven nature of the system, traditional unit testing approaches need adaptation to handle asynchronous callbacks and non-blocking I/O.

Timer Wheel Unit Testing

The hierarchical timer wheel is a complex data structure with specific performance guarantees. Testing it requires verifying both correctness (timers fire at the right time) and performance (O(1) operations).

Mental Model: Think of the timer wheel as a multi-tiered conveyor belt system in a factory. Items (timers) are placed on different belts based on when they need to be processed. The belts move at different speeds, and items must cascade from slower belts to faster belts as their deadline approaches. Testing ensures items don't fall off the belts, don't get processed too early or late, and the conveyor system doesn't jam under heavy load.

Test Strategy Table:

Test Category	Test Cases	Verification Method	Expected Outcome
Basic Operations	Insert single timer, tick before expiration	Manual timer counting	Timer doesn't fire early
Basic Operations	Insert single timer, tick past expiration	Callback invocation check	Timer fires exactly once
Basic Operations	Insert and cancel timer before expiration	Memory leak check	Timer callback never invoked, memory freed
Cascading Behavior	Insert timer that spans multiple wheel levels	Internal state inspection	Timer cascades correctly through levels
Performance	Insert 10,000 timers with random expirations	Time measurement	All operations complete in O(1) time
Edge Cases	Insert timer with 0ms timeout	Immediate execution check	Timer fires on next `timer_wheel_tick`
Edge Cases	Insert timer with very large timeout (overflow)	Boundary checking	Timer handled correctly or rejected

Architecture Decision Record: Timer Wheel Testing Approach

> **Decision: Mockable Time Source for Deterministic Testing**
>
> - **Context**: The timer wheel depends on monotonic time for determining when timers expire. Real-time sources make tests non-deterministic and flaky.
> - **Options Considered**:
>   1. Use real `clock_gettime(CLOCK_MONOTONIC)` and rely on timing tolerances
>   2. Inject a mock time source via function pointer
>   3. Build deterministic test harness with controllable virtual time
> - **Decision**: Use a mockable time source via function pointer in the `timer_wheel` structure
> - **Rationale**: Deterministic tests are essential for CI/CD pipelines. A mock time source allows tests to "fast-forward" time without waiting, making tests run quickly and predictably. The function pointer approach adds minimal overhead (one indirect call) while providing full test control.
> - **Consequences**: The `timer_wheel` structure must store a `get_time_ms` function pointer. Production code uses `clock_gettime`, while tests use a controlled counter. This adds one function pointer dereference per time query but enables comprehensive testing.

Timer Wheel Test Comparison Table:

Option	Pros	Cons	Selected?
Real time with tolerances	Simple implementation, no abstraction	Flaky tests, slow test execution	❌
Function pointer injection	Deterministic tests, fast execution	Slight runtime overhead, more complex interface	✅
Virtual time harness	Most control, can simulate edge cases	Heavy infrastructure, diverges from production	❌

Common Timer Wheel Testing Pitfalls:

⚠️ Pitfall: Assuming Linear Time Progression in Tests

- **Description**: Writing tests that assume `timer_wheel_tick` is called at regular intervals matching the wheel resolution.

- **Why It's Wrong**: In production, `timer_wheel_tick` may be called irregularly based on I/O activity. The wheel must handle time jumps correctly (timer cascading).
- **Fix**: Test with irregular time progressions: small jumps, large jumps, backward jumps (though monotonic time shouldn't go backward), and skipped ticks.

⚠️ **Pitfall: Not Testing Timer Cancellation Race Conditions**

- **Description**: Testing timer cancellation only in isolation, not when a timer is about to expire.
- **Why It's Wrong**: In production, a timer might be cancelled in a callback while the wheel is iterating through expired timers.
- **Fix**: Create stress tests where timers are cancelled from other timer callbacks, and verify no use-after-free or double-free occurs.

HTTP Parser Unit Testing

The incremental HTTP parser must handle partial data, malformed requests, and various HTTP/1.1 features. Unlike a batch parser that receives complete requests, the incremental parser maintains state between calls to `http_parser_feed`.

Mental Model: Imagine the HTTP parser as a factory assembly line inspector. Parts of an HTTP request (bytes) arrive on a conveyor belt at irregular intervals. The inspector must examine each part as it arrives, determine if it's a valid piece of a request, and raise a flag when a complete, valid request has passed through. Sometimes defective parts arrive (malformed bytes), and the inspector must reject them without stopping the entire line.

HTTP Parser Test Matrix:

Test Scenario	Input Pattern	Expected Parser State	Expected Output
Complete request in one chunk	Full HTTP request	`HTTP_READING` → `HTTP_WRITING`	Request fully parsed
Split across read boundaries	"GET / HT" then "TP/1.1\r\n\r\n"	`HTTP_READING` persists between feeds	Request parsed after second feed
Headers across multiple reads	Headers split mid-line	Header state maintained	All headers correctly assembled
Chunked encoding	Transfer-Encoding: chunked	Handles chunk size and data separately	Body correctly reconstructed
Malformed method	"GEX / HTTP/1.1\r\n\r\n"	Error state with error code	`http_parser_feed` returns error
Request too large	Headers exceed buffer	Error state for overflow	Parser rejects with size error
Keep-alive vs close	Connection header variations	Sets keep-alive flag correctly	Correct connection handling

Step-by-Step Parser Test Procedure:

1. **Initialize parser**: Create `http_parser` instance with `http_parser_create()`
2. **Feed partial data**: Call `http_parser_feed()` with first chunk of request data
3. **Verify intermediate state**: Check parser state indicates more data needed
4. **Feed remaining data**: Provide remaining bytes to complete request
5. **Verify completion**: Check parser indicates headers are complete
6. **Extract parsed request**: Use accessor functions to validate parsed fields
7. **Reset for next request**: Call `http_parser_reset()` and verify clean state

Edge Case Testing Table:

Edge Case	Test Approach	Validation
Empty read (EAGAIN)	Feed 0 bytes to parser	Parser state unchanged
CRLF vs LF line endings	Mix of \r\n and \n	Headers parsed correctly regardless
Unicode in headers	UTF-8 in header values	Preserved or rejected based on policy
Multiple consecutive requests	Keep-alive connection	Parser resets correctly between requests
Extremely long URL	URL exceeds buffer	Handled with configurable max
Missing Host header (HTTP/1.1)	Request without Host	Parser rejects or handles based on spec

Event Loop Integration Testing

Integration testing verifies that the event loop coordinates correctly between I/O, timers, and callbacks. This requires creating mock sockets or using socketpair to simulate network behavior without actual network stack.

Test Architecture Decision:

> **Decision: Use Socketpair for Loopback Testing**
>
> - **Context**: Testing network code requires simulating connection establishment, data transfer, and closure without involving the actual network stack.
> - **Options Considered**:
>   1. Real network sockets (localhost)
>   2. Socketpair() for loopback within process
>   3. Mock file descriptor layer
> - **Decision**: Use `socketpair(AF_UNIX, SOCK_STREAM | SOCK_NONBLOCK, 0)` to create connected socket pairs
> - **Rationale**: Socketpair provides real file descriptors with full TCP-like semantics (stream-oriented, bidirectional) but operates entirely in kernel memory without network overhead. The non-blocking flag ensures it works with the event loop. This approach tests the actual epoll code path without network variability.
> - **Consequences**: Tests run fast and deterministically. However, some TCP-specific behaviors (Nagle's algorithm, TCP_NODELAY, SO_LINGER) aren't tested, requiring supplemental integration tests with real network.

Integration Test Sequence:

1. **Setup phase**:
   - Create event loop with `event_loop_create(MAX_EVENTS)`
   - Create socket pair: `socketpair()` yields two FDs (server_fd, client_fd)
   - Set both to non-blocking with `set_nonblocking()`
   - Register server_fd with event loop for read events
2. **Connection establishment test**:
   - Write data to client_fd (simulating client request)
   - Run event loop for one iteration with `event_loop_run(loop, 10)`
   - Verify read callback invoked on server_fd
   - Verify data read matches data written
3. **Timer integration test**:
   - Add timer via `timer_wheel_add()` with short timeout
   - Run event loop with timeout longer than timer expiration

- Verify timer callback invoked
- Verify timer removed from wheel after firing

4. **Backpressure simulation test**:
   - Fill write buffer on server_fd
   - Attempt to write more data (should return EAGAIN)
   - Verify EPOLLOUT registration occurs
   - Drain buffer from client side
   - Verify write callback invoked and remaining data sent

Common Integration Testing Pitfalls:

⚠️ **Pitfall: Not Testing Event Loop with Full epoll_wait Buffer**

- **Description**: Testing with only a few events, never hitting the `max_events` limit of epoll_wait.
- **Why It's Wrong**: In production with 10K connections, epoll_wait may return hundreds of ready events per iteration. The event loop must handle all returned events correctly.
- **Fix**: Create tests that generate more ready file descriptors than `MAX_EVENTS` (typically 1024) and verify all are processed in a single loop iteration.

⚠️ **Pitfall: Assuming Immediate Callback Execution**

- **Description**: Tests that assume a callback runs immediately after the triggering event.
- **Why It's Wrong**: In the actual event loop, callbacks are scheduled in the deferred task queue and execute after all I/O events in the current tick.
- **Fix**: Structure tests to run full event loop iterations and verify callback execution order matches the documented sequence (I/O events → timer tick → deferred tasks).

Milestone Checkpoints and Verification

Each milestone has specific acceptance criteria that must be verified through both automated tests and manual verification. The following checkpoints provide concrete steps to validate progress.

Milestone 1: epoll Basics

Acceptance Criteria Verification Table:

Criteria	Verification Method	Expected Result	Command/Tool
epoll instance creation	Check return value and error handling	`epoll_create1()` returns valid FD > 0	Program startup logs
Listening socket registration	Monitor epoll events	Listening FD appears in epoll_wait returns	`strace -e epoll_ctl`
Non-blocking socket configuration	Attempt blocking operation	`accept()` returns EAGAIN when no connections	Test with `fcntl` inspection
Multiple concurrent connections	Connect multiple clients	All connections accepted and serviced	`telnet` x10 concurrently
Echo functionality	Send data, check echo	Data returned identically	`echo "test"
Non-blocking read/write handling	Send partial data	Server handles EAGAIN correctly	Slow client simulation

Manual Verification Procedure:

1. **Start the echo server**:

```bash
./event_loop_server --port 8080 --echo
```

2. **Verify basic connectivity** (in separate terminal):

```bash
echo "Hello, event loop" | nc localhost 8080
```

Expected output: "Hello, event loop" echoed back.

3. **Test concurrent connections**:

```bash
# Start 10 concurrent connections
for i in {1..10}; do
  (echo "Connection $i" | nc localhost 8080 -w 1) &
done
wait
```

Expected: All connections receive correct echo without server crash.

4. **Test non-blocking behavior**:

```bash
# Use telnet to send data slowly
telnet localhost 8080
# Type characters one by one with delays
```

Expected: Server handles partial reads without blocking, eventually echoes complete input.

5. **Monitor epoll activity** (for debugging):

```bash
strace -e epoll_wait,epoll_ctl -p $(pidof event_loop_server)
```

Expected: See regular epoll_wait returns with multiple events under load.

Milestone 1 Success Criteria:

- Server handles at least 100 concurrent echo connections without dropping
- CPU usage remains low (< 5%) when idle
- Memory usage grows linearly with connections (check for leaks)
- No blocking calls (verify with `strace`)

Milestone 2: Timer Wheel and Timeouts

Acceptance Criteria Verification Table:

Criteria	Verification Method	Expected Result	Validation Command
O(1) timer operations	Performance measurement	Insert/delete 10K timers < 10ms	Benchmark test program
Connection idle timeouts	Inactive connection test	Connection closes after timeout	`telnet` then wait
epoll_wait timeout integration	Timer-driven wakeups	epoll_wait returns before timer expiry	Log timestamps
Timer cancellation	Active connection reset	Timer removed on activity, connection stays open	Send data before timeout
Hierarchical wheel correctness	Timer distribution test	Timers fire at correct times across time scales	Test with varied timeouts

Verification Test Suite:

1. **Timer Wheel Unit Test**:

```bash
./test_timer_wheel
```

Expected output: All tests pass, including performance test showing O(1) characteristics.

2. **Idle Timeout Integration Test**:

```bash
# Start server with 5-second idle timeout
./event_loop_server --port 8080 --idle-timeout 5000

# Connect but send no data
nc localhost 8080 &
NC_PID=$!

# Wait 6 seconds
sleep 6

# Check if connection closed
kill -0 $NC_PID 2>/dev/null && echo "FAIL: Connection still alive" || echo "PASS: Connection closed"
```

3. **Timer Cancellation Test**:

```bash
# Start server with 10-second timeout
./event_loop_server --port 8080 --idle-timeout 10000

# Connect and send periodic data
(echo "initial"; sleep 3; echo "refresh"; sleep 3; echo "refresh") | nc localhost 8080
```

Expected: Connection remains open through refreshes, closes only after final 10-second inactivity.

4. **Timer Performance Benchmark**:

```bash
./benchmark_timer --timers 10000 --operations 100000
```

Expected: Average operation time < 1µs, linear scaling with operation count.

Common Milestone 2 Verification Issues:

- **Timer not firing**: Check that `epoll_wait` timeout is calculated from `timer_wheel_next_expiration()`
- **Premature timeout**: Verify timer cancellation on I/O activity removes timer from wheel
- **Memory leak in timers**: Use valgrind to track timer allocation/deallocation
- **Clock skew issues**: Ensure using `CLOCK_MONOTONIC` not `CLOCK_REALTIME`

Milestone 3: Async Task Scheduling

Acceptance Criteria Verification Table:

Criteria	Verification Method	Expected Result	Test Approach
Callback registration API	API usage test	Clean registration/deregistration	Unit test with mock FDs
Deferred task queue	Execution order test	Tasks run after I/O, in registration order	Test with sequence logging
One-shot timers	Single execution test	Callback fires once, not re-registered	Timer test with counter
Repeating timers	Multiple execution test	Callback fires at each interval	Test with 3+ iterations
Event registration set safety	Modification during iteration	No crash when callbacks modify registrations	Stress test with concurrent mods
Callback re-entrancy	Nested registration test	System handles callbacks registering new callbacks	Recursive callback test

Verification Procedures:

1. **Callback API Test**:

```bash
./test_callback_api
```

Expected: Tests demonstrate registering read/write/timer callbacks, modifying interest, and clean teardown.

2. **Deferred Task Order Verification**:

```bash
./event_loop_server --port 8080 --test-deferred-tasks
```

Expected server output:

```
[INFO] I/O event processed
[INFO] Deferred task 1 executed
[INFO] Deferred task 2 executed
```

Tasks must execute after all I/O events in the same tick.

3. **Re-entrancy Stress Test**:

```bash
./stress_test_reentrant --callbacks 1000 --depth 5
```

Expected: System handles nested callback registration without stack overflow or memory leak.

4. **Event Loop API Completeness Test**:

```bash
./test_event_loop_api
```

Tests all API functions: `event_loop_register_read`, `event_loop_register_write`, `event_loop_defer_task`, `event_loop_modify_fd`, `event_loop_unregister_fd`.

Success Criteria for Milestone 3:

- All API functions behave as documented with proper error handling
- Deferred tasks never execute before I/O events in the same loop iteration
- Repeating timers maintain accurate scheduling despite timer wheel ticks
- No crashes when callbacks modify the event registration set

Milestone 4: HTTP Server on Event Loop

Acceptance Criteria Verification Table:

Criteria	Verification Method	Expected Result	Benchmark Target
Incremental HTTP parsing	Partial request test	Request parsed correctly across multiple reads	Functional test suite
Buffer management	Large request/response test	Handles requests up to buffer limit gracefully	Memory boundary tests
Keep-alive connections	Multiple request test	Single connection handles sequential requests	Connection reuse test
10K+ concurrent connections	Load test	All connections serviced, latency acceptable	< 100ms p95 latency
Write backpressure	Slow client simulation	EPOLLOUT registration when buffer full	Flow control test
Idle timeout with keep-alive	Mixed activity test	Connections timeout only after inactivity	Timing test

Comprehensive Verification Suite:

1. **HTTP Protocol Compliance Test**:

```bash
# Test with standard HTTP benchmarking tool
h2load -n 100000 -c 100 -m 100 http://localhost:8080/test


# Or with simpler tool
ab -n 10000 -c 100 http://localhost:8080/
```

Expected: All requests succeed (100% success rate), no protocol errors.

2. **Incremental Parsing Test**:

```bash
# Send request byte-by-byte
python3 -c "
import socket, time
s = socket.create_connection(('localhost', 8080))
request = b'GET / HTTP/1.1\r\nHost: localhost\r\n\r\n'
for byte in request:
    s.send(bytes([byte]))
    time.sleep(0.01)  # 10ms between bytes
print(s.recv(4096))
"
```

Expected: Complete HTTP response received despite slow send.

3. **Keep-alive Connection Test**:

```bash
# Use curl with keep-alive
curl -v http://localhost:8080/ \
     http://localhost:8080/ \
     http://localhost:8080/ \
     --next \
     --keepalive-time 10
```

Expected: Single TCP connection used for all three requests (check with `netstat` or `ss`).

4. **10K Concurrent Connection Benchmark**:

```bash
# Use specialized tool for many connections
slowhttptest -c 10000 -B -i 10 -r 200 -s 8192 -t GET -u http://localhost:8080/ -x 10 -p 3


# Or with wrk for realistic load
wrk -t4 -c10000 -d30s http://localhost:8080/
```

Success Criteria:

- No connection resets or failures
- Memory usage < 500MB for 10K connections
- CPU utilization < 80% on a single core
- P95 latency < 100ms for simple requests

5. **Backpressure Test**:

```bash
# Create slow-reading client
python3 -c "
import socket, time
s = socket.create_connection(('localhost', 8080))
s.send(b'GET /large-response HTTP/1.1\r\nHost: localhost\r\n\r\n')
# Read very slowly
while True:
    data = s.recv(10)  # Only 10 bytes at a time
    if not data: break
    time.sleep(0.1)  # 100ms delay
"
```

Expected: Server handles slow client without blocking, uses write timeouts.

Performance Benchmark Matrix:

Metric	Measurement Method	Target Value	Tool
Connections per second	New connections established	> 10,000/sec	`wrk` or `h2load`
Concurrent connections	Simultaneous active connections	10,000+	`slowhttptest`
Memory per connection	RSS divided by connection count	< 50 KB/conn	`ps`, `pmap`
Request latency	95th percentile response time	< 100 ms	`wrk` with latency stats
CPU utilization	Under 10K connection load	< 80% single core	`top`, `perf`
Error rate	Failed requests	0%	Monitoring tool

Final Milestone 4 Validation Checklist:

- [ ] HTTP/1.1 basic compliance (GET, POST, headers, chunking optional)
- [ ] 10,000 concurrent connections sustained for 5 minutes
- [ ] Memory growth stable (no leaks under load)
- [ ] Slow clients handled gracefully (write timeouts)
- [ ] Keep-alive connections reuse efficiently
- [ ] All resource descriptors (FDs, memory) properly cleaned on disconnect

Implementation Guidance

Technology Recommendations Table

Component	Simple Option (for Learning)	Advanced Option (Production)
Unit Testing Framework	Custom test harness + asserts	**Check** (C unit testing) or **Criterion**
Integration Testing	Socketpair + manual verification	**libtap** for TAP output, CI integration
HTTP Testing	Manual curl/telnet tests	**httest** or custom test server
Load Testing	`ab` (Apache Bench)	**wrk**, **h2load**, or **vegeta**
Profiling	`time`, `strace`, `valgrind`	**perf**, **Heaptrack**, **BPF tools**
Debug Builds	`-g -O0` with asserts	**AddressSanitizer**, **UndefinedBehaviorSanitizer**
Logging	`printf` to stderr	**syslog** or structured JSON logging

Recommended File/Module Structure for Tests

```
event-loop-project/
├── src/
│   ├── event_loop.c
│   ├── timer_wheel.c
│   ├── http_parser.c
│   └── connection.c
├── include/
│   ├── event_loop.h
│   ├── timer_wheel.h
│   └── http_parser.h
├── tests/                    # All test code
│   ├── unit/                # Unit tests
│   │   ├── test_timer_wheel.c
│   │   ├── test_http_parser.c
│   │   ├── test_event_loop.c
│   │   └── test_connection.c
│   ├── integration/         # Integration tests
│   │   ├── test_echo_server.c
│   │   ├── test_http_server.c
│   │   └── test_concurrent.c
│   ├── benchmarks/          # Performance tests
│   │   ├── bench_timer.c
│   │   ├── bench_connections.c
│   │   └── bench_http.c
│   └── utils/               # Test utilities
│       ├── test_helpers.c
│       ├── mock_time.c
│       └── socket_helpers.c
├── scripts/
│   ├── run_tests.sh         # Run all tests
│   ├── load_test.sh         # Load testing script
│   └── valgrind_check.sh    # Memory leak check
└── Makefile
```

Infrastructure Starter Code for Testing

Complete Test Harness Header (`tests/utils/test_helpers.h`):

```c
#ifndef TEST_HELPERS_H

#define TEST_HELPERS_H


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>


// Test assertion macros

#define TEST_ASSERT(cond, msg) \

    do { \

        if (!(cond)) { \

            fprintf(stderr, "FAIL: %s:%d: %s\n", __FILE__, __LINE__, msg); \

            exit(EXIT_FAILURE); \

        } \

    } while (0)


#define TEST_ASSERT_EQ(a, b, msg) \

    TEST_ASSERT((a) == (b), msg)


#define TEST_ASSERT_STR_EQ(a, b, msg) \

    TEST_ASSERT(strcmp(a, b) == 0, msg)


#define TEST_ASSERT_NOT_NULL(ptr, msg) \

    TEST_ASSERT((ptr) != NULL, msg)


// Socketpair helper for testing

int create_test_socketpair(int fds[2]);


// Mock time source for timer tests

typedef uint64_t (*mock_time_getter_t)(void);

void set_mock_time_source(mock_time_getter_t getter);

uint64_t get_mock_time_ms(void);

void advance_mock_time_ms(uint64_t ms);


// HTTP test request generation

char* create_http_request(const char* method, const char* path,
```

```
                        const char* headers, const char* body);

#endif // TEST_HELPERS_H
```

Complete Socketpair Test Helper (`tests/utils/socket_helpers.c`):

```c
#include "test_helpers.h"
#include <sys/socket.h>
#include <fcntl.h>
#include <unistd.h>

int create_test_socketpair(int fds[2]) {
    if (socketpair(AF_UNIX, SOCK_STREAM | SOCK_NONBLOCK, 0, fds) == -1) {
        perror("socketpair");
        return -1;
    }
    return 0;
}

// Helper to make a socket non-blocking
int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) return -1;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}
```

Core Logic Skeleton Code for Tests

Timer Wheel Unit Test Skeleton (`tests/unit/test_timer_wheel.c`):

```c
#include "timer_wheel.h"
#include "test_helpers.h"

// Mock time variable
static uint64_t mock_time = 0;

uint64_t mock_get_time_ms(void) {
    return mock_time;
}

void test_timer_single_fire(void) {
    printf("Testing single timer fire...\n");

    // TODO 1: Create timer wheel with mock time source
    // Hint: Modify timer_wheel_create to accept time source function

    // TODO 2: Add timer with 100ms timeout
    // Use timer_wheel_add with callback that sets a flag

    // TODO 3: Advance time by 50ms and tick wheel
    // Verify timer hasn't fired yet

    // TODO 4: Advance time by another 60ms and tick wheel
    // Verify timer fired exactly once

    // TODO 5: Advance time more and tick again
    // Verify timer doesn't fire again (one-shot)

    // TODO 6: Clean up timer wheel
}

void test_timer_cancellation(void) {
    printf("Testing timer cancellation...\n");

    // TODO 1: Create timer wheel
```

```c
    // TODO 2: Add timer with 100ms timeout

    // TODO 3: Cancel timer using timer_wheel_remove

    // TODO 4: Advance time past expiration and tick

    // TODO 5: Verify callback was NOT invoked

    // TODO 6: Verify memory was freed (use valgrind)
}

void test_timer_performance(void) {
    printf("Testing timer performance...\n");

    // TODO 1: Create timer wheel

    // TODO 2: Time insertion of 10,000 timers with random expirations
    // Use clock_gettime(CLOCK_MONOTONIC) for measurement

    // TODO 3: Time tick operation that expires half the timers

    // TODO 4: Verify O(1) characteristics
    // Insertion time should be constant per timer
}

int main(void) {
    printf("Running timer wheel tests...\n");

    test_timer_single_fire();
    test_timer_cancellation();
    test_timer_performance();

    printf("All timer wheel tests passed!\n");
```

```
    return 0;

}
```

HTTP Parser Test Skeleton (`tests/unit/test_http_parser.c`):

```c
#include "http_parser.h"
#include "test_helpers.h"

void test_http_parser_complete_request(void) {
    printf("Testing complete HTTP request parsing...\n");

    // TODO 1: Create http_parser instance

    // TODO 2: Feed complete HTTP request in one call
    const char* request = "GET /index.html HTTP/1.1\r\n"
                          "Host: localhost:8080\r\n"
                          "User-Agent: Test/1.0\r\n"
                          "\r\n";

    // TODO 3: Verify parser state indicates headers complete

    // TODO 4: Verify parsed method is "GET"

    // TODO 5: Verify parsed path is "/index.html"

    // TODO 6: Verify Host header is "localhost:8080"

    // TODO 7: Reset parser and verify clean state
}

void test_http_parser_incremental(void) {
    printf("Testing incremental HTTP parsing...\n");

    // TODO 1: Create http_parser instance

    // TODO 2: Feed request in two parts
    // Part 1: "GET /index.html HTTP/1.1\r\nHost: local"
    // Part 2: "host:8080\r\n\r\n"
```

```
    // TODO 3: After first part, verify parser needs more data


    // TODO 4: After second part, verify headers complete


    // TODO 5: Verify all headers parsed correctly
}

void test_http_parser_malformed(void) {
    printf("Testing malformed HTTP request handling...\n");

    // TODO 1: Test missing Host header (HTTP/1.1 requirement)

    // TODO 2: Test invalid method characters

    // TODO 3: Test header without colon

    // TODO 4: Test request line with too many parts

    // TODO 5: Verify parser returns appropriate error codes
}

int main(void) {
    printf("Running HTTP parser tests...\n");

    test_http_parser_complete_request();
    test_http_parser_incremental();
    test_http_parser_malformed();

    printf("All HTTP parser tests passed!\n");
    return 0;
}
```

Language-Specific Hints for C Testing

1. **Memory Leak Detection**: Always compile tests with `-g` and run under Valgrind:

```bash
valgrind --leak-check=full --show-leak-kinds=all ./test_timer_wheel
```

2. **Compiler Sanitizers**: Use for additional checks:

```bash
CFLAGS="-fsanitize=address,undefined -fno-omit-frame-pointer"

make test
```

3. **epoll Testing**: When testing epoll code, remember edge-triggered vs level-triggered differences. For tests, level-triggered is often simpler.

4. **Non-blocking I/O Simulation**: To simulate EAGAIN in tests:

```c
// Fill socket buffer to force EAGAIN on write

int send_buffer_size;

socklen_t optlen = sizeof(send_buffer_size);

getsockopt(fd, SOL_SOCKET, SO_SNDBUF, &send_buffer_size, &optlen);

// Send data until write returns EAGAIN
```

5. **Time Mocking**: For deterministic timer tests, override the time source:

```c
// In timer_wheel.h, add:

typedef uint64_t (*time_getter_t)(void);

struct timer_wheel {

    time_getter_t get_time;

    // ... other fields

};
```

Milestone Checkpoint Verification Commands

After Milestone 1:

```bash
# Build and run echo server
make
./event_loop_echo_server 8080 &

# Test with multiple concurrent connections
for i in {1..20}; do
  (echo "Test $i"; sleep 0.1) | nc localhost 8080 &
done
wait

# Check server is still running
kill -0 $! && echo "Server alive - SUCCESS" || echo "Server died - FAIL"
```

After Milestone 2:

```bash
# Build and run server with timeouts
./event_loop_server --port 8080 --idle-timeout 2000 &

# Connect but don't send data
nc localhost 8080 &
NC_PID=$!

# Wait for timeout
sleep 3

# Check connection closed
kill -0 $NC_PID 2>/dev/null && echo "FAIL: Still connected" || echo "PASS: Timeout worked"
```

After Milestone 3:

```bash
# Build and test callback API
make test_callback
./test_callback

# Expected output shows callback execution order:
# [INFO] I/O callback executed
# [INFO] Deferred task executed
# [INFO] Timer callback executed
```

After Milestone 4:

```bash
# Build and run HTTP server
./http_server 8080 &

# Benchmark with Apache Bench
ab -n 10000 -c 100 http://localhost:8080/

# Check for errors and performance
# Success: 100% requests served, latency < 50ms
```

Debugging Tips for Tests

Symptom	Likely Cause	How to Diagnose	Fix
Test passes alone but fails in suite	Shared global state	Add `printf` to show setup/teardown order	Ensure each test completely cleans up
Timer tests flaky (sometimes pass)	Real time source variability	Use mock time source	Implement deterministic time mocking
Memory leak in tests	Missing cleanup in error paths	Run valgrind with `--track-origins=yes`	Add cleanup to all return paths
Socketpair tests hang	Deadlock in event loop	Add logging to event loop states	Check EPOLLOUT registration logic
HTTP parser fails on partial data	Buffer management error	Log parser state after each feed	Ensure buffer pointers updated correctly
Concurrent test fails randomly	Race condition	Add sequence numbering to logs	Use proper synchronization or deterministic test order
Performance test slower than expected	Unoptimized data structures	Profile with `perf record`	Use power-of-2 sizes, cache-friendly layouts

---

Debugging Guide

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Debugging an event-driven system with 10,000+ concurrent connections requires different mental models than traditional sequential programs. Instead of stepping through a linear execution path, you're observing emergent behavior in a complex state machine. Think of debugging this system as **diagnosing a living organism**: you can't stop its heart (the event loop) to examine it, so you must use external monitoring, logging, and selective instrumentation to understand its health. The key insight is that most problems manifest as statistical anomalies—gradual performance degradation, sporadic connection failures, or resource leaks—rather than deterministic crashes.

Diagnostic Mental Model: The Traffic Control Center Analogy

Imagine the event loop as a **traffic control center** for a busy highway system with thousands of vehicles (connections). When something goes wrong:

1. **Traffic Jam (High CPU)**: All controllers (CPU cores) are overwhelmed processing vehicles, but throughput drops.
2. **Stalled Vehicles (Stuck Connections)**: Individual cars aren't moving despite open lanes.
3. **Missing Patrols (Timer Failures)**: Scheduled maintenance vehicles don't arrive at their destinations.

4. **Radio Static (Partial Data)**: Communication arrives garbled or incomplete.

Just as traffic controllers use cameras (logging), vehicle trackers (`strace`), and traffic flow sensors (`perf`), you need multiple observational tools to diagnose which component of the system is malfunctioning and why.

Symptom-Cause-Fix Table

The following table maps observable symptoms to their likely root causes, diagnostic steps, and corrective actions. These patterns emerge from common implementation mistakes in event-driven systems.

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
CPU at 100% with low throughput	1. **Busy-loop on `epoll_wait`** with zero timeout 2. **Missing `EAGAIN` handling** causing infinite retry loops 3. **Level-triggered epoll starvation** where ready events never drain 4. **Timer wheel cascade storm** from incorrectly configured resolution	1. Check `epoll_wait` timeout value in logs 2. Add counters for read/write retries per connection 3. Monitor `EPOLLIN` event frequency for specific FDs 4. Log timer wheel cascade operations per tick	1. Set `epoll_wait` timeout to `timer_wheel_next_expiration()` minimum 2. Implement proper `categorize_error()` for `EAGAIN` / `EWOULDBLOCK` 3. Consider edge-triggered mode or ensure read buffers fully drain 4. Adjust timer wheel resolution to match expected timer frequency
Connections stuck in `STATE_WRITING`	1. **Missing `EPOLLOUT` registration** when write buffer fills 2. **Write buffer management error** where `write_buffer_sent` doesn't advance 3. **Slow client backpressure** without write timeout 4. **Deadlock in deferred task queue** blocking the event loop	1. Trace `event_loop_register_write()` calls for stuck FD 2. Log `write_buffer_used` vs `write_buffer_sent` per connection 3. Check for `write_timeout_callback()` registration 4. Monitor deferred task queue length	1. Call `event_loop_register_write()` when `write_buffer_used > 0` 2. Update `write_buffer_sent` only after successful `safe_write()` 3. Add `write_timeout_callback` with `WRITE_TIMEOUT_MS` 4. Ensure deferred tasks don't block or modify event registrations
Timer not firing or firing late	1. **Timer wheel not being ticked** due to infinite `epoll_wait` 2. **Incorrect timer reference storage** in `connection.timer_ref` 3. **Timer cascading bug** where timers get stuck in upper levels 4. **Clock source mismatch** using wall-clock instead of monotonic	1. Verify `timer_wheel_tick()` is called each event loop iteration 2. Check `timer_ref` is saved after `timer_wheel_add()` 3. Add debug logs for timer movement between wheel levels 4. Compare `timer_wheel_current_time()` with `clock_gettime(CLOCK_MONOTONIC)`	1. Call `timer_wheel_tick()` after processing epoll events 2. Store `timer_wheel_add()` return value in `connection.timer_ref` 3. Implement proper cascade when wheel pointer wraps 4. Use `clock_gettime(CLOCK_MONOTONIC)` for all timer calculations
Memory leak (RSS grows steadily)	1. **Connection objects not freed** on close/error 2. **Buffer expansion without contraction** for large requests 3. **Timer reference leak** from not removing canceled timers 4. **Task queue growth** from unchecked deferred task submission	1. Track `connection_create()` vs `connection_destroy()` counts 2. Monitor `read_buffer` / `write_buffer` size distribution 3. Check `timer_wheel_remove()` calls match `timer_wheel_add()` 4. Log deferred task queue length over time	1. Call `connection_destroy()` in all cleanup paths (timeout, error, normal close) 2. Implement buffer shrinking when usage drops below watermark 3. Pair `timer_wheel_remove()` with every `timer_wheel_add()` on connection cleanup 4. Add max limit to deferred task queue

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
Random connection resets	1. **Missing error handling** for `EPOLLERR` / `EPOLLHUP` 2. **Idle timeout too short** for slow clients 3. **Kernel resource exhaustion** (file descriptors, memory) 4. **TCP keepalive misconfiguration**	1. Check if `EPOLLERR` / `EPOLLHUP` events are processed 2. Monitor connection lifespan vs `IDLE_TIMEOUT_MS` 3. Check `/proc/sys/fs/file-nr` for FD exhaustion 4. Verify TCP keepalive settings with `ss -o`	1. Handle `EPOLLERR` by calling `check_socket_error()` and closing 2. Adjust `IDLE_TIMEOUT_MS` based on expected client behavior 3. Implement connection limits and graceful degradation 4. Set appropriate TCP keepalive with `setsockopt(SO_KEEPALIVE)`
HTTP request parsing failures	1. **Incremental parser state not preserved** across read events 2. **Buffer overflow** from missing size checks 3. **Content-Length mismatch** due to partial body reads 4. **Keep-alive state machine error**	1. Log parser state transitions (`http_parser.state`) 2. Check `buffer_used` against `READ_BUFFER_SIZE` 3. Trace `content_length` vs bytes actually read 4. Monitor `STATE_KEEPALIVE` transitions	1. Ensure `http_parser` persists in `connection.proto_data` 2. Add guard: `if (buffer_used == READ_BUFFER_SIZE) resize_or_close` 3. Track body bytes separately and validate against `content_length` 4. Implement proper keep-alive FSM with timeout reset
Event loop stops processing	1. **Blocking system call** in callback (DNS, file I/O) 2. **Deadlock in shared data structure** 3. **Signal interruption** without proper restart 4. **epoll instance corruption** from closed FD reuse	1. Use `strace` to identify blocking calls 2. Check for locks held during callback execution 3. Verify `EINTR` handling in `epoll_wait` loop 4. Monitor FD numbers vs epoll registration	1. Move all blocking operations to dedicated threads or use async alternatives 2. Never hold locks across callbacks; use deferred modifications 3. Wrap `epoll_wait` in while loop with `errno == EINTR` check 4. Always call `event_loop_unregister_fd()` before `close()`
High latency under load	1. **O(n) operations in critical path** (e.g., linear FD scans) 2. **Excessive system calls** per connection per tick 3. **Memory cache thrashing** from poor data locality 4. **Timer wheel granularity too fine** causing frequent ticks	1. Profile with `perf record` to identify hotspots 2. Count `read()` / `write()` / `epoll_ctl()` calls per second 3. Check cache-miss rates with `perf stat` 4. Measure timer tick frequency vs event rate	1. Replace linear scans with hash tables or pointer arrays 2. Batch writes with `writev()` and defer `epoll_ctl()` modifications 3. Reorganize `connection` struct for cache-line alignment 4. Coarsen timer wheel resolution to match required precision

Tools and Techniques

Effective debugging requires combining passive observation (logging) with active instrumentation (tracing) and statistical analysis (profiling). The event-driven nature means you often need to correlate events across time and multiple connections to identify patterns.

1. Strategic Logging Infrastructure

> **Decision: Structured Logging with Connection Context**
>
> - **Context**: Traditional printf-style logging becomes unmanageable at 10K connections and can itself affect performance through syscall overhead.
> - **Options Considered**:
>   - **Option A**: Verbose per-connection logging (high overhead, hard to filter)
>   - **Option B**: Minimal error-only logging (insufficient for diagnosis)
>   - **Option C**: Configurable structured logging with ring buffers
> - **Decision**: Implement connection-contextual structured logging that can be enabled per-FD or per-error-type.
> - **Rationale**: Structured logs allow post-hoc analysis without runtime overhead when disabled; per-FD enabling lets you "zoom in" on problematic connections without drowning in noise.
> - **Consequences**: Requires log aggregation tools but provides surgical debugging capability; ring buffers prevent unbounded memory growth.

Logging Component	Purpose	Recommended Data
Connection lifecycle	Track FD allocation/deallocation leaks	`timestamp, fd, action(create/destroy), peer_ip:port`
Event dispatch	Diagnose missing or stuck events	`timestamp, fd, events(EPOLLIN/OUT/ERR), callback_invoked`
Timer operations	Debug timeout issues	`timestamp, fd, timer_op(add/remove/fire), scheduled_ms, actual_ms`
Buffer watermarks	Identify backpressure problems	`timestamp, fd, buffer_type(read/write), used_bytes, capacity`
Error categorization	Classify failure patterns	`timestamp, fd, errno, category(ERROR_RETRYABLE/etc), action_taken`

Implementation Pattern: Use a lightweight macro system that compiles to nothing when debugging is disabled:

```c
#define CONN_DEBUG(conn, fmt, ...) \
    if ((conn)->debug_enabled) \
        log_structured(conn->fd, "DEBUG", fmt, ##__VA_ARGS__)
```

2. System Call Tracing with `strace`

`strace` reveals the actual interactions between your process and the kernel, which is essential for diagnosing issues that don't manifest in your program's logic.

Common `strace` Invocations and Their Insights:

Command	Purpose	Key Patterns to Diagnose
`strace -e epoll_wait,epoll_ctl -p PID`	Monitor epoll operations	Missing `epoll_ctl(EPOLL_CTL_MOD)` after write buffer drain
`strace -e read,write -p PID 2>&1 | grep EAGAIN`		Identify retry storms
`strace -c -p PID -S calls`	Count system call distribution	High `epoll_ctl` count suggesting suboptimal registration patterns
`strace -e accept4,close -p PID`	Track connection lifecycle	`close()` calls not matching `accept4()` indicating leaks
`strace -yy -e poll,epoll_wait,read,write -p PID`	See FD numbers and types	Correlations between FD numbers and operations

Example Diagnosis Flow:

1. **Symptom**: High CPU usage
2. **Command**: `strace -c -p $(pgrep server) -S time`
3. **Observation**: 80% of time spent in `epoll_wait` with 0 timeout
4. **Inference**: Event loop is busy-waiting because timeout calculation is wrong
5. **Fix**: Ensure `epoll_wait` timeout uses `timer_wheel_next_expiration()`

3. Performance Profiling with `perf`

`perf` provides CPU-level insights that complement the system-call view from `strace`. It's particularly valuable for identifying cache inefficiencies and hotspots in your data structures.

Key `perf` Subcommands for Event Loop Analysis:

Subcommand	Target Issue	Interpretation Guide
`perf top -p PID`	CPU hotspots in real-time	Look for `timer_wheel_tick` or `connection_read` dominating
`perf record -g -p PID` + `perf report`	Call graph analysis	Identify deep call stacks from simple operations
`perf stat -e cache-misses,L1-dcache-loads -p PID`	Cache efficiency	High cache-miss rate suggests poor data locality in `connection` array
`perf trace --syscalls=epoll_wait,read,write -p PID`	Combined syscall + timing	See duration of each system call within event loop tick

Mental Model for `perf` Output: Think of the event loop as a **factory assembly line**. `perf` tells you which workstations (functions) have the longest queues (CPU time), which parts are waiting at shipping (cache misses), and where cross-station travel is inefficient (branch mispredictions).

4. Runtime Inspection with `/proc` Filesystem

The Linux `/proc` filesystem provides real-time visibility into your process's kernel-state relationships.

Critical `/proc` Entries for Event Loop Debugging:

Path	Information	Diagnostic Value
`/proc/PID/fd`	Open file descriptors	Count and types of FDs; look for unexpected growth
`/proc/PID/fdinfo/FD`	Per-FD epoll status	`pos: 0 flags: 02004002` shows O_NONBLOCK and other flags
`/proc/PID/status`	Memory and thread stats	`VmRSS` for memory leaks, `Threads: 1` confirms single-threaded
`/proc/PID/stack`	Kernel stack trace	Reveals if stuck in blocking syscall
`/proc/sys/fs/file-nr`	System-wide FD usage	Diagnose FD exhaustion across all processes
`/proc/net/tcp`	TCP connection states	Verify `ESTABLISHED` count matches your connection count

Practical Walkthrough: When connections appear stuck:

1. `ls -l /proc/PID/fd | wc -l` → Count open FDs
2. `cat /proc/PID/fdinfo/123` → Check flags for FD 123
3. `grep -A2 -B2 "123:" /proc/net/tcp` → Verify kernel TCP state
4. **Discovery**: FD 123 shows `tcp` state `CLOSE_WAIT` but your program still has it registered
5. **Root Cause**: Missing `EPOLLHUP` handling or not checking `read() == 0`

5. Network Diagnostic Tools

Network-level issues often masquerade as application bugs in high-concurrency servers.

Tool	Command Example	Diagnostic Purpose
`ss` (socket statistics)	`ss -tlnp	grep PID`	
	`ss -to state established dst :8080`	TCP timers and options for established connections
`netstat`	`netstat -tpn	grep PID`	
`tcpdump`	`tcpdump -i lo -n port 8080 -c 10`	Capture packets to verify protocol compliance
`nc` (netcat)	`nc -zv localhost 8080`	Quick connectivity test
`socat`	`socat - TCP4:localhost:8080`	Manual protocol interaction

Network Debugging Mindset: When a connection behaves strangely, ask: "Is this my program's fault, or is the network/TCP stack doing something unexpected?" Use `ss -to` to see TCP retransmits, zero windows, or keepalive expirations that explain application-level symptoms.

Common Pitfalls in Event Loop Debugging

⚠️ **Pitfall: Heisenbugs from Debug Logging** Adding `printf` to diagnose a problem changes timing enough to make the bug disappear. This is especially common in race conditions around timer expiration and event registration.

- **Why it's wrong**: The debugging itself becomes a confounding variable.
- **Fix**: Use atomic counters instead of I/O in hot paths, or write to memory-mapped ring buffers that have minimal timing impact.

⚠️ **Pitfall: Assuming Linear Causality** In event-driven systems, symptom and cause may be separated by thousands of event loop ticks. A connection drop at time T might be caused by a memory allocation failure at time T-1000.

- **Why it's wrong**: Looking only at immediate context misses root causes.
- **Fix**: Implement correlation IDs that track chains of events, and use timeline visualization tools.

⚠️ **Pitfall: Overlooking Kernel Resources** Your application logic may be perfect, but the kernel may be out of TCP buffers, file descriptors, or ephemeral ports.

- **Why it's wrong**: Application-level debugging shows no bugs, but the system still fails.
- **Fix**: Monitor `/proc` metrics alongside application metrics, set `ulimit -n` appropriately, and implement graceful degradation.

⚠️ **Pitfall: Statistically Rare Events** A bug that affects 1 in 10,000 connections might not appear in small-scale testing but will cause periodic failures in production.

- **Why it's wrong**: "It works on my machine" with 100 connections doesn't guarantee correctness at scale.
- **Fix**: Use property-based testing that generates edge cases, and run load tests that exceed target concurrency by 2x.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debug Logging	Custom macro with compile-time disable	Systemd journal with structured fields
Tracing	`strace` for syscall-level debugging	BPF/eBPF for custom in-kernel tracing
Profiling	`perf` with sampling mode	Intel VTune with cache-aware optimizations
Memory Analysis	Custom allocation trackers	Valgrind Massif for heap profiling
Network Inspection	`ss`, `tcpdump` textual analysis	Wireshark with custom dissectors

B. Recommended Debug Infrastructure Code Structure

```
event-loop-project/
├── src/
│   ├── debug/              # Debugging infrastructure
│   │   ├── log.c           # Structured logging implementation
│   │   ├── counters.c      # Atomic performance counters
│   │   ├── ringbuf.c       # Lock-free ring buffer for trace events
│   │   └── debug.h
│   ├── event_loop.c        # Main event loop (with debug hooks)
│   └── ...
├── tools/
│   ├── analyze_logs.py     # Parse and visualize structured logs
│   ├── fd_monitor.sh       # Monitor FD leaks over time
│   └── stress_test.c       # Connection stress tester
└── scripts/
    └── debug_profile.sh    # Automated perf/strace collection
```

C. Infrastructure Starter Code: Structured Logging

```c
// src/debug/log.c - Complete structured logging implementation

#include <stdio.h>
#include <stdarg.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>
#include "debug.h"

// Thread-local buffer to avoid locking in logging hot path
__thread char log_buffer[1024];
__thread struct timeval log_tv;

// Simple ring buffer for last N log entries (lock-free for single producer)
typedef struct {
    char entries[MAX_LOG_ENTRIES][256];
    unsigned int head;
    unsigned int tail;
    unsigned int count;
} log_ringbuf_t;

static log_ringbuf_t ringbuf;
static int debug_enabled = 0;
static FILE *log_stream = NULL;

void debug_init(const char *logfile) {
    if (logfile && strcmp(logfile, "-") != 0) {
        log_stream = fopen(logfile, "a");
        if (!log_stream) {
            perror("Failed to open log file");
            log_stream = stderr;
        }
    } else {
```

```
        log_stream = stderr;
    }

    // Enable debugging if DEBUG env var is set
    if (getenv("DEBUG") != NULL) {
        debug_enabled = 1;
    }
}

void log_structured(int fd, const char *category, const char *fmt, ...) {
    if (!debug_enabled) return;

    gettimeofday(&log_tv, NULL);
    va_list args;
    va_start(args, fmt);

    // Format: timestamp_ms fd category message
    int offset = snprintf(log_buffer, sizeof(log_buffer),
                          "%lu.%06lu %d %s ",
                          log_tv.tv_sec, log_tv.tv_usec,
                          fd, category);

    if (offset < sizeof(log_buffer)) {
        vsnprintf(log_buffer + offset, sizeof(log_buffer) - offset, fmt, args);
    }

    va_end(args);

    // Write to stream
    fprintf(log_stream, "%s\n", log_buffer);
    fflush(log_stream);

    // Also store in ring buffer for later inspection
    unsigned int next_head = (ringbuf.head + 1) % MAX_LOG_ENTRIES;
```

```
    if (next_head != ringbuf.tail) {
        strncpy(ringbuf.entries[ringbuf.head], log_buffer, 255);
        ringbuf.entries[ringbuf.head][255] = '\0';
        ringbuf.head = next_head;
        if (ringbuf.count < MAX_LOG_ENTRIES) ringbuf.count++;
    }
}

const char **debug_get_recent_logs(int *count) {
    static const char *result[MAX_LOG_ENTRIES];
    *count = ringbuf.count;

    unsigned int idx = ringbuf.tail;
    for (int i = 0; i < ringbuf.count; i++) {
        result[i] = ringbuf.entries[idx];
        idx = (idx + 1) % MAX_LOG_ENTRIES;
    }

    return result;
}

// src/debug/debug.h
#ifndef DEBUG_H
#define DEBUG_H

#define MAX_LOG_ENTRIES 10000

void debug_init(const char *logfile);
void log_structured(int fd, const char *category, const char *fmt, ...);
const char **debug_get_recent_logs(int *count);

// Convenience macros
#define LOG_EVENT(fd, fmt, ...) \
    log_structured(fd, "EVENT", fmt, ##__VA_ARGS__)

#define LOG_TIMER(fd, fmt, ...) \
```

```
    log_structured(fd, "TIMER", fmt, ##__VA_ARGS__)


#define LOG_ERROR(fd, fmt, ...) \
    log_structured(fd, "ERROR", fmt, ##__VA_ARGS__)


#define LOG_BUFFER(fd, fmt, ...) \
    log_structured(fd, "BUFFER", fmt, ##__VA_ARGS__)

#endif
```

D. Core Debug Skeleton: Connection State Tracker

```c
// src/connection.c - Add debugging hooks to connection management

// TODO 1: Add a debug_enabled flag to struct connection
// TODO 2: In connection_create(), initialize debug_enabled based on
//         random sampling (e.g., 1% of connections) or specific conditions
// TODO 3: Add LOG_EVENT calls at critical state transitions:
//         - When changing connection.state
//         - When registering/deregistering from epoll
//         - When adding/removing timers
// TODO 4: Implement connection_inspect(fd) function that prints
//         current state, buffer usage, and recent events
// TODO 5: Add a control socket or signal handler that toggles
//         debug on specific connections at runtime
// TODO 6: Track statistics (min/max/avg) for buffer sizes,
//         read/write latencies, and state durations

// Example TODO expansion for connection state transition:
void connection_set_state(struct connection *conn, int new_state) {
    // TODO 1: Log the state transition with timestamp
    // LOG_EVENT(conn->fd, "state %d -> %d", conn->state, new_state);

    // TODO 2: Update duration tracking for the old state
    // uint64_t duration = current_time - conn->state_entered_time;
    // update_stats(conn->state, duration);

    // TODO 3: Set new state and record entry time
    conn->state = new_state;
    // conn->state_entered_time = current_time;

    // TODO 4: If debug is enabled, record this in connection's event ring buffer
}
```

E. Language-Specific Hints for C

- **Atomic Counters**: Use `__atomic_add_fetch(&counter, 1, __ATOMIC_RELAXED)` for performance counters without locking.

- **Backtrace on Signal**: Install handler with `sigaction(SIGSEGV, ...)` and use `backtrace()` from `<execinfo.h>` to get stack traces.
- **FD Status Checking**: Use `fcntl(fd, F_GETFL)` to verify `O_NONBLOCK` is set, and `getsockopt(fd, SOL_SOCKET, SO_ERROR, ...)` to check for pending errors.
- **Memory Sanitizers**: Compile with `-fsanitize=address,undefined` for runtime memory error detection during development.
- **Static Analysis**: Use `clang-tidy` with checks for null pointer dereferences, resource leaks, and concurrency issues.

F. Milestone Debugging Checkpoints

Milestone 1 Completion Check:

```bash
# Start server with debug logging
DEBUG=1 ./server 8080 &
SERVER_PID=$!

# Make 100 rapid connections
for i in {1..100}; do (echo "test $i" | nc -N localhost 8080 &) done

# Check for errors
grep -i "error\|EAGAIN\|block" server.log | head -20

# Expected: No "block" warnings, occasional EAGAIN is normal
# If stuck: Use `strace -p $SERVER_PID -e accept4,read,write`
```

Milestone 2 Timer Verification:

```bash
# Start server with timer debug
DEBUG_TIMERS=1 ./server 8080 &
SERVER_PID=$!

# Connect but send no data (should timeout in 30s)
nc localhost 8080 &
NC_PID=$!

# Monitor timer operations
tail -f server.log | grep -i "timer"

# After 30s, connection should close
# If not: Check timer_wheel_tick() is called, verify idle timer registration
```

Milestone 3 Callback Debugging:

```bash
# Test deferred task queue
DEBUG_TASKS=1 ./server 8080 &

# Use test client that triggers multiple callbacks
./test_client --callbacks=1000

# Verify: No callbacks modify event registration during iteration
# Check log for "deferred" vs "immediate" task patterns
```

Milestone 4 HTTP Stress Test:

```bash
# Benchmark with ApacheBench
ab -n 10000 -c 1000 http://localhost:8080/test

# Monitor during test
watch -n1 "ss -tlnp | grep 8080"

# Expected: Steady connection count, no errors in logs
# Debug: If connections grow unbounded, check keep-alive timeout handling
```

G. Debugging Tips Table

Symptom	How to Diagnose	Quick Fix
Server accepts then immediately closes	`strace -e accept4,close -p PID`	Ensure `set_nonblocking()` on accepted sockets
Data corruption in HTTP responses	Hex dump of buffers before/after write	Check for buffer overrun in `http_generate_response()`
Some connections never get data	`strace -e epoll_ctl -p PID	grep MOD`	
Server stops after many connections	`cat /proc/PID/limits` and `ls /proc/PID/fd	wc -l`	
Response times increase linearly with connections	`perf stat -e cache-misses -p PID`	Optimize `connection` struct layout for cache locality
Timers fire in bursts, not smoothly	Log timer wheel pointer vs current time	Fix `timer_wheel_tick()` to process multiple slots if behind

Future Extensions

> **Milestone(s):** Beyond core milestones (potential next steps after completing Milestone 4)

This section explores how the single-threaded event loop architecture can be extended to tackle more complex scenarios and higher performance requirements. The foundational design—with its clean separation of I/O multiplexing, timer management, and protocol handling—provides a solid base for several advanced enhancements. Think of the current event loop as a high-performance single-

engine aircraft; these extensions represent potential upgrades to multi-engine configurations, improved navigation systems, and specialized payload capabilities.

Multi-threaded Reactor and Protocol Upgrades

The single-threaded reactor pattern achieves remarkable efficiency for I/O-bound workloads but faces limitations with CPU-intensive tasks or when maximum hardware utilization is required. Extending the architecture to leverage multiple CPU cores while preserving the event-driven model is a natural progression.

Mental Model: The Post Office with Specialized Departments

Imagine our single-threaded event loop as a small-town post office with one exceptionally efficient postal worker. This worker handles everything: sorting mail, operating the counter, and loading trucks. As mail volume grows, we could:

1. **Open Multiple Identical Post Offices (Multi-Process)**: Run independent event loops in separate processes, each with its own epoll instance and connection set.
2. **Add Specialized Workers (Thread Pool)**: Keep the main postal worker for sorting but hire specialized workers for heavy tasks like package processing.
3. **Create Regional Sorting Centers (Multi-Threaded Reactor)**: Build a larger facility where multiple workers collaboratively sort mail using shared infrastructure.

The third approach—multi-threaded reactor—provides the best scalability for our server architecture while maintaining the event-driven paradigm.

Multi-Threaded Reactor Pattern

Decision: Multi-Threaded Reactor with Worker Threads

> **Decision: Threading Model for CPU-Intensive Work**
>
> - **Context**: The single-threaded event loop can efficiently handle 10K+ connections but saturates a single CPU core. HTTP parsing, compression, and business logic remain CPU-bound. We need to utilize multiple CPU cores without introducing blocking or excessive context switching.
> - **Options Considered**:
>   1. **Multi-Process (Prefork)**: Run multiple independent event loops in separate processes, balancing connections via SO_REUSEPORT.
>   2. **Thread Pool for Blocking Operations**: Maintain single-threaded reactor but offload CPU-intensive tasks to a worker thread pool.
>   3. **Multi-Threaded Reactor**: Run multiple reactor threads, each with its own epoll instance, sharing the listening socket and load-balancing connections.
> - **Decision**: Hybrid approach combining Option 2 and 3—multi-threaded reactor with protocol-specific worker threads for CPU-heavy operations.
> - **Rationale**: Pure multi-process wastes memory (duplicate buffers) and complicates shared state. Pure thread pool still bottlenecks on single epoll thread. Hybrid approach maximizes CPU utilization while keeping I/O dispatch efficient. Each reactor thread can handle its own set of connections, with expensive operations dispatched to shared worker pools.
> - **Consequences**: Enables horizontal scaling across CPU cores, adds complexity for shared data synchronization, requires careful connection affinity to avoid thread contention.

Option	Pros	Cons	Chosen?
Multi-Process (Prefork)	Simple, strong isolation, crash doesn't affect all connections	Memory overhead (per-process buffers), difficult shared state, load balancing less precise	No
Thread Pool for Blocking Ops	Maintains simple single-threaded I/O model, good for isolated CPU tasks	Single epoll thread remains bottleneck, all I/O still serialized	Partial
Multi-Threaded Reactor	Scales I/O handling across cores, maximizes hardware utilization	Complex synchronization, potential for thread contention	**Yes** (with worker pool)

The multi-threaded reactor architecture introduces several new components and modifications to the existing design:

Component	New Responsibility	Integration with Existing System
Reactor Thread Pool	Multiple threads running independent `event_loop_run()` instances	Share listening socket via `SO_REUSEPORT` or round-robin `accept()`
Load Balancer	Distributes new connections across reactor threads	Integrated into modified `accept()` logic using thread-local queues
Worker Thread Pool	Executes CPU-intensive callbacks (HTTP parsing, compression)	Receives tasks via thread-safe queue from reactor threads
Synchronization Primitives	Protects shared timer wheel and statistics	Uses atomic operations and fine-grained locking where necessary

The connection state machine expands to handle cross-thread operations:

Current State	Event	Next State	Actions Taken
`STATE_READING`	Data requires CPU-intensive parsing	`STATE_PROCESSING`	1. Suspend read events on reactor thread 2. Enqueue parsing task to worker pool 3. Store connection reference in pending tasks map
`STATE_PROCESSING`	Worker completes parsing	`STATE_WRITING` or `STATE_READING`	1. Worker thread signals completion via pipe 2. Reactor thread resumes event registration 3. Apply parsed results to connection
`STATE_ACCEPTED`	Load balancer decision	`STATE_READING` (on different thread)	1. Main acceptor thread selects target reactor thread 2. Passes socket via Unix domain socket or eventfd 3. Target thread registers socket with its epoll instance

Common Pitfalls in Multi-Threaded Extension:

⚠️ **Pitfall: Shared Timer Wheel Contention**

- **Description**: Having all reactor threads share a single `timer_wheel` structure causes lock contention on every tick and timer operation.
- **Why it's wrong**: Timer operations should be O(1); adding locks makes them unpredictable and creates scaling bottlenecks.
- **Fix**: Use thread-local timer wheels with a shared "next expiration" coordinator, or implement a lock-free hierarchical timer wheel with atomic operations.

⚠️ **Pitfall: Cross-Thread Connection Handoff Blocking**

- **Description**: Passing sockets between threads using blocking writes to pipes or sockets.
- **Why it's wrong**: If the receiving thread's pipe buffer fills, the sending thread blocks, potentially stalling connection acceptance.
- **Fix**: Use `eventfd` with non-blocking writes and edge-triggered epoll monitoring, or implement a lock-free ring buffer for socket descriptors.

⚠️ **Pitfall: Worker Pool Task Queue Backpressure**

- **Description**: Reactor threads enqueue parsing tasks faster than workers can process them, causing unbounded queue growth.
- **Why it's wrong**: Memory exhaustion and increased latency as tasks wait in queue.
- **Fix**: Implement bounded task queues with backpressure signaling—when queue reaches high watermark, reactor threads temporarily pause parsing and buffer data locally.

Protocol Upgrades: WebSockets and TLS

The event loop's protocol-agnostic design makes it an excellent foundation for additional protocols beyond HTTP/1.1.

Decision: Protocol Handler Plug-in Architecture

> **Decision: Extensible Protocol Support**
>
> - **Context**: The event loop currently handles HTTP/1.1, but real applications need WebSockets for real-time communication and TLS for secure connections. We need a design that accommodates multiple protocols without rewriting core components.
> - **Options Considered**:
>   1. **Monolithic Protocol Switch**: Extend `http_parser` with flags and states for WebSockets and TLS handshakes.
>   2. **Protocol-Specific Connection Types**: Create separate `websocket_connection` and `tls_connection` structures with their own state machines.
>   3. **Protocol Handler Interface**: Define a clean interface that all protocols implement, with the connection delegating to the active protocol handler.
> - **Decision**: Protocol Handler Interface with connection upgrade capability.
> - **Rationale**: Monolithic approach becomes unwieldy and violates single responsibility. Separate connection types duplicate I/O buffer logic. The interface approach keeps protocol logic isolated while sharing common infrastructure. Connection upgrade (HTTP → WebSockets, plain → TLS) becomes a simple handler swap.
> - **Consequences**: Clean separation of concerns, easy to add new protocols, slight indirection overhead, requires careful design of handler lifecycle.

The protocol handler interface would define these core operations:

Method	Parameters	Returns	Description
`handle_read`	`conn struct connection*`	`int` (bytes processed)	Process incoming data from read buffer, update connection state
`handle_write`	`conn struct connection*`	`int` (bytes sent)	Prepare and send data from write buffer
`handle_timeout`	`conn struct connection*`	`void`	Protocol-specific idle timeout logic
`can_upgrade_to`	`proto_name const char*`	`int` (1 if possible)	Check if protocol can upgrade to another protocol
`upgrade`	`conn struct connection*`, `new_handler void*`	`int` (success)	Transition connection to new protocol handler

WebSockets Implementation Strategy:

WebSockets begin as HTTP connections with an upgrade request, making them ideal for the protocol handler approach:

1. **HTTP Handler Phase**: Normal HTTP parsing until complete headers
2. **Upgrade Detection**: `http_parser` detects `Connection: Upgrade` and `Upgrade: websocket`
3. **Handler Swap**: Connection's protocol handler changes from `http_handler` to `websocket_handler`
4. **Handshake Response**: Original handler generates HTTP 101 response
5. **WebSocket Frame Processing**: New handler processes binary frames using its own state machine

The WebSocket handler maintains additional state:

Field	Type	Description
`frame_state`	`enum`	Current frame parsing state (OPCODE, LENGTH, MASK, PAYLOAD)
`frame_opcode`	`uint8_t`	Type of WebSocket frame (TEXT, BINARY, CLOSE, etc.)
`frame_payload_length`	`uint64_t`	Total expected payload length
`frame_bytes_received`	`uint64_t`	Bytes received for current frame
`fragmentation_buffer`	`char*`	Buffer for fragmented frames
`mask_key`	`uint32_t`	XOR mask for client-to-server frames
`close_code`	`uint16_t`	Close status code for graceful shutdown

TLS/SSL Integration Strategy:

TLS adds encryption/decryption layers between the socket and application protocol. The most practical approach is the **Bump-in-the-Stack** pattern:

1. **TLS Handler Wrapping**: Create a `tls_handler` that wraps another protocol handler (HTTP or WebSocket)
2. **Bio Callbacks**: Use OpenSSL's BIO abstraction with custom read/write callbacks that use the connection's buffers
3. **Handshake State Machine**: TLS handler manages the multi-phase handshake before delegating to the wrapped protocol
4. **Record Layer Buffering**: TLS requires reading/writing complete records, necessitating additional buffering

The sequence for a TLS upgrade looks like:

1. Client connects on HTTPS port (typically 443)
2. Listening socket accepts and creates connection with `tls_handler` as initial protocol
3. TLS handler performs handshake via OpenSSL library calls
4. During handshake, TLS handler uses `connection_read()` and `connection_write()` for network I/O
5. After handshake completion, TLS handler swaps itself for `http_handler` but remains as a wrapper
6. All subsequent `handle_read`/`handle_write` calls go through TLS decryption/encryption first

Performance Considerations for Protocol Upgrades:

Protocol Aspect	Performance Impact	Mitigation Strategy
TLS Handshake	CPU-intensive asymmetric crypto	Session resumption, TLS tickets, worker thread offloading
WebSocket Frame Parsing	Per-message overhead	Batch small frames, use binary protocols where possible
Protocol Switching	Memory allocation/free	Pool protocol handler objects, reuse buffers
TLS Record Buffering	Additional memory copies	Use scatter/gather I/O (writev/readv) with OpenSSL

Common Pitfalls in Protocol Implementation:

⚠️ **Pitfall: Half-Implemented WebSocket Close Handshake**

- **Description**: Implementing WebSocket data frames but neglecting proper close frame exchange and connection teardown.
- **Why it's wrong**: Clients may hang waiting for close response, and servers may not release resources properly.
- **Fix**: Implement complete WebSocket state machine including close frame sending/receiving, with proper timeout for unresponsive peers.

⚠️ **Pitfall: TLS Renegotiation without Backpressure**

- **Description**: Allowing TLS renegotiation during data transfer without pausing application data flow.
- **Why it's wrong**: Interleaved handshake and application records cause protocol violations and decryption failures.
- **Fix**: Buffer application data during renegotiation, or disable renegotiation entirely for security and simplicity.

⚠️ **Pitfall: Mixed Protocol Port Sharing**

- **Description**: Running HTTP and HTTPS on the same port using opportunistic TLS (detecting TLS client hello).
- **Why it's wrong**: Complexity increases, and some clients may timeout during detection phase.
- **Fix**: Use separate ports (80/443) or implement proper ALPN/SNI with a single TLS-wrapped handler that can fall back to plain HTTP only if explicitly configured.

Load Balancing Across Reactor Threads

For the multi-threaded reactor to scale effectively, connections must be distributed evenly across threads. Several load balancing strategies are possible:

Strategy	Implementation	Pros	Cons	Best For
Accept Lock	Global mutex around `accept()`	Simple, guaranteed even distribution	Contention point, doesn't consider thread load	Small core counts (2-4)
SO_REUSEPORT	Multiple listeners on same port, each thread binds independently	Kernel-level balancing, no contention	Uneven distribution under some workloads, connection skew	Linux 3.9+, high connection rate
Thread Local Queue	Dedicated acceptor thread passes sockets via queue	Customizable balancing logic	Extra thread overhead, queue management	Heterogeneous workloads
Least Connections	Acceptor tracks per-thread counts, assigns to least busy	Good load distribution	Requires shared atomic counters	Mixed connection durations

A hybrid approach often works best: use `SO_REUSEPORT` for raw connection acceptance speed, complemented by work-stealing for load rebalancing when threads become unevenly loaded.

> **Key Insight**: The optimal threading model depends on workload characteristics. For many small, short-lived connections (HTTP/1.0), `SO_REUSEPORT` works well. For persistent connections with variable processing needs (WebSockets with messaging), work-stealing between reactor threads provides better balance.

Implementation Guidance

While the full implementation of these extensions is beyond the project's scope, this guidance provides starting points for exploration.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Threading Library	POSIX Threads (`pthread`)	Custom thread pool with work-stealing
TLS/SSL	OpenSSL 1.1+ with basic configuration	OpenSSL with async mode, or BoringSSL for simpler API
WebSockets	Custom RFC 6455 implementation	`libwebsockets` library with integration layer
Load Balancing	`SO_REUSEPORT` with round-robin accept	Intel DPDK or io_uring for kernel bypass
Synchronization	`pthread_mutex_t` and `pthread_cond_t`	Atomic operations with futex for low contention
Inter-Thread Communication	Unix pipes or `eventfd`	Lock-free ring buffers (SPSC/MPSC)

B. Recommended File/Module Structure:

```
event-loop-epoll/
├── src/
│   ├── core/                   # Original single-threaded core
│   │   ├── event_loop.c        # Now becomes base reactor implementation
│   │   ├── timer_wheel.c
│   │   ├── connection.c
│   │   └── task_queue.c
│   ├── threading/              # Multi-threading extensions
│   │   ├── reactor_pool.c      # Manages multiple reactor threads
│   │   ├── worker_pool.c       # CPU worker thread pool
│   │   ├── load_balancer.c     # Connection distribution logic
│   │   └── synchronization.c   # Atomic ops, locks, barriers
│   ├── protocols/              # Protocol implementations
│   │   ├── protocol.c          # Base protocol interface
│   │   ├── http_protocol.c     # HTTP handler (refactored from http_parser.c)
│   │   ├── websocket_protocol.c # WebSocket RFC 6455 implementation
│   │   └── tls_protocol.c      # TLS wrapper handler
│   └── main.c                  # Entry point with configuration
```

C. Infrastructure Starter Code (Multi-Threaded Reactor Skeleton):

```c
/* threading/reactor_pool.h */

#ifndef REACTOR_POOL_H

#define REACTOR_POOL_H


#include "../core/event_loop.h"

#include <pthread.h>


typedef struct reactor_thread {

    pthread_t thread;

    struct event_loop* loop;

    int thread_id;

    int event_fd;  // For cross-thread wakeups

    struct connection_queue* pending_connections;

    // Statistics

    uint64_t connections_handled;

    uint64_t events_processed;

} reactor_thread;


typedef struct reactor_pool {

    int thread_count;

    reactor_thread* threads;

    int next_thread_idx;  // For simple round-robin

    pthread_mutex_t accept_mutex;

    struct timer_wheel* shared_timer_wheel;  // Optional shared timer

    int running;

} reactor_pool;


// Create pool with specified number of threads

reactor_pool* reactor_pool_create(int thread_count, int max_events_per_thread);


// Start all reactor threads

int reactor_pool_start(reactor_pool* pool);


// Stop all reactor threads gracefully

void reactor_pool_stop(reactor_pool* pool);


// Distribute a new connection to a reactor thread
```

```
int reactor_pool_dispatch_connection(reactor_pool* pool, int client_fd);

// Get thread with fewest active connections
reactor_thread* reactor_pool_get_least_loaded(reactor_pool* pool);

#endif /* REACTOR_POOL_H */
```

D. Core Logic Skeleton Code (Protocol Handler Interface):

C

```c
/* protocols/protocol.h */

#ifndef PROTOCOL_H

#define PROTOCOL_H


#include "../core/connection.h"


// Protocol handler function signatures

typedef int (*protocol_handle_read_t)(struct connection* conn);

typedef int (*protocol_handle_write_t)(struct connection* conn);

typedef void (*protocol_handle_timeout_t)(struct connection* conn);

typedef int (*protocol_can_upgrade_t)(struct connection* conn, const char* proto_name);

typedef int (*protocol_upgrade_t)(struct connection* conn, void* new_handler);


// Base protocol handler interface

typedef struct protocol_handler {

    const char* name;

    protocol_handle_read_t handle_read;

    protocol_handle_write_t handle_write;

    protocol_handle_timeout_t handle_timeout;

    protocol_can_upgrade_t can_upgrade;

    protocol_upgrade_t upgrade;



    // Protocol-specific data

    void* proto_data;



    // For chaining (e.g., TLS wraps HTTP)

    struct protocol_handler* inner_handler;

} protocol_handler;


// Common protocol handlers

extern protocol_handler* http_protocol_create(void);

extern protocol_handler* websocket_protocol_create(void);

extern protocol_handler* tls_protocol_create(protocol_handler* inner);


// Connection protocol management

void connection_set_protocol(struct connection* conn, protocol_handler* handler);
```

```
protocol_handler* connection_get_protocol(struct connection* conn);


#endif /* PROTOCOL_H */
```

```c
/* protocols/tls_protocol.c - Partial Implementation */

#include "protocol.h"

#include <openssl/ssl.h>

#include <openssl/err.h>


typedef struct tls_protocol_data {

    SSL* ssl;

    BIO* rbio;  // Read BIO

    BIO* wbio;  // Write BIO

    int handshake_complete;

    int handshake_state;

    char* pending_decrypted;  // Decrypted data pending for inner handler

    int pending_len;

} tls_protocol_data;


static int tls_handle_read(struct connection* conn) {

    protocol_handler* handler = connection_get_protocol(conn);

    tls_protocol_data* tls_data = (tls_protocol_data*)handler->proto_data;



    // TODO 1: Read raw data from connection's read_buffer into SSL

    // TODO 2: If handshake incomplete, call SSL_do_handshake()

    // TODO 3: If handshake complete and data available, call SSL_read()

    // TODO 4: Store decrypted data in pending_decrypted for inner handler

    // TODO 5: If inner_handler exists, call its handle_read with decrypted data

    // TODO 6: If SSL wants to write (handshake or renegotiation), add to write_buffer

    // TODO 7: Return bytes processed from connection's read_buffer

    return -1; // Replace with actual implementation

}


static int tls_handle_write(struct connection* conn) {

    protocol_handler* handler = connection_get_protocol(conn);

    tls_protocol_data* tls_data = (tls_protocol_data*)handler->proto_data;



    // TODO 1: If inner_handler exists and has data to send, call SSL_write()

    // TODO 2: Get encrypted data from wbio and add to connection's write_buffer
```

```
    // TODO 3: Send write_buffer contents via connection_write()
    // TODO 4: Return bytes successfully written to socket
    return -1; // Replace with actual implementation
}

protocol_handler* tls_protocol_create(protocol_handler* inner) {
    protocol_handler* handler = malloc(sizeof(protocol_handler));
    if (!handler) return NULL;

    handler->name = "tls";
    handler->handle_read = tls_handle_read;
    handler->handle_write = tls_handle_write;
    handler->handle_timeout = NULL; // Use default
    handler->can_upgrade = NULL; // TLS is usually initial protocol
    handler->upgrade = NULL; // Cannot upgrade from TLS
    handler->inner_handler = inner;

    // TODO: Initialize OpenSSL context if not already done
    // TODO: Create SSL object and BIOs
    // TODO: Allocate and initialize tls_protocol_data

    return handler;
}
```

E. Language-Specific Hints (C with POSIX and OpenSSL):

- **Thread-Local Storage**: Use `__thread` keyword or `pthread_key_create()` for thread-local variables like error buffers or temporary workspaces.
- **Atomic Operations**: For counters shared between threads, use GCC's `__atomic_*` builtins or C11 `stdatomic.h` if available.
- **OpenSSL Thread Safety**: Call `CRYPTO_set_locking_callback()` to provide thread-safe locking for OpenSSL, or use OpenSSL 1.1.0+ which has built-in thread support.
- **Non-Blocking OpenSSL**: Configure SSL with `SSL_set_mode(ssl, SSL_MODE_ENABLE_PARTIAL_WRITE | SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER)` for non-blocking behavior.
- **Eventfd for Wakeups**: Use `eventfd()` instead of pipes for inter-thread signaling—it's lighter weight and edge-triggered friendly.
- **CPU Affinity**: Consider `pthread_setaffinity_np()` to pin reactor threads to specific CPU cores, reducing cache invalidation.

F. Milestone Checkpoint for Extensions:

After implementing a multi-threaded reactor prototype:

1. **Verify Thread Scaling**:

```bash
# Start server with 4 reactor threads
./server --threads 4 --port 8080

# Benchmark with increasing concurrent connections
ab -c 100 -n 10000 http://localhost:8080/
ab -c 1000 -n 50000 http://localhost:8080/

# Monitor CPU utilization - should spread across cores
top -H -p $(pgrep server)
```

2. **Check WebSocket Upgrade**:

```bash
# Use wscat or similar WebSocket client
wscat -c ws://localhost:8080/chat

# Server should log: "HTTP upgrade to WebSocket successful"
# Multiple concurrent WebSocket connections should work
```

3. **Test TLS Handshake**:

```bash
# Start TLS server on port 8443
./server --tls --cert server.crt --key server.key --port 8443

# Test with curl
curl -k https://localhost:8443/

# Verify in logs: "TLS handshake completed"
```

G. Debugging Tips for Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
CPU not scaling with threads	All connections assigned to one thread	Check load balancer logic, `SO_REUSEPORT` kernel version	Implement work-stealing or explicit balancing
TLS handshake hangs	OpenSSL waiting for more data or blocking	Check SSL_want_read/write state after each call	Ensure non-blocking sockets and proper SSL mode flags
WebSocket random disconnects	Incorrect frame masking or fragmentation	Log raw frames and compare with RFC 6455	Debug with known-good WebSocket test suite
Memory growth with idle connections	Timer wheel not expiring in some threads	Check per-thread timer tick calls	Ensure all reactor threads call `timer_wheel_tick()`
Random crashes in OpenSSL	Thread safety not initialized	Check `CRYPTO_set_locking_callback()` call	Initialize OpenSSL thread support before creating threads
High latency under load	Worker pool queue saturation	Monitor queue depth and worker CPU	Increase worker threads or implement backpressure

These extensions transform the event loop from a learning exercise into a production-capable foundation. Each addition builds upon the core principles established in earlier milestones: non-blocking I/O, efficient timer management, and clean separation of concerns.

Glossary

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section provides definitive explanations of key terms and concepts used throughout this design document. Understanding this terminology is essential for working with the event loop architecture, debugging issues, and extending the system. Terms are listed alphabetically for quick reference.

Terminology Table

Term	Definition	First Reference/Context
ALPN (Application-Layer Protocol Negotiation)	A TLS extension that allows a client and server to negotiate which application-layer protocol (like HTTP/1.1, HTTP/2, or WebSocket) will be used within a TLS connection before data transmission begins.	Future Extensions (TLS integration)
backpressure	A flow control mechanism where a data consumer (like our event loop when writing to a slow client) signals to the producer (our application logic) to slow down or stop sending data because buffers are full and cannot be processed quickly enough.	Error Handling and Edge Cases (slow clients)
bump-in-the-stack	A design pattern for implementing TLS/SSL where encryption/decryption is inserted as a layer between the raw socket I/O and the application protocol handler. The event loop sees encrypted bytes, but the protocol handler works with plaintext.	Future Extensions (protocol handler architecture)
buffer management	The systematic approach to allocating, tracking, filling, and draining memory buffers used for reading from and writing to network sockets. This includes managing `read_buffer_used`, `write_buffer_used`, and `write_buffer_sent` fields in the `connection` struct.	Data Model (connection fields)
buffer watermark	A predetermined threshold (e.g., 80% full) within a buffer that triggers specific actions. For example, a write buffer reaching its high watermark might stop registering for `EPOLLOUT` to prevent busy loops, while dropping below a low watermark might re-enable it.	Error Handling and Edge Cases (flow control)
C10K problem	The classic challenge of designing a network server capable of handling ten thousand concurrent connections efficiently on commodity hardware. It exposed the limitations of the thread-per-connection model and spurred the adoption of event-driven architectures and I/O multiplexing.	Context and Problem Statement
callback API	A programming interface where the application registers functions (callbacks) to be invoked by the framework when specific events occur (e.g., `http_handle_read` for `EPOLLIN`). This is the core of the Reactor pattern's inversion of control.	Component Design (Async Task Scheduling)
connection state machine	A finite state machine (FSM) that models the lifecycle of a network connection using states like `STATE_ACCEPTED`, `STATE_READING`, `STATE_WRITING`, and `STATE_CLOSING`. Transitions between states are driven by I/O events, timer expirations, and protocol logic.	Data Model (state constants) & Interactions and Data Flow
CPU affinity	The practice of binding a specific thread or process to a particular CPU core (or set of cores). This can improve performance in multi-threaded reactors by reducing cache invalidation and context-switching overhead between cores.	Future Extensions (multi-threaded reactor)
edge-triggered	An event notification mode (set with `EPOLLET`) where `epoll` signals a file descriptor only when its I/O status *changes* (e.g., when data first becomes available). The application must assume it can read/write until it receives `EAGAIN`.	Component Design (Event Loop - pitfalls)
epoll	A Linux-specific I/O event notification facility that efficiently monitors multiple file descriptors to see if I/O is possible on any of them. It scales better than `select()` or `poll()` for large numbers of connections.	Context and Problem Statement (comparison table)
error categorization	The process of classifying system call errors (from `errno`) into categories like `ERROR_RETRYABLE` (`EAGAIN`), `ERROR_CONNECTION_FATAL` (`ECONNRESET`), and `ERROR_SYSTEM_FATAL` (`ENOMEM`) to determine the appropriate recovery action.	Error Handling and Edge Cases (error table)

Term	Definition	First Reference/Context
event loop	The central coordinating component of the Reactor pattern. It continuously waits for events (using `epoll_wait`), then dispatches them to registered handlers or callbacks. Also referred to as the "reactor" or "dispatcher."	High-Level Architecture (component diagram)
event loop tick	A single complete iteration of the event loop's main cycle: calling `epoll_wait`, processing all returned I/O events, ticking the timer wheel to process expired timers, and executing any deferred tasks.	Interactions and Data Flow (sequence diagram)
finite state machine (FSM)	An abstract model of computation consisting of a finite number of states, transitions between those states, and actions. Used extensively to model connection lifecycles and protocol parsing states (e.g., in `http_parser`).	Data Model (connection states)
half-closed connection	A TCP connection state where one endpoint has sent a FIN packet (indicating it has finished sending data) but is still willing to receive data from the other endpoint. This state must be handled correctly to avoid writing to a closed connection.	Error Handling and Edge Cases (edge cases)
hierarchical timer wheel	A multi-level circular buffer data structure for managing large numbers of timers with O(1) amortized cost for insertion, deletion, and expiration. Higher levels represent larger time intervals (e.g., minutes), lower levels represent smaller ones (e.g., milliseconds).	Component Design (Timer Wheel)
HTTP keep-alive	An HTTP/1.1 feature that allows multiple request/response transactions to occur over a single TCP connection, reducing connection establishment overhead. The server must manage an idle timeout for these persistent connections.	Component Design (HTTP Server)
idle timeout	A timer associated with a connection that fires if no read or write activity occurs within a configured period (e.g., `IDLE_TIMEOUT_MS`). This prevents resource exhaustion from clients that connect but never send data (a "slow loris" attack pattern).	Component Design (Timer Wheel - acceptance criteria)
incremental processing	A design approach where data is processed as it arrives in chunks, rather than waiting for a complete message. Essential for non-blocking I/O, as a single `read()` may only deliver part of an HTTP request.	Component Design (HTTP Server)
I/O multiplexing	The capability of an operating system or system call to monitor multiple I/O sources (file descriptors) simultaneously and report which ones are ready for reading or writing. `epoll`, `select`, and `poll` are all I/O multiplexing mechanisms.	Context and Problem Statement
level-triggered	The default event notification mode for `epoll`. The kernel notifies the application that a file descriptor is ready for I/O for as long as the condition persists (e.g., data remains in the socket buffer). This can lead to repeated notifications if not handled.	Component Design (Event Loop - pitfalls)
monotonic clock	A system clock that continuously increases at a constant rate and is not affected by discontinuous jumps in system time (e.g., from NTP adjustments). Used for measuring intervals and timer expirations via `clock_gettime(CLOCK_MONOTONIC)`.	Component Design (Timer Wheel - implementation)
MPSC (Multi Producer Single Consumer) ring buffer	A concurrent data structure where multiple threads (producers) can enqueue items, but only one thread (the consumer, typically the event loop) can dequeue them. Useful for passing tasks or connections between threads in a multi-threaded reactor.	Future Extensions (work-stealing)
multi-threaded reactor	An extension of the Reactor pattern where multiple threads, each running its own event loop, work together to handle connections. Connections are distributed among the threads, often using techniques like `SO_REUSEPORT` or a central acceptor.	Future Extensions
non-blocking I/O	A mode of socket operation where calls to `read()`, `write()`, and `accept()` return immediately, either with the completed result or with an error	Component Design (Event Loop)

Term	Definition	First Reference/Context
	(`EAGAIN` / `EWOULDBLOCK`) if the operation cannot be completed without waiting.	
protocol handler interface	A common interface (represented by the `protocol_handler` struct) that abstracts the specifics of different application-layer protocols (HTTP, WebSocket, TLS). It allows the event loop to treat connections uniformly, delegating protocol logic to handler functions.	Future Extensions (protocol upgrades)
Reactor pattern	An architectural pattern for event-driven applications. It demultiplexes incoming events (I/O, timers) and dispatches them synchronously to associated request handlers (callbacks). The pattern separates the mechanism of event detection from the application logic.	High-Level Architecture (postal system analogy)
slow loris attack	A denial-of-service attack that operates by opening many connections to a server and periodically sending small amounts of data to keep the connections open indefinitely, exhausting server resources. Defended against using aggressive idle timeouts.	Error Handling and Edge Cases (edge cases)
SO_REUSEPORT	A socket option (`setsockopt(SO_REUSEPORT)`) that allows multiple sockets (typically in different processes or threads) to bind to the same IP address and port number. The kernel performs load balancing of incoming connections.	Future Extensions (multi-threaded reactor)
SPSC (Single Producer Single Consumer) ring buffer	A concurrent data structure optimized for the case where exactly one thread produces data and exactly one different thread consumes it. Often used for communication between an I/O thread and a worker thread with minimal locking.	Future Extensions (task queues)
thread-local storage (TLS)	A mechanism for allocating variables that have a distinct instance for each thread. In a multi-threaded reactor, each thread's event loop state and connection tables can be stored in thread-local storage to avoid synchronization.	Future Extensions (implementation notes)
timer cascading	The process in a hierarchical timer wheel where, as the current time pointer advances, timers that have "overflowed" from a higher-precision wheel level (e.g., milliseconds) are moved down to a lower-precision level (e.g., seconds) for future expiration checking.	Component Design (Timer Wheel - algorithm)
timer wheel	A data structure for efficient timer management, conceptually a circular buffer of linked lists. Each slot corresponds to a time interval. A pointer advances at a fixed "tick," and all timers in the current slot expire.	Component Design (Timer Wheel)
work-stealing	A load balancing strategy used in concurrent systems where idle worker threads "steal" tasks from the queues of busy threads. This helps keep all threads utilized evenly and can improve throughput in a multi-threaded reactor.	Future Extensions
write timeout	A timer that fires if a write operation cannot make progress (i.e., the socket's send buffer remains full) within a specified period (e.g., `WRITE_TIMEOUT_MS`). This protects the server from extremely slow clients that cannot accept data.	Error Handling and Edge Cases (slow clients)

Implementation Guidance

While the glossary defines concepts, this section provides concrete guidance on how these terms manifest in the codebase and how to navigate the implementation.

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Production/Extension)
I/O Multiplexing	`epoll` with level-triggered mode (simpler logic)	`epoll` with edge-triggered mode (`EPOLLET`) for maximum performance
Timer Management	Single-level timer wheel (64-256 slots)	Hierarchical timer wheel (4 levels: 64ms, 4s, 4min, 4hr)
Concurrency Model	Single-threaded reactor (master all concepts)	Multi-threaded reactor with `SO_REUSEPORT` and work-stealing queues
Protocol Stack	HTTP/1.1 with keep-alive	Pluggable protocol handlers (HTTP/1.1, WebSocket, TLS via bump-in-the-stack)
Buffer Management	Simple linear buffers with realloc on overflow	Ring buffers or linked list of buffers for zero-copy operations

B. Recommended File/Module Structure The glossary terms map directly to modules and header files in the project structure:

```
project/
├── include/                # Public API headers
│   ├── event_loop.h        # `event_loop`, `event_loop_register_fd`, etc. (Reactor pattern)
│   ├── timer_wheel.h       # `timer_wheel`, `timer_wheel_add` (Timer management)
│   ├── connection.h        # `connection`, `connection_read` (I/O & state machine)
│   ├── protocols/          # Protocol handler interface
│   │   ├── http.h          # `http_parser`, `http_handle_read` (Incremental processing)
│   │   ├── websocket.h     # WebSocket protocol handler
│   │   └── tls.h           # `tls_protocol_data` (bump-in-the-stack TLS)
│   └── utils/
│       ├── error.h         # `error_category_t`, `categorize_error`
│       └── buffers.h       # Buffer management utilities
├── src/
│   ├── core/               # Core event loop implementation
│   │   ├── event_loop.c    # Event loop tick, callback dispatch
│   │   ├── timer_wheel.c   # Hierarchical wheel with cascading
│   │   └── connection.c    # Non-blocking I/O, backpressure logic
│   ├── protocols/
│   │   ├── http.c          # HTTP incremental parser, keep-alive
│   │   └── tls.c           # ALPN, TLS record layer handling
│   └── utils/
│       ├── error.c         # Error categorization logic
│       └── socket.c        # `set_nonblocking`, `safe_write`
└── examples/
    ├── echo_server.c       # Demonstrates basic event loop usage
    └── http_server.c       # Full HTTP server with idle timeouts
```

C. Infrastructure Starter Code The following complete header file establishes the foundational constants and type definitions used throughout the system, ensuring consistent terminology:

`include/definitions.h` :

```c
#ifndef DEFINITIONS_H

#define DEFINITIONS_H


#include <stdint.h>


// --- Connection State Machine States ---

#define STATE_ACCEPTED       0  // Connection accepted, waiting for first read

#define STATE_READING        1  // Actively reading request data

#define STATE_WRITING        2  // Writing response data (may be registered for EPOLLOUT)

#define STATE_CLOSING        3  // Connection is being shut down

#define STATE_PROCESSING     4  // Offloaded to worker thread (future extension)


// --- HTTP Protocol Specific States ---

#define HTTP_READING         10 // Reading HTTP request (headers/body)

#define HTTP_WRITING         11 // Writing HTTP response

#define HTTP_KEEPALIVE       12 // Request complete, waiting for next on same connection

#define HTTP_UPGRADING       13 // In process of upgrading to WebSocket


// --- Buffer Configuration ---

#define READ_BUFFER_SIZE     4096   // Initial size for per-connection read buffer

#define WRITE_BUFFER_SIZE    4096   // Initial size for write buffer

#define MAX_EVENTS           1024   // Maximum events to process per epoll_wait call


// --- Timeout Defaults (milliseconds) ---

#define IDLE_TIMEOUT_MS      30000  // Close connection after 30s of inactivity

#define WRITE_TIMEOUT_MS     30000  // Close connection if write stalls for 30s


// --- Error Categories (see error.c) ---

typedef enum {

    ERROR_RETRYABLE,          // EAGAIN, EWOULDBLOCK, EINTR - retry operation

    ERROR_CONNECTION_FATAL,   // ECONNRESET, EPIPE, ECONNABORTED - close connection

    ERROR_SYSTEM_FATAL,       // ENOMEM, EMFILE, ENFILE - log and abort/restart

    ERROR_PROGRAMMING,        // EBADF, EINVAL - bug in code

    ERROR_UNKNOWN             // Default for unclassified errno

} error_category_t;


// --- Common epoll event flags ---
```

```
// (These are defined in <sys/epoll.h> but repeated here for documentation)
// #define EPOLLIN     0x001  // Data available to read
// #define EPOLLOUT    0x004  // Ready for writing
// #define EPOLLERR    0x008  // Error condition
// #define EPOLLHUP    0x010  // Hang up (peer closed)
// #define EPOLLET     0x80000000 // Edge-triggered mode

#endif // DEFINITIONS_H
```

D. Core Logic Skeleton Code This skeleton for the main event loop function illustrates how the core glossary terms come together in the central dispatch mechanism:

`src/core/event_loop.c`:

```c
#include "event_loop.h"
#include "timer_wheel.h"
#include "connection.h"

int event_loop_run(struct event_loop* loop, int timeout_ms) {
    // TODO 1: Calculate the next timer expiration using timer_wheel_next_expiration()
    //         This ensures the event loop wakes up in time for the next timeout.
    // TODO 2: Determine the actual timeout for epoll_wait as the minimum of
    //         the provided timeout_ms and the time until the next timer expires.
    // TODO 3: Call epoll_wait() with the calculated timeout. Handle EINTR correctly.
    // TODO 4: For each returned epoll_event:
    //   a. Check for error conditions (EPOLLERR, EPOLLHUP) using check_socket_error()
    //   b. For EPOLLIN events: retrieve the connection* from event.data.ptr,
    //      call its registered read callback (e.g., http_handle_read)
    //   c. For EPOLLOUT events: retrieve connection*, call write callback,
    //      and deregister EPOLLOUT if the write buffer is now empty (backpressure)
    //   d. Handle any connection-fatal errors by calling connection_destroy()
    // TODO 5: After processing all I/O events, call timer_wheel_tick() to process
    //         any expired timers (e.g., idle timeouts, write timeouts).
    // TODO 6: Process the deferred task queue (loop->task_queue), executing each
    //         task's callback. This happens after I/O to avoid re-entrancy issues.
    // TODO 7: Return 0 to continue the loop, or -1 on fatal error.
}
```

E. Language-Specific Hints (C)

- **Monotonic Clock**: Use `clock_gettime(CLOCK_MONOTONIC, &ts)` for timer calculations, not `gettimeofday()`, to avoid issues with system time changes.
- **Non-blocking Sockets**: Always check the return value of `fcntl(fd, F_SETFL, O_NONBLOCK)` and handle errors. Consider using `accept4()` with `SOCK_NONBLOCK` for new connections.
- **Error Handling**: After an `epoll_wait` returns an event with `EPOLLERR`, call `getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &len)` to get the specific socket error.
- **Buffer Management**: When a `read()` returns `EAGAIN`, the data you've already read should be appended to the connection's `read_buffer`. The `read_buffer_used` field tracks how much is valid.

F. Milestone Checkpoint After implementing the full system (through Milestone 4), you can verify your understanding of the terminology by testing with the following scenario:

```bash
# Start the HTTP server with verbose logging
./build/http_server -p 8080 -v

# In another terminal, use netcat to simulate a slow loris attack
# (This tests idle timeout and connection state machine)
for i in {1..100}; do
  (echo -ne "GET / HTTP/1.1\r\nHost: localhost\r\n" | nc -q 0 localhost 8080 &)
done

# Monitor server logs - you should see connections being accepted (STATE_ACCEPTED),
# moving to HTTP_READING, then being closed by timer expiration after IDLE_TIMEOUT_MS
# when the incomplete request times out.
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
CPU at 100% with no traffic	Event loop spinning without blocking on `epoll_wait`	Check if `timer_wheel_next_expiration()` returns 0 incorrectly, or if timeout calculation is wrong.	Ensure the timeout passed to `epoll_wait` is `-1` (block forever) when there are no timers, not 0.
Connections stuck in `STATE_WRITING`	Write buffer full, but `EPOLLOUT` not registered or not firing	Log when `connection_write()` returns `EAGAIN` and verify `event_loop_register_write()` is called.	In edge-triggered mode, you must attempt to write until `EAGAIN` after receiving `EPOLLOUT`.
Timer not firing at correct time	Timer wheel cascading logic bug or incorrect time base	Log `current_time` in `timer_wheel_tick()` and compare with expected `expires_at` for a test timer.	Ensure you're using a monotonic clock and converting correctly to milliseconds.
Memory leak under load	`connection_destroy()` not called for all closed connections	Use `valgrind` or log `connection_create` / `connection_destroy` calls with connection FD.	Verify every code path that closes a socket (errors, timeouts, normal finish) calls `connection_destroy`.