Design Document

Marcos Costa 125882, José Mendes 114429

1. Introduction

This document describes the architecture and design of a high-performance concurrent HTTP server implemented in C. The system employs a multi-process/multi-thread architecture with sophisticated synchronization mechanisms, LRU caching, and comprehensive monitoring. The server is designed to handle thousands of concurrent connections while maintaining stability and performance under varying load conditions.

2. System Architecture

High-Level Architecture

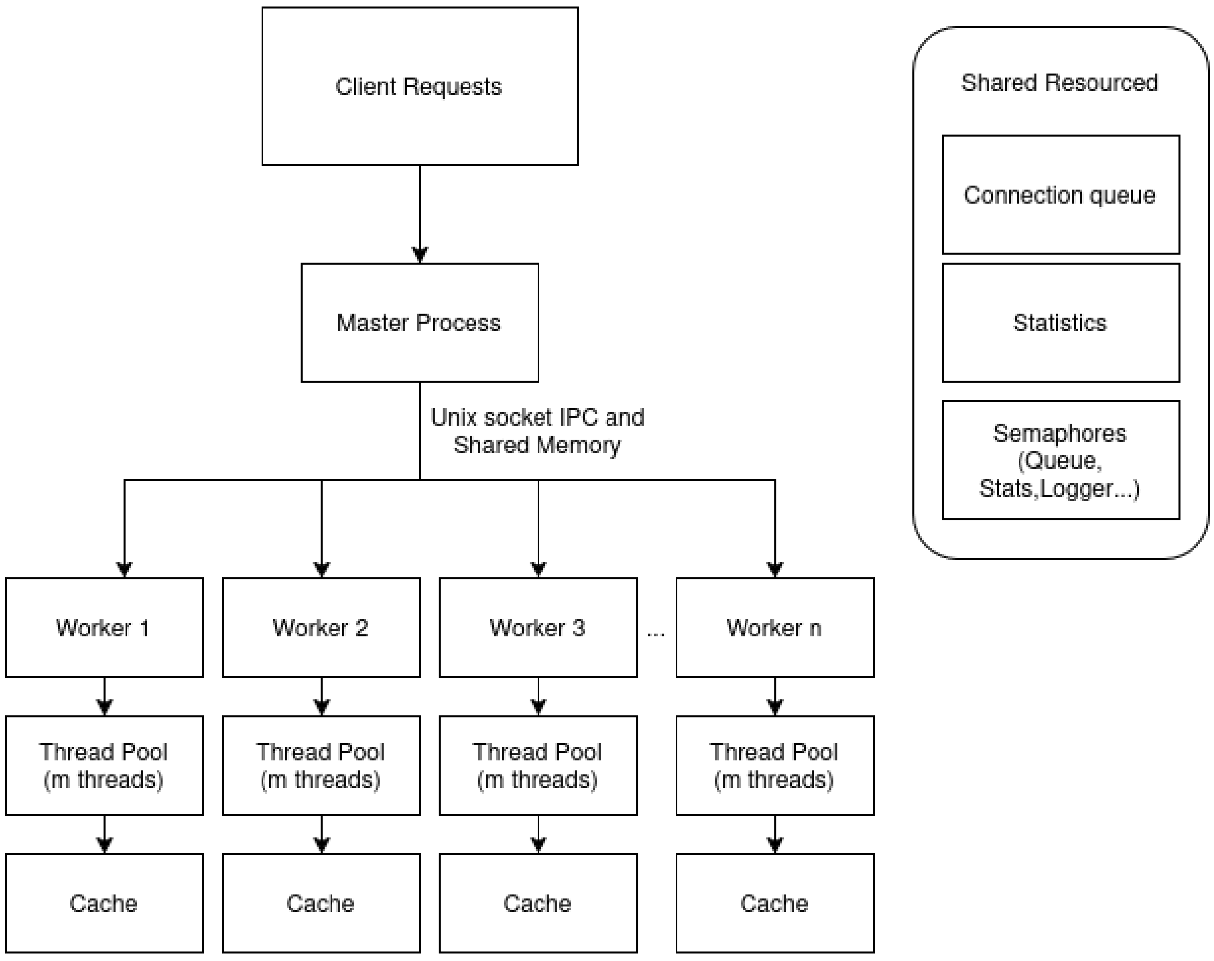


Figure 1: High-Level System Architecture

Process Hierarchy

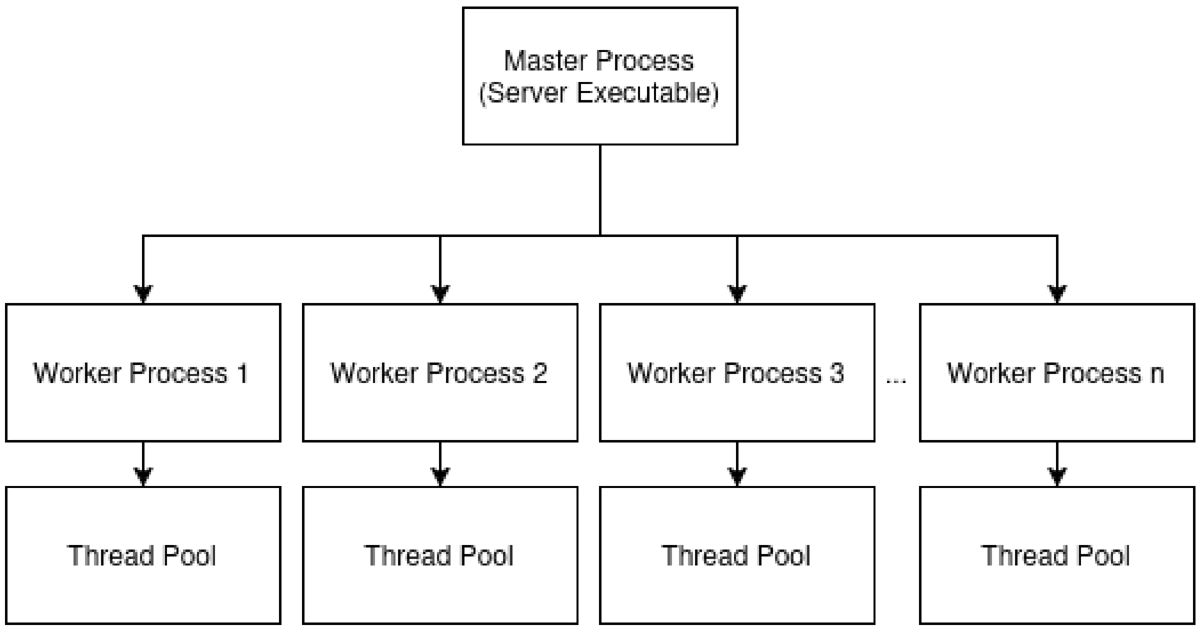


Figure 2: Process Hierarchy

The system follows an Master-Worker hierarchy where the master is responsible for accepting work and the worker for processing it, each worker also disposes of a thread pool for efficiency in the processing of said requests

3. Components Design

Master Process

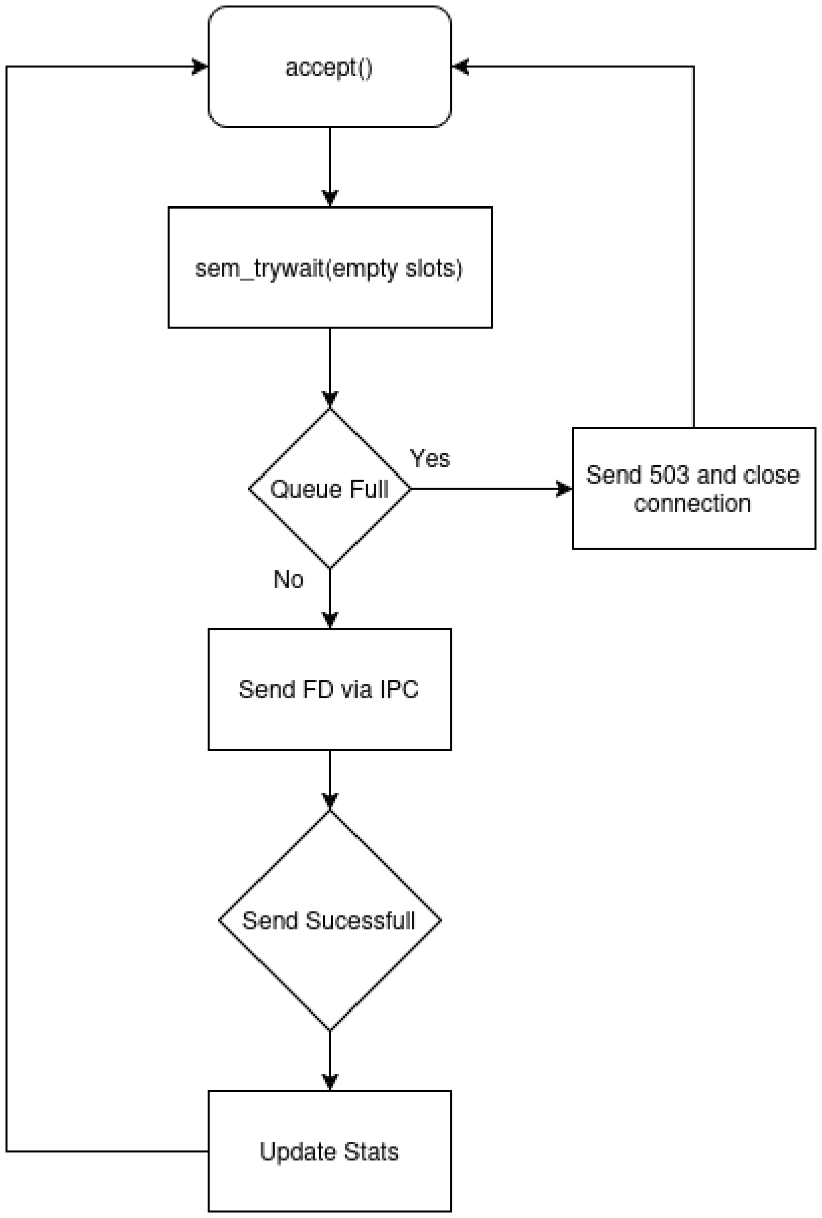


Figure 3: Master process Design

Semaphore System Design

Semaphore	***Initial Value***	***Purpose***	***Protects***
empty slots	queue size	Available queue space	Connection slots
filled slots	0	Items ready for processing	Worker readiness
queue mutex	1	Queue index protection	Queue front/rear pointers
stats mutex	1	Statistics protection	All statistical counters
log mutex	1	Log file protection	Log file writes

Table 1: Semaphores design

4. Synchronization Analysis

Critical Sections and Protection

Resource	Protection	Mechanism
Shared queue	queue mutex	Binary semaphore
Stats	stats mutex	Binary semaphore
Log file	log mutex	Binary semaphore

Local queue	pthread_mutex_t	Mutex
LRU cache	pthread_rwlock_t	Read-Write lock

Deadlock prevention

Circular Wait Prevention: Semaphores always acquired in consistent order:

- Master: empty_slots → queue_mutex → filled_slots
- Worker: filled_slots → queue_mutex → empty_slot

Timeout Mechanisms: Non-blocking `sem_trywait()` in master prevents indefinite blocking.

Race Condition Analysis

Queue Index Corruption

- Problem: Master and workers concurrently modify `front` and `rear` indices
- Solution: `queue_mutex` semaphore protects all queue index operations

Statistics Counter Updates

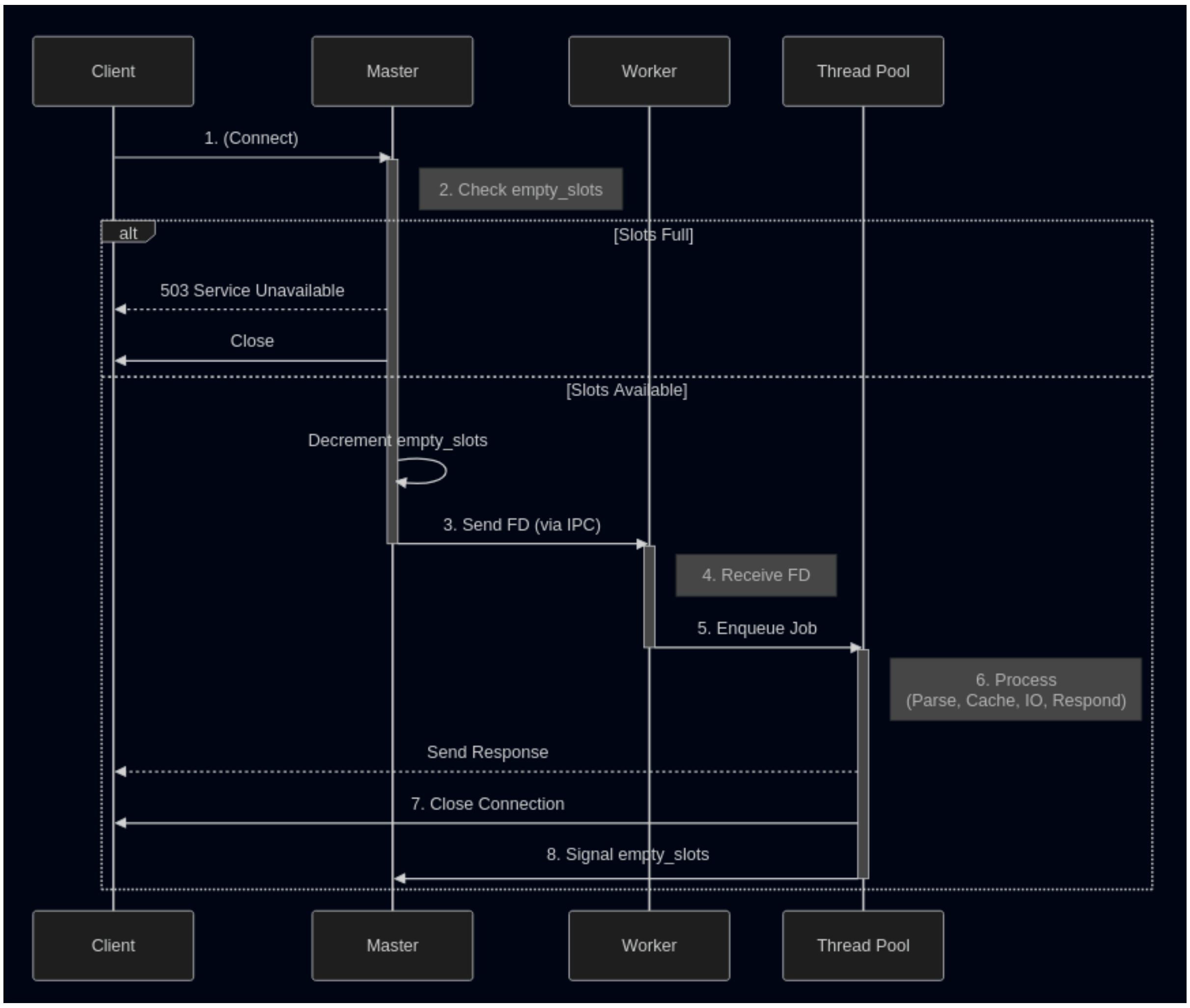
- Problem: Multiple threads updating same counters
- Solution: `stats_mutex` with atomic-like operations
- Impact: Minimal performance overhead due to short critical section

Cache Eviction During Read

- Problem: Thread reading while another evicts same entry
- Solution: Read-write lock allows concurrent reads, exclusive writes

5. Data Flow

Connection Lifecycle


Client
Master
Worker
Thread Pool
1. (Connect)
2. Check empty_slots
alt
[Slots Full]
503 Service Unavailable
Close
[Slots Available]
Decrement empty_slots
3. Send FD (via IPC)
4. Receive FD
5. Enqueue Job
6. Process (Parse, Cache, IO, Respond)
Send Response
7. Close Connection
8. Signal empty_slots
Client
Master
Worker
Thread Pool