Design Document

Multi-Threaded Web Server with IPC

Martim Gil (ID: 124833)
Nuno Leite Faria (ID: 112994)
Operating Systems - University of Aveiro

December 10, 2025

Contents

1 Architecture Overview

The system adopts a multi-process, multi-threaded architecture based on a Master–Worker model. This structure enables efficient handling of multiple simultaneous client connections while preserving process isolation and scalability across CPU cores.

1.1 Process Model

The application consists of one **Master Process** and N **Worker Processes**. Each performs a distinct role in the server's execution pipeline:

- **Master:** Initializes configuration parameters, binds the listening socket, and accepts incoming TCP connections. Each accepted connection is first represented as a lightweight descriptor stored in a bounded shared-memory queue, protected by semaphores. The master then transfers the corresponding client file descriptor to the designated worker via a UNIX domain socket using `SCM_RIGHTS`. This hybrid approach combines efficient interprocess signaling with safe descriptor passing.
- **Workers:** Spawned by the master at startup, each worker waits for connection assignments in the shared queue. Upon notification, it receives the actual socket descriptor through the UNIX channel and submits it to its local thread pool for processing. Each worker maintains its own cache and counters but contributes to global statistics through synchronized shared memory updates.

Figure 1: High-Level Architecture Diagram

1.2 Thread Model

Within each worker process, a fixed sized pool of threads is created during initialization. The pool is managed through a bounded queue protected by a `pthread_mutex_t` and a pair of condition variables for signaling. Threads block when the queue is empty and wake upon task submission. Each thread handles one complete HTTP request–response cycle before returning to the idle state. This design eliminates the overhead of thread creation at runtime and maintains steady throughput under heavy load.

1.3 Coordination and Synchronization

Interprocess coordination relies on POSIX shared memory and named semaphores. Shared memory stores both the connection queue and aggregated statistics, while semaphores enforce mutual exclusion and synchronize access between the master and workers. Within workers, thread-level synchronization is handled by mutexes and condition variables, ensuring safe access to shared structures such as the cache and log system. This combination of IPC primitives and in-process synchronization guarantees correctness even under high concurrency.

1.4 Design Rationale

The architecture separates responsibilities cleanly: the master focuses on connection management and system orchestration, while workers specialize in request handling. The use of shared memory for lightweight signaling and descriptor passing through UNIX domain sockets minimizes communication overhead and preserves process isolation. This design yields a scalable, fault-tolerant server capable of leveraging multi-core systems effectively while maintaining predictable and stable performance.

2 Shared Data Structures

Inter-Process Communication (IPC) between the master and workers relies on POSIX shared memory with named semaphores. The shared region holds a bounded queue for connection dispatch and a set of global runtime counters.

2.1 Memory Layout

Defined in `shared_mem.h`, the `shared_data_t` structure is allocated by the master process and mapped by all workers.

```
1  typedef struct {
2      request_queue_t queue;   // Connection queue
3      server_stats_t  stats;   // Global statistics
4  } shared_data_t;
```

Creation uses `shm_open()`, `ftruncate()`, and `mmap()`, with cleanup performed by the master on shutdown via `shm_unlink()`.

2.2 Circular Buffer

The interprocess queue is implemented as a bounded FIFO in shared memory:

```
1  typedef struct {
2      int worker_id;       // Target worker
3      int placeholder_fd;  // Marker only (real FD passed via
           SCM_RIGHTS)
4  } connection_item_t;
5
6  typedef struct {
7      connection_item_t items[MAX_QUEUE_SIZE];
8      int front;           // Dequeue index
9      int rear;            // Enqueue index
10     int count;           // Current occupancy
11 } connection_queue_t;
```

Semantics. The Master (producer) enqueues a `connection_item_t` selecting the `worker_id`; the `placeholder_fd` is not a transferable descriptor. The actual client socket is sent to the chosen Worker over a UNIX domain socket using `SCM_RIGHTS`.

Synchronization. Enqueue/dequeue follow classical producer–consumer control with named semaphores: `empty_slots` (free capacity), `filled_slots` (pending items), and `queue_mutex` (exclusive updates of `front`/`rear`/`count`). On enqueue: wait `empty_slots` → lock `queue_mutex` → write `items[rear]` and advance `rear` (mod `MAX_QUEUE_SIZE`); increment `count` → unlock → post `filled_slots`. On dequeue: wait `filled_slots` → lock `queue_mutex` → read `items[front]` and advance `front`; decrement `count` → unlock → post `empty_slots`. This guarantees correctness under concurrent producers/consumers and prevents overrun/underrun of the circular buffer.

2.3 Global Statistics

The `server_stats_t` structure, located in the same region, records cumulative server activity.

```
1    typedef struct {
2        long total_requests;
3        long bytes_transferred;
4        long status_200;
5        long status_404;
6        long status_500;
7        int  active_connections;
8        long total_response_time_ms;
9    } server_stats_t;
```

Workers increment these counters under a dedicated semaphore (`stats_mutex`) to guarantee atomic access. Values are read periodically by monitoring components such as the web dashboard and log system.

2.4 Synchronization Lifecycle

The master initializes shared memory and all semaphores during startup, while workers connect to existing objects. At termination, the master releases all IPC resources, preventing persistent semaphores or memory segments.

This design achieves reliable communication and coordinated state management with minimal interprocess overhead.

3 Synchronization Mechanisms

3.1 Process Synchronization (Named Semaphores)

Master and Workers coordinate over a bounded circular queue in shared memory using named POSIX semaphores (see `semaphores.h/.c`, `shared_mem.h/.c`):

- `empty_slots` (`/ws_empty`) — counts free positions in the interprocess queue; the Master (producer) waits when the queue is full.
- `filled_slots` (`/ws_filled`) — counts pending items. Workers (consumers) wait when the queue is empty.
- `queue_mutex` (`/ws_queue_mutex`) — mutual exclusion for updating queue indices (`front/rear/count`).

- `stats_mutex` (`/ws_stats_mutex`) — serializes updates to `server_stats_t` (see `stats.c`).
- `log_mutex` (`/ws_log_mutex`) — serializes log writes across processes.

Producer path (in `master.c`): wait `empty_slots` → lock `queue_mutex` → enqueue → unlock → post `filled_slots`. Consumer path (in `worker.c`): wait `filled_slots` → lock `queue_mutex` → dequeue → unlock → post `empty_slots`.

3.2 Thread Synchronization (Mutex & Condition Variables)

Inside each Worker, the thread pool uses in-process primitives (see `thread_pool.h/.c`):

- `pthread_mutex_t` — protects the internal job queue and shared pool state.
- `pthread_cond_t` — allows worker threads to sleep when the queue is empty and wake on job submission or shutdown.

Threads block on `pthread_cond_wait` when there is no work and are released by `pthread_cond_signal`/`broadcast` on submission or shutdown, ensuring stable throughput without busy-waiting.

4 Component Design

4.1 LRU Cache

Each Worker maintains an in-memory Least Recently Used (LRU) cache for static file contents, combining a hash table (for $O(1)$ lookups) with a doubly linked list (for recency order). Capacity is enforced in bytes; entries in active use are protected from eviction via reference counting.

Structures.

```
1     // Cache entry: path key, raw data, size, list links, bucket
          link, refcount
2     typedef struct cache_entry {
3         char *key;
4         uint8_t *data;
5         size_t size;
6         struct cache_entry *prev, *next;
7         struct cache_entry *hnext;
8         size_t refcnt;
9     } cache_entry_t;
10
11    // Cache container: capacity/accounting, LRU ends, hash table,
          RW lock, stats
12    struct file_cache {
13        size_t capacity;
14        size_t bytes_used;
15        size_t items;
```

```
        cache_entry_t *lru_head;
        cache_entry_t *lru_tail;
        size_t nbuckets;
        cache_entry_t **buckets;
        pthread_rwlock_t rwlock;
        size_t hits, misses, evictions;
    };
```

Concurrency. The cache is protected by `pthread_rwlock_t`: lookups acquire the read lock; insertions, updates, and evictions acquire the write lock. This enables high read concurrency while preserving exclusive access for structural changes.

Operation. `cache_get(key)` computes the bucket, scans for a matching entry, and on hit promotes it to the LRU head. On miss, the Worker loads the file, then `cache_put(key, data, size)` inserts the entry, updating `bytes_used`. If capacity is exceeded, the cache evicts from the LRU tail until within limit; entries with `refcnt>0` are skipped. Oversized files may be served but not cached.

Integration. Workers consult the cache before disk I/O. On hit/miss, they update the cache counters and the shared system statistics accordingly. Because each Worker owns its cache, no inter-process locking is required; only per-cache RW locking applies.

4.2 Thread-Safe Logging

All processes append to a single log. Writes are serialized with a named POSIX semaphore and the file is opened with `O_APPEND` to ensure atomic append.

API. The logger records structured entries via:

```
    void logger_write(const char* ip,
                      const char* method,
                      const char* path,
                      int status,
                      size_t bytes_sent,
                      long duration_ms);
```

Each call emits a single line containing client IP, method, path, status, response size, and request latency.

Synchronization. A named semaphore (`sem_open`, initial value 1) guards the critical section:

- `sem_wait(g_sem)` before formatting/buffering;
- optional buffered accumulation; periodic/time-based flush;
- rotation check; atomic `write`/`fflush`;
- `sem_post(g_sem)` on exit.

Using `O_APPEND` guarantees kernel-level append semantics even if multiple processes hold the file descriptor.

Rotation & Buffering. The logger maintains an internal buffer and rotates the file at about 10 MB, keeping a finite number of generations. A time-based flush (e.g., 5 s) reduces syscall frequency without relaxing mutual exclusion.

Integration. The Master logs accepts and lifecycle events; Workers log per-request lines with latency. This yields stable, non-interleaved audit trails suitable for throughput/latency analysis and error triage.

4.3 Summary

Component	Synchronization	Purpose
Cache	`pthread_rwlock_t`	Multi-reader/single-writer protection for static file caching.
Logger	`sem_t *log_mutex`	Serializes access to shared log file across all processes.
Threaded Logger	`pthread_mutex_t`, `pthread_cond_t`	Asynchronous batched logging for reduced I/O contention.

Table 1: Component Synchronization Summary

5 System Lifecycle

5.1 Initialization

At startup, the Master process loads configuration parameters from `server.conf`, then creates the required POSIX shared memory region and named semaphores that enable communication and synchronization across processes. It proceeds to initialize the TCP listening socket and fork the configured number of Worker processes. Each Worker attaches to the existing shared memory and semaphores, initializes its local components (cache, logger, and thread pool), and waits for tasks to arrive through the interprocess queue.

5.2 Steady Operation

During normal execution, the Master continuously accepts new client connections and enqueues them into the shared circular buffer. Each Worker dequeues available requests, dispatching them to a thread pool that handles HTTP parsing, response generation, cache lookup, and logging. Shared statistics are updated atomically under semaphore protection. This design ensures high concurrency while maintaining strict consistency of shared state.

5.3 Graceful Shutdown

When receiving a termination signal (`SIGINT` or `SIGTERM`), the Master stops accepting new connections and signals all Workers to exit. Each Worker halts its dispatch loop, terminates its thread pool after completing active requests, and releases local resources. Finally, the Master process waits for all Workers to terminate and then cleans up global

IPC objects, including shared memory segments and named semaphores, before exiting cleanly.

5.4 Error Recovery

If initialization or runtime failures occur—such as socket binding errors, semaphore creation issues, or shared memory faults—the system logs the event, cleans any partially allocated resources, and exits with an appropriate status code to prevent IPC leakage.

6 Flowcharts

Figure 2: Request Processing Flowchart