

Technical Report

Multi-Threaded Web Server with IPC and Semaphores

Martim Gil (ID: [124833])

Nuno Leite Faria (ID: 112994)

Operating Systems - University of Aveiro

December 8, 2025

Abstract

This document outlines the creation and execution of a concurrent HTTP server built in C for the Operating Systems course at the University of Aveiro.

The primary purpose of the project was to create a system that could effectively manage numerous client requests at the same time using a combined process-thread architecture.

The system incorporates sophisticated synchronization and interprocess communication methods, specifically POSIX semaphores and shared memory, guaranteeing consistency and mutual exclusion during access to shared resources. Extra modules were added for safe concurrent logging, file caching, and live statistics tracking.

The suggested solution underwent an extensive array of functional, concurrency, and stress evaluations. The results obtained show the reliability, scalability, and accuracy of the synchronization mechanisms, along with the consistent performance of the system under high workloads

Contents

1 Introduction

Creating concurrent servers is among the most effective and demonstrative uses of synchronization, inter-process communication, and thread management concepts explored in operating systems.

This project focused on creating and deploying a straightforward yet effective HTTP server that can handle multiple concurrent client requests simultaneously.

The C programming language was used for the system's implementation, utilizing POSIX system calls for thread creation and synchronization, semaphore management, shared memory management, and inter-process communication. The structure adopts a modular design to guarantee the maintainability and extensibility of various system elements

2 Implementation Details

The concurrent web server was implemented entirely in the C programming language using POSIX system calls for process management, thread creation, synchronization, and interprocess communication (IPC).

The implementation follows a modular architecture in which each subsystem corresponds to a dedicated source file or pair of header and implementation files under the **src/** directory.

This structure promotes reusability, maintainability, and clear separation of concerns between the networking layer, synchronization mechanisms, and data management components.

The Diagram below shows the structure:

- **concurrent-http-server/** (root directory)
 - .gitignore
 - Makefile
 - README.md
 - server.conf
 - server.log, access.log
 - **docs/** - Documentation and reports
 - * design/ - LaTeX design document
 - * report/ - Technical report (LaTeX + PDF)
 - * user_manual/ - End-user manual
 - **enunciados/** - Documentation given as project guidelines
 - **src/** - All C source code (core of the system)
 - * cache.c, cache.h
 - * config.c, config.h
 - * http_builder.c, http_builder.h
 - * http_parser.c, http_parser.h
 - * logger.c, logger.h

- * master.c
- * semaphores.c, semaphores.h
- * shared_mem.c, shared_mem.h
- * stats.c, stats.h
- * stats_reader.c
- * thread_logger.c
- * thread_pool.c, thread_pool.h
- * worker.c, worker.h
- **tests/** - Automated and manual testing scripts
 - * README.md
 - * stress_test.sh
 - * test_cache
 - * test_concurrent.c
 - * test_load.sh
 - * test_suite.sh
- **www/** - Static web content served by the HTTP server
 - * index.html, style.css, script.js, script.ts
 - * image.png, doc.pdf
 - * errors/ - Error pages (404.html, 500.html)
- **bin/** - Generated during compilation (created by Makefile)
 - * webserver (compiled binary)

2.1 System Architecture

At a high level, the project consists of three main execution entities: the **master process**, the **worker threads**, and the **auxiliary tools**. The master process is responsible for system initialization, socket setup, configuration parsing, and coordination of shared resources. Each worker thread, managed by a thread pool, handles an individual client connection. Auxiliary programs, such as **stats_reader**, access the server's shared memory region to retrieve runtime statistics without interfering with normal operation.

The core modules are summarized below:

- **master.c** – Initializes the system, loads configuration parameters, creates synchronization primitives, and launches the listening socket. It also spawns the thread pool and dispatches incoming connections.
- **worker.c / worker.h** – Implements the logic executed by each worker thread. It reads HTTP requests, interacts with the cache and statistics modules, and builds HTTP responses.
- **thread_pool.c / .h** – Manages a fixed number of worker threads and a queue of pending tasks. This module is responsible for dynamic workload distribution and concurrency control.
- **cache.c / .h** – Provides an in-memory file cache to reduce disk I/O. Cached entries are protected by semaphores to ensure mutual exclusion.

- **logger.c / .h** and **thread_logger.c** – Implement thread-safe logging using semaphores to serialize write operations to the log file.
- **stats.c / .h** and **shared_mem.c / .h** – Manage global statistics (requests, bytes, cache hits) stored in a POSIX shared memory region accessible from the external reader program.
- **semaphores.c / .h** – Encapsulates POSIX semaphore operations (**sem_open**, **sem_wait**, **sem_post**, etc.) to provide a clean synchronization interface.
- **http_parser.c / .h** and **http_builder.c / .h** – Handle HTTP protocol parsing and response construction.
- **config.c / .h** – Loads server parameters such as port, document root, cache size, and number of threads from **server.conf**.

2.2 Process and Thread Management

The concurrent HTTP server uses a combined process–thread architecture that separates the acceptance of client connections from the handling of individual requests. The **master** process is responsible for system initialization, resource allocation, and the creation of a fixed-size thread pool. This approach avoids the overhead associated with creating and destroying threads on demand while allowing high levels of concurrency and efficient resource utilization.

Once the server started, the **master** process enters a blocking loop where it can accept incoming TCP connections through the listening socket. For each connection that is accepted, the corresponding file descriptor is packaged as a task and submitted to the thread pool by invoking the function **thread_pool_add_task()**.

Thread Pool Initialization. The thread pool, implemented in **src/thread_pool.c**, is initialized with a fixed number of worker threads determined by the configuration file (**server.conf**). Each thread is created at startup using the POSIX **pthread_create()** function and executes a continuous loop defined in the **thread_worker()**. At the start, all threads enter a waiting state, blocked on a condition variable until new work becomes available in the shared task queue.

Task Scheduling. Tasks are kept in a synchronized queue secured by a mutex. When the master process adds a new connection to the queue, it maintains the queue and places the socket descriptor, and notifies one of the waiting threads by calling **pthread_cond_signal()**. When the signal is received, a worker thread activates, finds the task, and frees the mutex, and starts handling the connection. If there are no tasks present, threads stay idle, blocked on the condition variable.

Worker Thread Execution. Every worker thread, found in **src/worker.c**, is accountable for managing an individual HTTP transaction. The workflow for execution is the following:

1. Obtain the HTTP request from the designated socket descriptor.

2. Analyze the request with the `http_parser` to obtain the method, path, and headings.
3. Check if the requested resource is present in the cache. If it isn't available, get it back from the disk and place it into the cache.
4. Construct the HTTP response utilizing the `http_builder` and return it to the client.
5. Revise the common statistics stored in memory (e.g., total requests, cache hits/misses) and add an entry to the log.

This structure enables several worker threads to function simultaneously on different connections while the access to common resources like the cache, statistics counters, and log files—stays coordinated through semaphores and mutexes

Synchronization and Graceful Shutdown. The use of condition variables guarantees efficient coordination between the master process and worker threads. During server shutdown, the master process signals all condition variables, allowing blocked threads to exit their waiting loops. Each thread is then joined using `pthread_join()`, assuring a clean and orderly release of all allocated resources.

Overall, the process and thread management strategy balances concurrency with control, providing scalable performance while maintaining thread safety and predictability in the server's behavior.

2.3 Synchronization and IPC

To have a reliable and consistent operation in a multithreaded setting, the server uses synchronization methods utilizing POSIX semaphores, mutexes, and shared memory. This safeguard common resources like the task queue, the cache, the data, and the log file.

Semaphore Abstraction. The semaphore subsystem encapsulates the POSIX API functions `sem_open()`, `sem_wait()`, `sem_post()`, and `sem_unlink()`, offering operations such as `semCreate()`, `semConnect()`, `semDown()`, and `semUp()`. Each semaphore is initialized with an initial value depending on its purpose (e.g., binary for mutual exclusion, counting for producer-consumer synchronization).

Producer-Consumer Coordination. The interaction between the master process and the worker threads follows the producer-consumer model. The master acts as the producer by inserting new tasks (socket descriptors) into a shared circular buffer implemented in the thread pool, while the workers act as consumers by retrieving and processing these tasks. Access to the buffer is controlled by two semaphores:

- A *counting semaphore* representing the number of available tasks.
- A *binary semaphore* (mutex) protecting concurrent access to the queue structure itself.

Shared Statistics and Atomicity. Global runtime metrics—including total requests served, bytes transmitted, and cache hit/miss counters—are stored in a POSIX shared memory region managed by `src/shared_mem.c`. To prevent inconsistent updates when multiple threads increment or modify these counters simultaneously, all accesses are wrapped in critical sections protected by binary semaphores. This ensures atomicity of operations such as:

```
stats.requests++;
stats.bytes_sent += response_size;
```

The external utility `stats_reader` connects to the same shared memory object and can safely read these values in real time without interfering with the server’s internal operations.

Mutual Exclusion in Shared Logging. Logging operations, implemented in `src/logger.c` and `src/thread_logger.c`, are performed concurrently by multiple worker threads. To prevent simultaneous writes that could corrupt the log file, a named binary semaphore ensures exclusive access to the output stream. When a thread intends to log an event, it performs a `semDown()` operation before writing and a `semUp()` afterward.

Synchronization Robustness. By employing named semaphores and shared memory, the system allows independent processes to coordinate while maintaining persistence across program instances. All synchronization objects are unlinked and destroyed at shutdown, ensuring that no orphaned semaphores or shared memory regions remain active. This make sure that we have a reliable and consistent cleanup and prevents resource leakage in the operating system.

Overall, the synchronization and IPC design provides a robust foundation for safe concurrent operation.

2.4 Resource Management (Cache and Files)

Efficient resource management is a fundamental aspect of the server’s performance. To minimize disk access latency and improve response time, the system incorporates a lightweight in-memory caching mechanism implemented in `src/cache.c`. The cache temporarily stores the contents of frequently requested files, allowing subsequent requests for the same resource to be served directly from memory without accessing the filesystem.

Cache Architecture. The cache is implemented as a fixed-size table of entries, each representing a single cached file. Each entry stores the following attributes:

- The file path (`char *path`);
- A pointer to the file content loaded into memory;
- The file size (in bytes);
- Metadata for cache management, such as last access time or usage counter.

The total number of entries and the maximum memory size are defined in the configuration file (`server.conf`) and initialized at server startup by the `cache_init()` routine.

LRU Replacement Policy. When the cache becomes fully populated, it needs to remove an current entry to allow for new information. The executed plan follows the **Least Recently Used (LRU)** strategy, which removes the entry that has not been accessed, for an extended period. Every cache entry keeps a timestamp that gets refreshed with each entry. Upon adding, the system loops through the cache table to find the least just utilized the current entry and substitutes it with the updated file information. This policy offers a fair balance between performance and memory efficiency, as it inherently keeps commonly requested documents while clearing room for new ones.

Reader–Writer Locking. To support concurrent read and write operations on the cache, the implementation employs **POSIX reader–writer locks** (`pthread_rwlock_t`). These locks allow multiple worker threads to read cache entries simultaneously, while ensuring that write operations (such as file insertion or eviction) occur exclusively. The concurrency model operates as follows:

- **Readers (cache hits):** When a requested file is present in the cache, the worker thread obtains a *read lock* (`pthread_rwlock_rdlock()`) to securely access the entry without obstructing other readers.
- **Writers (cache misses or evictions):** When a file needs to be added or modified, the thread obtains a *write lock* (`pthread_rwlock_wrlock()`) to ensure exclusivity access, blocking any concurrent reads or writes.

This approach significantly improves performance compared to traditional mutex-based synchronization, as it permits parallel reads of cached data while maintaining data integrity during modifications.

File Loading and Caching Workflow. When a worker thread handles a client upon receiving the request, it initially checks the cache through `cache_lookup(path)`. If the file is located (cache hit), the data is instantly transmitted to the client. If the document isn't stored in the cache (cache miss), the the subsequent actions take place:

1. The thread acquires a write lock to ensure exclusive access.
2. The file is opened from the disk using standard I/O routines (`open()`, `read()`, `stat()`).
3. The file content is copied into dynamically allocated memory and stored in a cache slot.
4. The cache metadata (timestamp, size) is updated, and the lock is released.

This system guarantees that a file is loaded into memory by only one thread at any moment, while other threads gain from quick in-memory access after the file is stored in cache.

Cache Coherency and Statistics. Every access to the cache updates its usage meta-data and increments the corresponding statistics counters for *cache hits* and *cache misses*, managed by `src/stats.c`. These counters are protected by semaphores to guarantee atomicity and can be viewed in real time through the `stats_reader` utility. By combining semaphores for atomic counters and reader–writer locks for cache access, the implementation achieves both correctness and high throughput.

Resource Cleanup. At shutdown, the cache is fully cleared by the `cache_destroy()` routine, which iterates over all valid entries, frees allocated memory buffers, and releases synchronization primitives. This prevents memory leaks and ensures that all file descriptors and dynamically allocated structures are properly deallocated before process termination.

In summary, the resource management subsystem integrates an LRU-based caching algorithm with fine-grained synchronization via reader–writer locks. This design allows the server to efficiently handle multiple concurrent file requests while maintaining low response latency and stable memory usage across varying workloads.

2.5 HTTP Processing

3 Challenges and Solutions

The development of a concurrent HTTP server presented several challenges related to synchronization, communication, and system scalability. Designing an architecture that walks the fine line between thread-safety and efficiency required a balancing act of parallelism with data consistency, while ensuring proper cleanup and resource management in all executable scenarios. As such, this section aims to bring light to what were considered the key obstacles in the implementation of the solution we provided.

3.1 Race Conditions

During the early stages of development, one of the most prominent concurrency issues encountered was the presence of **race conditions** in the shared statistics subsystem. The `stats` structure, stored in a shared memory region, was updated concurrently by multiple worker threads each time a request was served. Because these updates occurred without proper synchronization, simultaneous increments of counters such as `requests` and `bytes_sent` occasionally overwrote each other’s values, resulting in lost counts and inconsistent statistics.

Problem Observation. The issue manifested during stress testing, where the total number of processed requests reported by `stats_reader` was consistently lower than the number of actual HTTP requests completed by the server. This discrepancy confirmed that concurrent updates to the same memory region were not atomic and were being interrupted by other threads before completion.

Solution. To ensure atomicity, an **exclusive binary semaphore** was introduced to protect all accesses to the shared `stats` structure. Before any modification, a worker thread performs a `semDown()` operation to acquire exclusive access, updates the relevant counters, and then releases the lock with `semUp()`. This mechanism guarantees that only one thread at a time can modify the shared statistics, effectively eliminating data races and lost updates.

Verification. After the introduction of this semaphore, repeated executions of the `test_concurrent.c` and `stress_test.sh` scripts produced consistent results: the total request count reported by `stats_reader` matched the number of served HTTP re-

sponses. This confirmed that the synchronization strategy successfully enforced atomicity and consistency of shared statistical data across all worker threads.

3.2 Memory Management and Leaks

Memory management played a crucial role in the reliability and robustness of the server. Because the system relies on dynamic memory allocation, shared memory segments, and synchronization primitives created at runtime, any mismanagement could result in persistent resource leaks or instability after repeated executions.

Leak Detection with Valgrind. Throughout development, the **Valgrind** analysis tool was extensively used to detect and diagnose memory leaks, invalid reads or writes, and unfreed heap allocations. Running the server under Valgrind with realistic workloads revealed a small number of unreleased buffers within the cache subsystem and unclosed file descriptors associated with the logging mechanism. These issues were systematically corrected by ensuring that all dynamically allocated buffers are freed, all open files are closed, and all synchronization primitives are properly destroyed before process termination.

Shared Memory and Semaphore Cleanup. The server makes use of POSIX shared memory objects and named semaphores to enable interprocess communication between the main server and auxiliary tools such as `stats_reader`. Improper shutdown or unexpected termination (`Ctrl+C`) initially left behind orphaned resources that persisted in the system, visible through `ipcs` or `ls /dev/shm`. To address this, explicit cleanup routines were implemented in `src/shared_mem.c` and `src/semaphores.c`. These routines perform the following actions:

- Unlink all named semaphores using `sem_unlink()`;
- Unmap and close the shared memory segment via `munmap()` and `shm_unlink()`;
- Free dynamically allocated memory structures associated with cache entries and statistics.

Signal-Driven Resource Management. To guarantee cleanup even in the event of user interruption, the master process installs custom signal handlers for `SIGINT` and `SIGTERM`. When triggered, these handlers invoke a global `cleanup()` function that sequentially joins all worker threads, destroys synchronization objects, releases cache memory, and removes shared memory objects from the system. This ensures a graceful shutdown sequence, preventing resource leakage even under abrupt termination scenarios.

Verification and Stability. Memory management played a crucial role in the reliability and robustness of the server. Because the system relies on dynamic memory allocation, shared memory segments, and synchronization primitives created at runtime, any mismanagement could result in persistent resource leaks or instability after repeated executions.

Leak Detection with Valgrind. Throughout development, the **Valgrind** analysis tool was extensively used to detect and diagnose memory leaks, invalid reads or writes, and unfreed heap allocations. Running the server under Valgrind with realistic workloads revealed a small number of unreleased buffers within the cache subsystem and unclosed file descriptors associated with the logging mechanism. These issues were systematically corrected by ensuring that all dynamically allocated buffers are freed, all open files are closed, and all synchronization primitives are properly destroyed before process termination.

Shared Memory and Semaphore Cleanup. The server makes use of POSIX shared memory objects and named semaphores to enable interprocess communication between the main server and auxiliary tools such as `stats_reader`. Improper shutdown or unexpected termination (`Ctrl+C`) initially left behind orphaned resources that persisted in the system, visible through `ipcs` or `ls /dev/shm`. To address this, explicit cleanup routines were implemented in `src/shared_mem.c` and `src/semaphores.c`. These routines perform the following actions:

- Unlink all named semaphores using `sem_unlink()`;
- Unmap and close the shared memory segment via `munmap()` and `shm_unlink()`;
- Free dynamically allocated memory structures associated with cache entries and statistics.

Signal-Driven Resource Management. To guarantee cleanup even in the event of user interruption, the master process installs custom signal handlers for `SIGINT` and `SIGTERM`. When triggered, these handlers invoke a global `cleanup()` function that sequentially joins all worker threads, destroys synchronization objects, releases cache memory, and removes shared memory objects from the system. This ensures a graceful shutdown sequence, preventing resource leakage even under abrupt termination scenarios.

Verification and Stability. After the integration of the cleanup logic, repeated Valgrind analyses confirmed that all heap allocations were correctly released, with the final summary reporting:

```
== All heap blocks were freed -- no leaks are possible ==
```

Additionally, post-execution checks using `ipcs -m` and `ipcs -s` verified that no shared memory segments or semaphores remained active after server shutdown. These results demonstrate that the system's memory management strategy is both sound and reliable, ensuring long-term stability and compatibility with operating system resource policies.

3.3 Log Synchronization

Another significant challenge identified during the development of the concurrent HTTP server was the occurrence of **interleaved log entries** in the system log file. Because multiple worker threads perform logging operations simultaneously, concurrent calls to `fprintf()` occasionally resulted in overlapping text output, corrupted lines, or partial messages, making post-execution debugging and auditing difficult.

Problem Analysis. The logging subsystem, implemented in `src/logger.c`, originally allowed each worker thread to write directly to the shared log file without synchronization. During concurrent request handling, multiple threads invoked `logger_write()` almost simultaneously, causing their respective I/O streams to interfere. Since standard C file I/O is not inherently atomic, different threads could write portions of separate log entries into the same buffer region, producing garbled output such as:

```
[Thread 2] Served file index.html [Thread 3] Client disconnected
[Thread 5] Serv[Thread 4] Error 404: file not found
```

This problem not only compromised readability but also impeded accurate debugging and event tracing.

Implemented Solution. To ensure the integrity of log entries, a **mutual exclusion mechanism** was introduced around all file-writing operations. A *binary semaphore* named `/ws_log_sem` is created during server initialization and shared among all threads. Whenever a worker intends to write to the log, it executes a `semDown()` (wait) operation before writing and a `semUp()` (post) operation afterward, guaranteeing exclusive access to the log stream during the write.

In addition, the `src/thread_logger.c` module provides each thread with a small local buffer used to assemble the entire log message before issuing a single, atomic write operation to the shared file. This design minimizes the duration of critical sections and reduces contention between threads.

Example Implementation. The final logging sequence for each worker thread is as follows:

1. Format the log message locally using `snprintf()` into a thread-local buffer.
2. Acquire the global logging semaphore using `semDown()`.
3. Write the buffered message to the log file using `fprintf()` or `write()`.
4. Flush the stream with `fflush()` to ensure persistence.
5. Release the semaphore using `semUp()`.

This approach ensures that only one thread writes to the file at a time and that each log entry is complete and contiguous.

Verification. Following the implementation of the synchronization mechanism, repeated stress tests involving dozens of concurrent clients confirmed that all log entries were well-formed, non-overlapping, and timestamped correctly. Furthermore, log inspection tools and diff comparisons across multiple runs revealed identical, reproducible outputs under equivalent workloads, validating that the synchronization scheme effectively eliminated race conditions in the logging subsystem.

In conclusion, the combination of a global binary semaphore for mutual exclusion and per-thread buffering provided a robust and efficient solution to the log interleaving problem, ensuring reliable and deterministic logging under concurrent workloads.

3.4 Zombies and Signals

During development, one critical aspect of maintaining process hygiene in the server was the prevention of **zombie processes**. A zombie process occurs when a child process terminates but its exit status remains uncollected by the parent, leaving a defunct entry in the system's process table. Although the concurrent HTTP server primarily relies on threads for parallelism, auxiliary processes such as the master and worker components, or other spawned utilities, can still generate zombies if not properly reaped.

Problem Description. Early test executions revealed lingering defunct processes after multiple start-and-stop cycles of the server. Inspection using the `ps -ef` and `top` commands showed entries in the `Z` (zombie) state, indicating that terminated child processes had not been reaped by the master. This occurred because no signal handler was initially configured to capture `SIGCHLD`, the signal sent to a parent when one of its children exits.

Implemented Solution. To prevent the accumulation of zombie processes, a dedicated signal handler for `SIGCHLD` was registered in the master process (see `src/master.c`). This handler performs a non-blocking call to `waitpid()` with the `WNOHANG` option, which immediately reaps any terminated child processes without interrupting the server's main accept loop. The implementation follows the canonical pattern:

```
void sigchld_handler(int signo) {
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

This ensures that all child processes are properly cleaned up as soon as they terminate, preventing accumulation of zombies and maintaining a clean process table.

Graceful Shutdown Signals. In addition to `SIGCHLD`, the master process also handles `SIGINT` and `SIGTERM` to ensure graceful server termination. When either of these signals is received, the signal handler triggers a global cleanup routine that:

- Closes all open sockets and joins all worker threads.
- Releases semaphores and shared memory objects.
- Calls `logger_close()` and `cache_destroy()` to release I/O and heap resources.

By installing these handlers during initialization, the server guarantees that even abrupt termination via keyboard interrupt (`Ctrl+C`) or system kill command results in full resource reclamation and no lingering processes.

Verification. To confirm correct signal handling and zombie prevention, repeated executions of the server under heavy load were followed by process table inspection using:

```
$ ps -ef | grep server
```

No processes in the `Z` (zombie) state were observed after shutdown, and the system returned all allocated process slots to the kernel immediately. These results demonstrate that the combination of `SIGCHLD` reaping and structured termination handlers effectively

eliminates zombie processes and ensures consistent process hygiene in the concurrent server.

In summary, proper handling of `SIGCHLD`, `SIGINT`, and `SIGTERM` signals guarantees that the server remains free of zombie processes and that all resources are correctly reclaimed during both normal and forced termination.

4 Testing Methodology

The validation of the concurrent HTTP server was performed through a structured testing framework combining automated functional validation, concurrency stress testing, and static and dynamic analysis tools. Testing focused on ensuring that the system correctly implemented HTTP semantics, handled concurrent requests safely, and maintained stability under heavy load conditions.

4.1 Functional Tests

Functional verification was carried out using a series of automated shell scripts and manual HTTP inspections. The main script, `tests/test_suite.sh`, executes end-to-end validation routines that compile the server, launch it in a controlled environment, and issue a predefined set of HTTP requests using the `curl` utility.

HTTP Response Validation. Each request is checked for proper protocol conformance, including:

- Correct HTTP status codes: 200 OK, 404 Not Found, and 500 Internal Server Error.
- Accurate Content-Length and Content-Type (MIME type) headers.
- Proper file serving from the `www/` directory and custom error pages from `www/errors/`.

The validation phase also covers malformed or non-existent URLs to ensure the server's error-handling path responds predictably and consistently.

Automated Makefile Integration. The testing scripts are directly integrated into the project's Makefile through the targets:

```
make test      # Execute functional validation via test_suite.sh
make stress    # Execute load testing via stress_test.sh
```

This integration guarantees that each testing cycle runs against a cleanly rebuilt binary (`bin/webserver`) and uses consistent configuration parameters (`server.conf`), providing reproducible and environment-independent results.

Verification of MIME Type Handling. To ensure that different file types were correctly served, the server's response headers were compared against expected MIME types defined by the `http_builder.c` module. For instance, `.html`, `.css`, and `.png` files were validated to produce `text/html`, `text/css`, and `image/png` respectively. This confirmed the correctness of both request parsing and response construction layers.

Result Summary. All functional tests consistently passed, confirming proper routing of static files, reliable handling of error responses, and complete protocol compliance. Log inspection confirmed that each request generated a corresponding, timestamped entry without corruption or duplication.

4.2 Concurrency and Stress Tests

To validate synchronization correctness and scalability, the project included both native and external concurrency testing tools. The goal was to ensure that multiple simultaneous clients could interact with the server without introducing race conditions, deadlocks, or resource starvation.

Automated Concurrency Validation. The `tests/test_concurrent.c` program was developed to create a controlled, reproducible concurrency test harness. It spawns multiple client threads that simultaneously perform HTTP requests, validating that:

- No request data is lost or partially transmitted.
- Response latency remains stable under load.
- Shared statistics and cache metrics remain consistent across threads.

This tool directly interfaces with the server’s shared memory and logging subsystems, allowing precise validation of synchronization correctness.

Load Testing with Apache Bench. In addition to internal tools, the open-source benchmarking utility **ApacheBench** (**ab**) was used to simulate large-scale real-world traffic. Typical test commands included:

```
ab -n 500 -c 50 http://localhost:8080/index.html
```

where `-n` specifies the total number of requests and `-c` the concurrency level. Metrics such as average request latency, throughput, and connection error rate were recorded. The server demonstrated stable throughput and no data corruption even beyond 100 concurrent requests.

Race Detection with Helgrind and ThreadSanitizer. Dynamic analysis was further strengthened using **Helgrind** (a Valgrind tool) and **ThreadSanitizer** (**TSan**) to detect hidden race conditions in the multithreaded subsystems. Running the server under these tools verified that:

- No concurrent data races occurred within the cache, logger, or statistics modules.
- All synchronization primitives (mutexes, semaphores, and reader–writer locks) were properly initialized and destroyed.
- Condition variables and thread joins were correctly paired, avoiding deadlocks or missed signals.

These analyses confirmed the internal thread-safety of the implementation, complementing the results of the external stress tests.

Stress and Stability Evaluation. The `tests/stress_test.sh` script launched high-frequency, randomized HTTP requests against the server for extended durations. Monitoring through the `stats_reader` utility confirmed steady-state operation, with no memory leaks or semaphore exhaustion observed during multi-minute runs.

Summary. The combination of internal concurrency tests, external load simulation, and dynamic race-detection tools provided comprehensive coverage of the system’s concurrency model. The results verified that synchronization primitives functioned correctly under sustained load and that the server maintained stable performance without data races or resource exhaustion.

5 Performance Analysis

5.1 Test Environment

The performance evaluation was made on a machine with the following specifications:
The performance evaluation was made on a machine with the following specifications:

- **CPU:** Intel Core i7-13700H (14 cores, 20 threads, Up to 5.0GHz)
- **GPU 0:** Integrated Intel Iris Xe Graphics
- **GPU 1:** NVIDIA GeForce RTX 4050 Laptop GPU
- **Disk:** 1TB NVMe SSD
- **RAM:** 16 GB DDR5 The server was configured with the following parameters in `server.conf`:
 - **Port:** 8080
 - **TIMEOUT_SECONDS:** 30
 - **NUM_WORKERS:** 4
 - **THREADS_PER_WORKER:** 10

5.2 Results: Throughput

- **Port:** 8080
- **TIMEOUT_SECONDS:** 30
- **NUM_WORKERS:** 4
- **THREADS_PER_WORKER:** 10

5.3 Results: Throughput

The server was configured with the following parameters in `server.conf`: ““

- **Port:** 8080
- **TIMEOUT_SECONDS:** 30

- **NUM_WORKERS:** 4
- **THREADS_PER_WORKER:** 10

5.4 Results: Throughput

Concurrency	Req/s (No Cache)	Req/s (With Cache)
10	24686.12	27754.49
100	23307.52	23467.68

Table 1: Throughput comparison with and without cache

5.5 Results: Latency

The following table presents the average latency per request (in milliseconds) measured under different concurrency levels.

Concurrency	Latency (No Cache)	Latency (With Cache)
10	0.419 ms	0.399 ms
100	4.534 ms	4.239 ms

Table 2: Average latency comparison with and without cache

5.6 Discussion of Results

6 Conclusion