Concurrent HTTP Server: Design Document

Inês Batista, 124877 & Maria Quinteiro, 124996 (P3G4)

1. Introduction

This document describes the architecture and synchronization methods for the ConcurrentHTTP Server, a multi-threaded, multi-process HTTP/1.1 server created for the Operating Systems course. The objective of the project is to develop a production-level web server that can effectively manage large numbers of simultaneous client connections, replicating real-world obstacles in concurrent systems programming. The main goal is to showcase proficiency in essential operating systems principles, such as process management, thread synchronization, and inter-process communication. The major technical hurdle involves handling simultaneous access to shared resources while avoiding race conditions, deadlocks, and data corruption. This design document outlines the server's master-worker framework, the synchronization methods used to manage processes and threads, and the data structures that facilitate effective request processing. This project examines the essential principles that support contemporary, high-performance web infrastructure.

2. System Architecture

Following the high-level objectives outlined in the introduction, this section explores the specific architectural design of the ConcurrentHTTP Server. The structure is a multi-threaded, multi-process framework that purposefully divides concerns to attain stability, scalability, and effective resource usage.

2.1. High-Level Overview

The system is built upon a master-worker pattern with direct message passing via UNIX Domain Sockets, creating a clear producer-consumer relationship between its components. The most effective way to understand the overall data flow and component interaction is through the following architectural diagram:

The following diagram illustrates the main data path and core components. The Master Process acts as the producer, accepting incoming socket connections and distributing them directly to Worker Processes using a Round-Robin scheduling algorithm via dedicated socket pairs. Multiple Worker Processes, forked by the master, act as consumers.

Each worker runs a Thread Pool where individual threads (T) process the HTTP requests delivered through the socket channel and generate responses. To optimize performance, each worker maintains a local LRU Cache for frequently accessed files. Crucially, shared resources include Global Statistics residing in shared memory, while all threads and processes write to a common Log File. The entire system is coordinated by synchronization primitives to ensure data consistency and prevent race conditions.

This architecture effectively decouples connection acceptance from request processing while eliminating queue contention, allowing the system to scale and handle a high volume of concurrent clients efficiently.

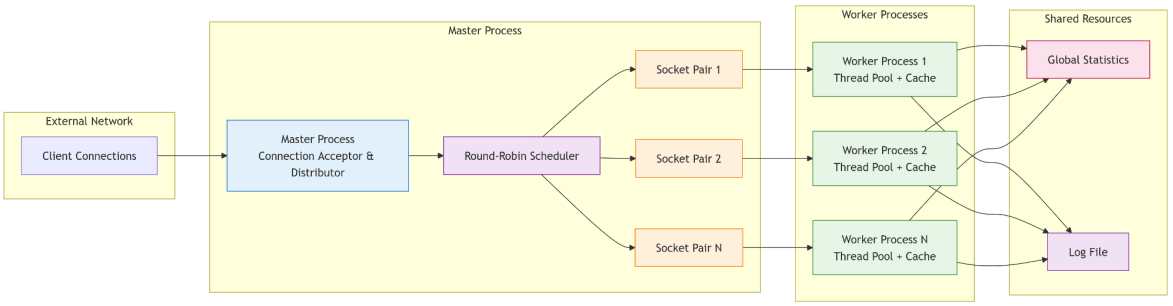

Figure 1: High-level architecture of the master-worker system with UNIX Domain Socket communication

3. Main Components

3.1. Master Process

The Master Process orchestrates system initialization and connection management. It handles socket creation, worker process forking, socket pair creation, and shared resource setup. Operating as the producer in the producer-consumer pattern, it accepts incoming connections and distributes them directly to workers via UNIX Domain Sockets using Round-Robin scheduling while managing graceful shutdown procedures.

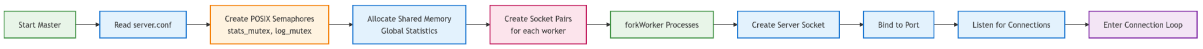

Figure 2: Step-by-step initialization sequence of the Master Process.

3.2. Worker Process and Synchronization Mechanisms

Worker Processes receive connections directly from the master via dedicated UNIX Domain Sockets and utilize thread pools for concurrent request processing. Each worker maintains an LRU cache to optimize file serving performance.

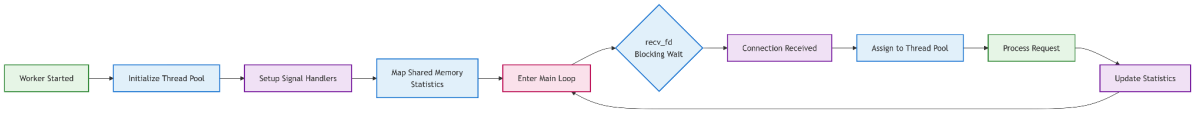

Figure 3: Lifecycle of a Worker Process from startup to continuous request processing.

Synchronization employs POSIX semaphores for inter-process coordination (global statistics) and pthread mechanisms for thread-level synchronization within workers. UNIX Domain Sockets enable efficient file descriptor passing between processes while maintaining data consistency under concurrent access.

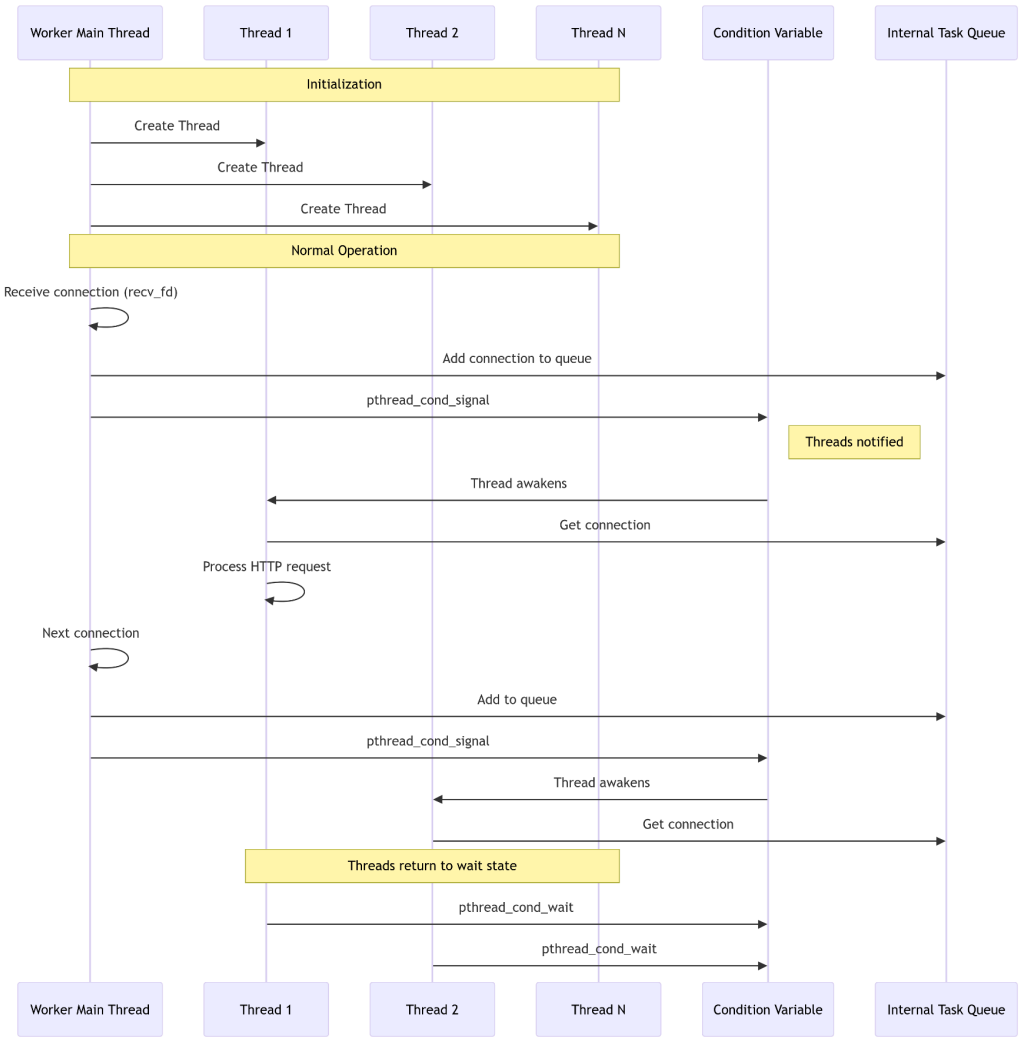

Figure 4: Thread pool operation within a worker showing task distribution and synchronization.

4. Synchronization Diagrams

4.1. Connection Queue (Producer-Consumer Pattern)

The connection distribution mechanism implements a direct message-passing model using UNIX Domain Sockets (socket pairs). This design choice simplifies synchronization, eliminates shared queue contention, and provides a clean, efficient channel for passing file descriptors between the master and worker processes.

Architectural Overview
The master process creates a socket pair (AF_UNIX, SOCK_STREAM) for each worker during initialization. Each worker inherits one end of its socket pair, while the master retains the other. Connections are distributed using a Round-Robin scheduling algorithm, where the master rotates through workers sequentially, ensuring balanced load distribution without requiring workers to compete for shared resources.

Synchronization Protocol

Unlike the classic producer-consumer pattern with semaphores, this approach uses blocking I/O operations on the socket pairs as the synchronization mechanism. The master sends file descriptors using send_fd(), and workers block on recv_fd() until a descriptor arrives. This naturally serializes access without additional locks

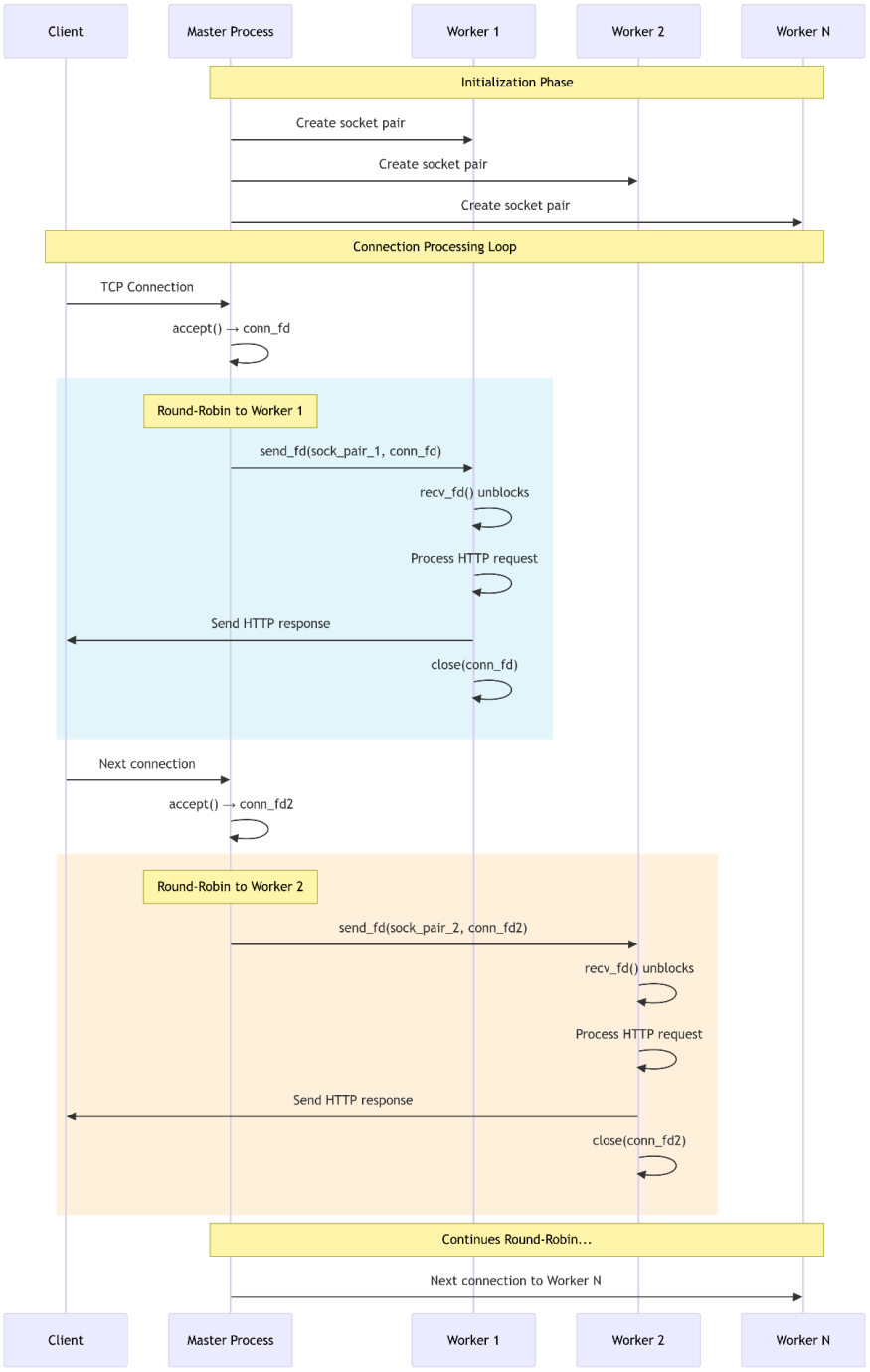

Figure 5: Sequence diagram illustrating the message-passing protocol for connection distribution.

The Master Process (Producer):
Operates as the connection distributor in a continuous loop. Upon accepting a new client connection via `accept()`, the master obtains the connection file descriptor and selects the next worker in the Round-Robin sequence. It then uses `send_fd()` to transmit the descriptor through the corresponding socket pair to the designated worker. After successful transmission, the master closes its local copy of the descriptor and immediately returns to accept the next connection. This streamlined process eliminates queue management overhead and ensures predictable, fair distribution of incoming requests across all worker processes.

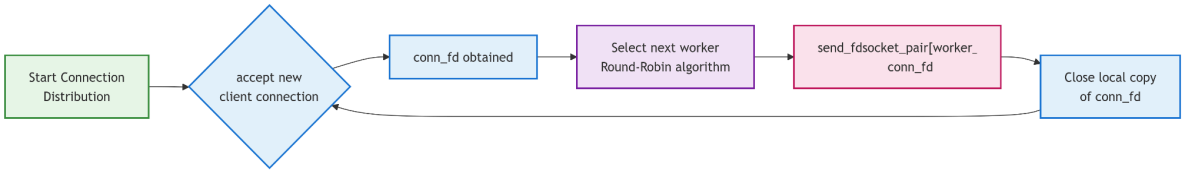

Figure 6: Master Process connection distribution flowchart using Round-Robin scheduling.

The Worker Process (Consumer):
Functions as a request processor that blocks on recv_fd() while waiting for incoming connection descriptors. When a descriptor arrives through its dedicated socket pair, the worker awakens and assigns the connection to an available thread from its internal thread pool. The worker thread then processes the HTTP request, serves the appropriate response, and closes the connection. Upon completion, the worker immediately returns to blocking on recv_fd() for the next descriptor. This blocking-wait approach provides natural backpressure: if all worker threads are busy, the master will block when attempting to send additional descriptors, preventing system overload without explicit synchronization mechanisms.

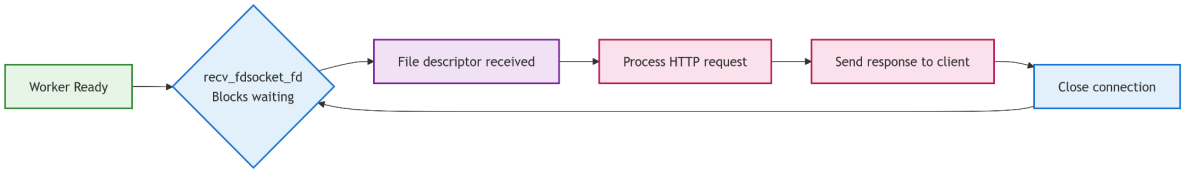

Figure 7: Worker Process request handling flowchart showing blocking receive and processing.

4.2. Cache Access (Read-Write Locks)

The LRU file cache employs pthread read-write locks to maximize parallelism for read operations while ensuring consistency during modifications.
For **Reader Threads**, multiple threads can simultaneously acquire read locks (pthread_rwlock_rdlock) and access cached data concurrently. This allows high throughput when serving popular static files. Each reader releases the lock with pthread_rwlock_unlock when finished.
For **Writer Threads**, an exclusive write lock (pthread_rwlock_wrlock) is required for cache modifications. This call blocks until all active readers release their locks, ensuring no threads are accessing the cache during structural changes. After completing the update (insertion or eviction), the writer releases the lock with pthread_rwlock_unlock, allowing waiting readers or writers to proceed.

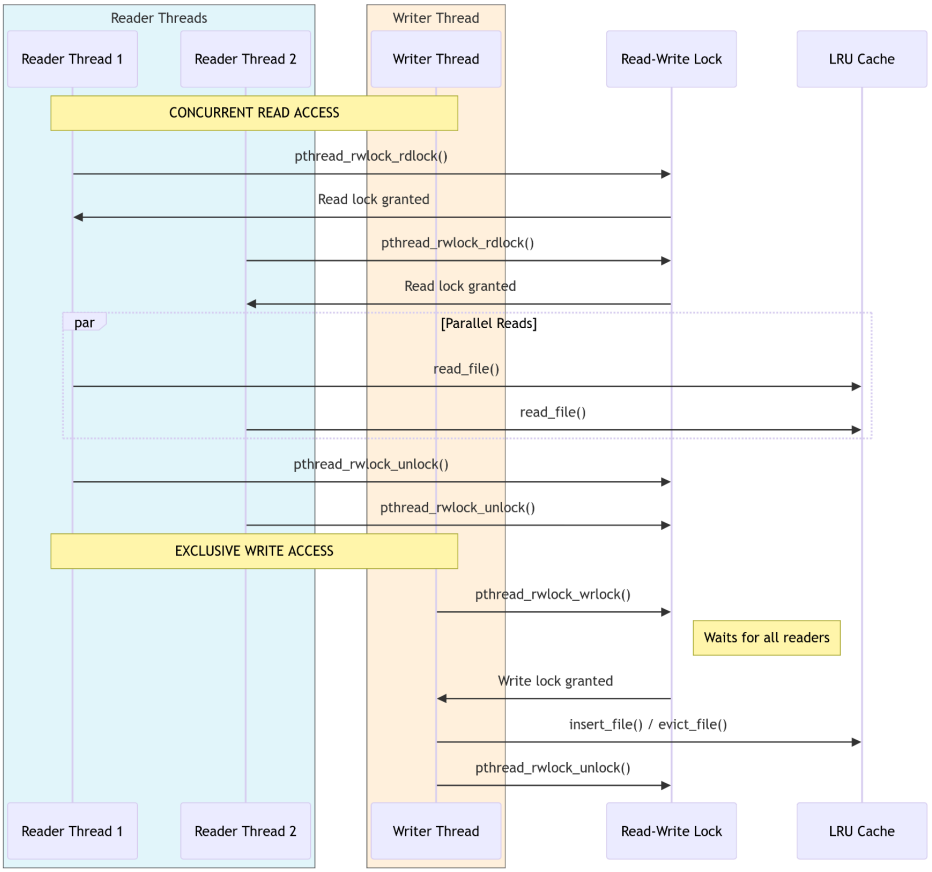

Figure 8: Reader-writer lock mechanism for concurrent cache access.

5. Data Structures and Algorithms

5.1. UNIX Domain Socket Structure

Inter-process communication between the Master Process and Worker Processes is implemented using UNIX Domain Sockets, created via the socketpair() function to establish bidirectional communication channels. For each worker, the Master creates a socket pair (AF_UNIX, SOCK_STREAM) that serves as a dedicated communication channel. After fork(), the Master retains one end of each socket pair, while each Worker inherits the corresponding other end. File descriptor transfer is accomplished using the send_fd() and recv_fd() functions, allowing the Master to send accepted client connections directly to Workers without requiring a shared queue. This approach eliminates contention for shared resources and simplifies synchronization, as socket I/O operations are kernel-managed, providing implicit synchronization and atomicity guarantees for descriptor passing. Worker selection follows a Round-Robin algorithm, ensuring balanced load distribution across all Worker processes.

5.2. LRU Cache Implementation

The file cache employs a combination of a doubly-linked list and hash map to achieve O(1) complexity for both access and eviction operations. While the detailed structure is omitted here for brevity, the algorithm maintains files in access order: recently accessed files are moved to the

head of the list, while the Least Recently Used files are evicted from the tail when the cache exceeds its 10MB limit.

5.3. Shared Memory Statistics

Global statistics are maintained in shared memory to provide real-time server metrics across all processes. Atomic updates are ensured through semaphore protection.
When a worker process completes a request, it follows a strict sequence: `sem_wait(stats.lock)` to acquire exclusive access, updates the relevant counters, and finally `sem_post(stats.lock)` to release the lock. This approach maintains data consistency while allowing the master process to periodically read accurate aggregate statistics.

6. Execution Flow

6.1. Server Startup and initialization

The server initialization follows a precise sequence to establish all necessary components for handling concurrent connections. The Master Process begins by reading the configuration file (server.conf) to obtain parameters such as port number, document root, number of workers, and thread pool sizes. Following configuration parsing, the master engages in critical system resource setup: it creates and initializes POSIX named semaphores required for inter-process synchronization, including stats_mutex for statistics protection and log_mutex for coordinated logging. Subsequently, it allocates and maps the shared memory segment that will host the global statistics structure. The master then creates UNIX Domain Socket pairs for each worker process, establishing dedicated communication channels. With the communication infrastructure in place, the master process executes a series of fork() system calls to create the specified number of Worker Processes. Each child worker process inherits the open socket descriptors and shared memory segments. During their initialization phase, the workers create their respective thread pools, where each thread is launched and immediately begins waiting on a condition variable for incoming connection descriptors. This comprehensive startup sequence ensures that when the master process finally creates the server socket, binds to the designated port, and transitions to a listening state, the entire ecosystem of workers and threads is prepared and waiting to process incoming client requests.

6.2. Request Processing Lifecycle

The lifecycle of an HTTP request is a coordinated journey through the server's concurrent architecture, as visualized in the following sequence diagram:

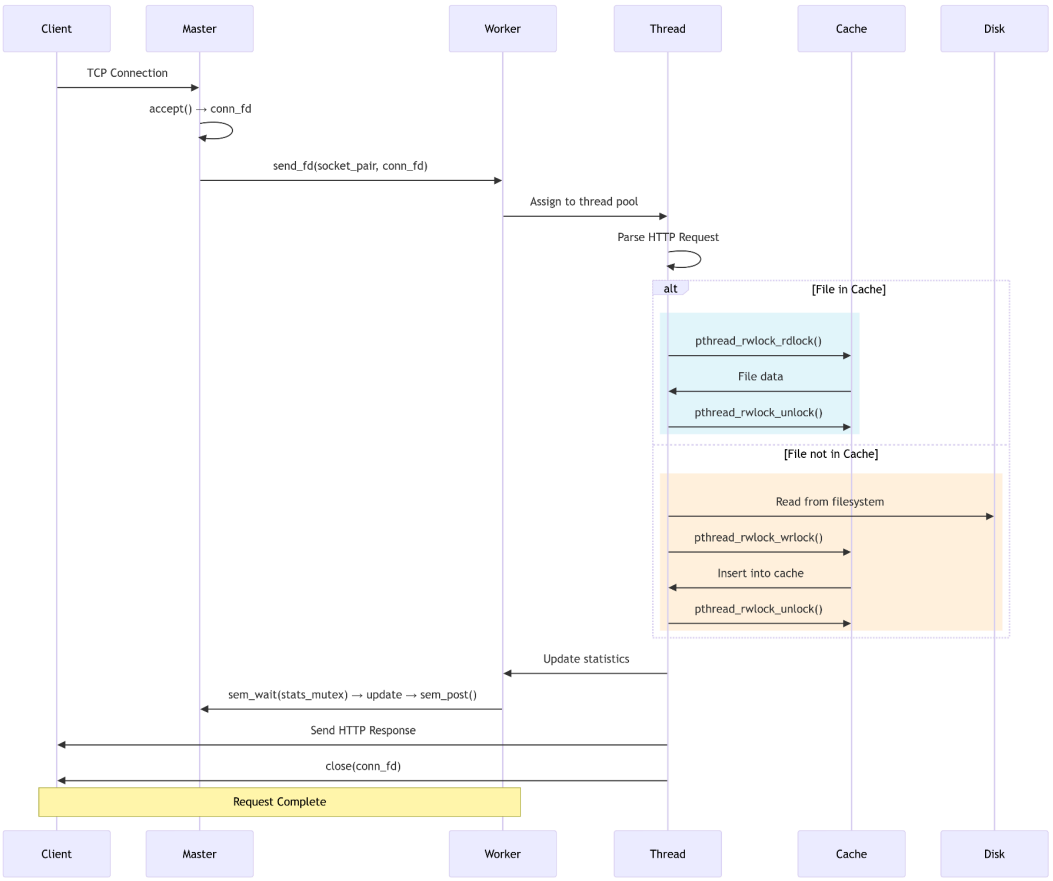

Figure 9: Complete HTTP request processing lifecycle from connection to response.

7. Concurrency and synchronization analysis

The server's design requires careful handling of simultaneous access to common resources to maintain data integrity and avoid system crashes. Regions that need protection are the Global Statistics, LRU File Cache, and Log File. Each utilizes customized synchronization methods: statistics use a dedicated mutex for atomic counter increments; the cache implements reader-writer locks to permit simultaneous reads while ensuring exclusive writes; and logging performs buffered writes with mutex safeguards to avoid interleaving while preserving efficiency. The connection distribution uses UNIX Domain Sockets which provide kernel-level synchronization, eliminating the need for application-level queue coordination.

To reduce concurrency risks, the design utilizes a tiered method for synchronization. Race conditions are avoided by thoroughly protecting all accesses to shared variables using suitable primitives—POSIX semaphores for inter-process coordination and pthread mechanisms for thread-level synchronization. Deadlock avoidance is attained by implementing a rigid, consistent order for obtaining locks across the codebase, ensuring that the cache write lock always has priority when multiple resources need concurrent access. Starvation avoidance is achieved via Round-Robin distribution that ensures fair access to connections for every worker process, upholding system equity during high load.