universidade de aveiro

deti
departamento de eletrónica,
telecomunicações e informática

User Manual

Concurrent HTTP Web Server

Operating Systems

Academic Year 2025/2026

Martim Gil
Student ID: 124833

Nuno Leite Faria
Student ID: 112994

Supervisor: Prof. Pedro Azevedo Fernandes

December 12, 2025

Contents

1 Introduction 2

2 System Requirements 2
2.1 Software Requirements . . . 2
2.2 Dependencies and Libraries . . . 3
2.3 Test Environment . . . 3
2.4 User Permissions . . . 3

3 Compilation 3
3.1 Build System Overview . . . 3
3.2 Standard Build . . . 4
3.3 Running the Server . . . 4
3.4 Cleaning Build Artifacts . . . 4
3.5 Debug and Analysis Builds . . . 4
3.6 Testing Integration . . . 5
3.7 Help and Documentation . . . 6

4 Configuration 6
4.1 Editing the Configuration File . . . 6
4.2 Configuration Parameters . . . 7
4.3 Runtime Behavior . . . 7
4.4 Parameter Requirements . . . 8
4.5 Validation and Error Handling . . . 8
4.6 Reloading Configuration . . . 8

5 Execution 8
5.1 Starting the Server . . . 8
5.2 Graceful Shutdown . . . 9

6 Usage Examples 9
6.1 Basic HTTP Request . . . 9
6.2 Accessing from a Web Browser . . . 10
6.3 Monitoring Logs . . . 10
6.4 Checking Live Statistics . . . 10
6.5 Concurrent Request Example . . . 11
6.6 Error Handling Verification . . . 11
6.7 Web Dashboard (Bonus Feature) . . . 11
6.8 Graceful Shutdown . . . 13

7 Troubleshooting 13
7.1 Compilation Issues . . . 13
7.2 Configuration and Startup Issues . . . 14
7.3 Runtime and Concurrency Issues . . . 15
7.4 Testing and Validation Problems . . . 16

1 Introduction

This user manual provides guidance for installing, configuring, and operating the Concurrent HTTP Server. The system is designed as a multi-threaded web server capable of handling multiple simultaneous client connections efficiently while ensuring synchronization, reliability, and safe resource management.

The primary objective of the Concurrent HTTP Server is to deliver static web content—such as HTML pages, CSS stylesheets, and multimedia files—directly from the local `www/` directory to connected clients via the HTTP/1.0 protocol. Internally, the server employs a hybrid process–thread model that combines interprocess communication (IPC), shared memory statistics tracking, and a thread pool for request handling. This design allows the server to maintain high throughput and responsiveness under concurrent workloads without compromising stability.

The manual is intended for users who wish to:

- Build and launch the server from source code using the provided `Makefile`.
- Understand the directory structure, configuration files, and runtime behavior.
- Execute standard and stress tests to validate the server's performance.
- Monitor real-time server statistics and logs through auxiliary utilities.

Each subsequent section of this document describes a specific aspect of system usage, including installation steps, configuration parameters, command-line execution, monitoring, and troubleshooting procedures. Screenshots, sample commands, and expected outputs are provided where appropriate to facilitate smooth deployment and operation.

In summary, this manual serves as a comprehensive reference for both technical and non-technical users to effectively install, operate, and evaluate the concurrent HTTP server in a POSIX-compliant environment.

2 System Requirements

The Concurrent HTTP Server is designed to operate in a POSIX-compliant environment supporting multi-threading and interprocess communication. This section outlines the minimum and recommended system requirements needed to build, execute, and evaluate the application successfully.

2.1 Software Requirements

The server and its utilities rely on standard POSIX libraries and tools commonly available on Unix-like operating systems.

- **Operating System:** Linux or any POSIX-compliant system.
- **Compiler:** GCC version 9.0 or higher (tested with GCC 13).
- **Build System:** GNU `make` utility for automated compilation and testing.
- **Threading Support:** POSIX Threads (pthreads) library.

- **IPC Support:** POSIX Shared Memory (`shm_open`) and Semaphores (`sem_open`).
- **Shell Environment:** Bash 4.0+ (for executing test scripts such as `test_suite.sh` and `stress_test.sh`).
- **Optional Tools:**
  - `curl` — for manual HTTP testing.
  - `ab` (ApacheBench) — for concurrency and performance testing.
  - `valgrind`, `helgrind`, or `tsan` — for memory and thread-safety analysis.

2.2 Dependencies and Libraries

All required dependencies are part of the standard GNU/Linux environment. The following libraries are automatically linked during the build process via the `Makefile`:

- `-pthread` — for thread creation and synchronization.
- `-lrt` — for POSIX real-time extensions used in shared memory and semaphores.
- `-lm` — for standard mathematical functions.

2.3 Test Environment

The system was developed and validated on Ubuntu 24.04 LTS (64-bit) with kernel version 6.x and GCC 13.2.0. All functional and concurrency tests were executed locally on both physical and virtualized environments without external dependencies.

2.4 User Permissions

Running the server requires user-level permissions only. Administrator privileges (`sudo`) are not required unless the user wishes to bind the server to a privileged port (below 1024). For standard operation, the default configuration uses TCP port 8080.

3 Compilation

The Concurrent HTTP Server project is built using a GNU `Makefile` that automates compilation, linking, and testing tasks. All compilation rules are compatible with POSIX systems using the `gcc` compiler and the GNU `make` utility.

3.1 Build System Overview

The Makefile separates source code, build artifacts, and final binaries into dedicated directories:

```
src/      → C source files and headers
build/    → Intermediate object (.o) and dependency (.d) files
bin/      → Final executables (webserver, stats_reader)
tests/    → Test scripts and utilities
```

The compilation process automatically generates dependency files using the compiler flags `-MMD -MP`, allowing incremental builds to recompile only modified files. POSIX threading and real-time libraries are linked through the flags `-pthread` and `-lrt`, ensuring full compatibility with shared memory and semaphore operations.

3.2 Standard Build

To compile the complete project (main server and statistics reader), navigate to the project root and execute:

```
make all
```

This target performs the following steps:

- Creates required directories (`build/` and `bin/`).
- Compiles all C sources located in `src/`.
- Links the main server executable `bin/webserver`.
- Builds the auxiliary utility `bin/stats_reader`.

On success, both executables are available in the `bin/` directory:

```
bin/
|-- webserver      -> Main HTTP server binary
‘-- stats_reader   -> Shared memory monitoring utility
```

3.3 Running the Server

Once compiled, the server can be launched directly via the Makefile:

```
make run
```

This command builds the project (if necessary) and executes `./bin/webserver` with default configuration parameters specified in `server.conf`.

3.4 Cleaning Build Artifacts

To remove all intermediate and binary files, execute:

```
make clean
```

This command deletes the `build/` and `bin/` directories, ensuring a clean state for subsequent recompilation or testing.

3.5 Debug and Analysis Builds

The Makefile includes predefined targets for debugging and race detection:

- **Debug Mode** — builds the project with extended debug symbols and preprocessor macros:

  ```
          make debug
  ```

  This enables additional logging and verbose output, suitable for development and inspection via `gdb` or similar tools.
- **Thread Sanitizer (TSan)** — compiles the project with the GCC/Clang Thread Sanitizer:

  ```
          make tsan
          ./bin/webserver
  ```

  This mode detects race conditions and synchronization misuse at runtime.
- **Helgrind (Valgrind Tool)** — while not a separate Make target, race detection can also be run through:

  ```
          valgrind --tool=helgrind ./bin/webserver
  ```

  This approach identifies data races and lock-order inversions in multithreaded execution.
- **Valgrind Leak Check** — to verify proper resource cleanup:

  ```
          valgrind --leak-check=full ./bin/webserver
  ```

  confirming that all heap, shared memory, and semaphore allocations are correctly freed upon shutdown.

3.6 Testing Integration

Automated validation is directly integrated into the build system via the following targets:

```
make test          # Executes the test suite (tests/test_suite.
   sh)
make build-tests   # Compiles test binaries (e.g.,
   test_concurrent.c)
```

The `make test` target compiles and launches the server before running a collection of automated scripts that verify HTTP compliance, concurrency safety, and log consistency. Additional stress tests can be launched via:

```
./tests/stress_test.sh
```

to evaluate performance under sustained load.

3.7 Help and Documentation

The Makefile provides a built-in help command summarizing all available targets:

```
make help
```

This command lists build, clean, test, and debug options with brief descriptions, ensuring that all project operations can be performed consistently from the command line.

4 Configuration

The behavior of the Concurrent HTTP Server is governed by the configuration file `server.conf`, located in the project root directory. This file defines all runtime parameters, including network settings, concurrency levels, and resource allocation limits. The configuration is parsed at startup by the `config.c` module, which reads each key–value pair, validates the input, and applies the settings to the master and worker processes.

If the file is missing or a parameter is invalid, the server automatically falls back to default values defined in the source code. Any configuration error or invalid syntax is reported to the console and logged in the main log file.

4.1 Editing the Configuration File

The configuration file uses a simple line-based `KEY = VALUE` format. Comments are supported using the `#` symbol at the start of a line. For example:

```
# Example server configuration
PORT = 8080
DOCUMENT_ROOT = www
NUM_WORKERS = 4
THREADS_PER_WORKER = 10
MAX_QUEUE_SIZE = 100
LOG_FILE = logs/server.log
CACHE_SIZE_MB = 10
```

Blank lines and extra whitespace are ignored. All paths can be specified relative to the project root or as absolute filesystem paths.

4.2 Configuration Parameters

Parameter	Example	Description
`PORT`	8080	TCP port on which the master process listens for incoming HTTP connections. Must be greater than 1024 for non-root users.
`TIMEOUT_SECONDS`	30	Maximum duration (in seconds) that an idle client connection may remain open. After this period, inactive connections are automatically closed to conserve resources.
`DOCUMENT_ROOT`	./www	Directory containing the static web content (HTML, CSS, JavaScript, images, and other resources) served by the server. Paths can be absolute or relative to the project root.
`NUM_WORKERS`	4	Number of worker processes forked by the master process. Each worker manages an independent thread pool for request handling.
`THREADS_PER_WORKER`	10	Number of threads created within each worker process. Each thread handles one client connection at a time, allowing parallel request servicing.
`MAX_QUEUE_SIZE`	100	Maximum number of pending client requests in the bounded task queue between the acceptor and worker threads. Higher values increase tolerance to bursts of connections but may consume more memory.
`CACHE_SIZE_MB`	10	Size (in megabytes) of the Least Recently Used (LRU) cache maintained by each worker. Larger caches improve response times for frequently requested files at the cost of additional memory usage.
`LOG_FILE`	access.log	Path to the access log file where all client requests and system messages are recorded. Writes are synchronized across processes using semaphores to avoid interleaved output.

Table 1: Updated configuration parameters as defined in `server.conf`.

4.3 Runtime Behavior

When the server starts, the `master.c` process reads and applies these parameters before spawning worker processes. Each worker initializes its own LRU cache and thread pool according to the configuration values. The `logger.c` module initializes a shared semaphore-protected log file defined by `LOG_FILE`, ensuring non-interleaved writes from multiple threads.

Configuration values such as `MAX_QUEUE_SIZE` and `CACHE_SIZE_MB` directly affect performance and memory footprint. Increasing these parameters can improve throughput but requires more system memory and may impact scheduling latency.

4.4 Parameter Requirements

Required keys. The following keys must be present and valid:

- `PORT` (integer > 0 and typically > 1024 for non-root users)
- `DOCUMENT_ROOT` (existing directory)
- `NUM_WORKERS` (positive integer)
- `THREADS_PER_WORKER` (positive integer)
- `MAX_QUEUE_SIZE` (non-negative integer)
- `CACHE_SIZE_MB` (non-negative integer)
- `LOG_FILE` (writable path)
- `TIMEOUT_SECONDS` (non-negative integer)
- `LOG_LEVEL` (`DEBUG|INFO|WARN|ERROR`)

4.5 Validation and Error Handling

Each configuration parameter is validated at startup:

- Invalid numeric values (negative or zero) are rejected and replaced with defaults.
- Nonexistent directories in `DOCUMENT_ROOT` or `LOG_FILE` paths trigger warnings and cause fallback creation.
- Invalid or unknown keys are ignored but reported in the log.

If any critical parameter (such as the listening port) cannot be initialized, the server prints an error message and terminates gracefully.

4.6 Reloading Configuration

Currently, configuration parameters are loaded only at startup. To apply changes, the server must be restarted:

```
make run
# or
./bin/webserver
```

This approach ensures deterministic initialization and avoids inconsistencies during live configuration updates.

5 Execution

5.1 Starting the Server

After compilation, start the server with:

```
./bin/webserver
# Or via make:
make run
```

5.2 Graceful Shutdown

To stop the server correctly (releasing shared memory and closing sockets):

1. Press **Ctrl+C** (sends `SIGINT`) in the terminal running the server.
2. The Master process will catch the signal, notify workers, and clean up resources.
3. Wait for the "Server shutdown complete" message.

6 Usage Examples

This section provides practical examples of how to interact with the Concurrent HTTP Server once it has been compiled and configured. The examples assume that the server has been started successfully using either:

```
make run
# or
./bin/webserver
```

and that it is listening on TCP port 8080 as defined in the configuration file (`server.conf`).

6.1 Basic HTTP Request

The server delivers static files stored in the directory specified by the `DOCUMENT_ROOT` parameter (default: `www/`). To test the server using the command line, use `curl` to request a file:

```
curl -v http://127.0.0.1:8080/index.html
```

The `-v` flag enables verbose output, displaying HTTP headers and response codes. If the file exists in the `www/` directory, the server responds with:

```
> GET /index.html HTTP/1.0
< HTTP/1.0 200 OK
< Content-Type: text/html
< Content-Length: 512
```

If the file is not found, the server returns the predefined error page:

```
< HTTP/1.0 404 Not Found
< Content-Type: text/html
< Content-Length: 238
```

To view a list of other available files:

```
ls www/
```

6.2 Accessing from a Web Browser

After launching the server, open any browser and navigate to:

`http://localhost:8080`

You should see the default landing page (`index.html`) loaded from the local `www/` directory. The server will log each access in the file defined by the `LOG_FILE` parameter (default: `logs/server.log`).

6.3 Monitoring Logs

All requests and internal events are recorded in the log file. To view logs in real time:

```
tail -f logs/server.log
```

Each log entry contains a timestamp, process ID, thread ID, client IP, and the requested resource. Concurrent access from multiple clients is safely synchronized through semaphores, ensuring clean, non-interleaved lines.

6.4 Checking Live Statistics

The server maintains real-time statistics on shared memory, including request counts, cache hits, and bytes served. These can be viewed using the monitoring utility:

```
./bin/stats_reader
```

Typical output:

```
=== Server Statistics ===
Total Requests=N
Bytes Requests=X
Status_200=Y
Status_404=Z
Status_500=A
Active Connections=B
Total_response_time_ms=C
Avg_response_time_ms=D
=========================
*The letters correspond to variable numbers
```

Statistics are refreshed periodically (default: every 30 seconds). If run in a separate terminal, `stats_reader` provides a non-intrusive live view of the server's performance.

6.5 Concurrent Request Example

To simulate multiple simultaneous clients, use the Apache Benchmark tool (`ab`):

```
ab -n 200 -c 20 http://127.0.0.1:8080/index.html
```

where:

- `-n 200` — total number of requests.
- `-c 20` — number of concurrent clients.

The output displays average latency, throughput, and failed request counts, allowing quick performance evaluation under load.

6.6 Error Handling Verification

To test custom error pages:

```
curl -v http://127.0.0.1:8080/nonexistent.html
```

Expected response:

```
< HTTP/1.0 404 Not Found
< Content-Type: text/html
< Content-Length: 238
```

This confirms that the server correctly maps missing files to the predefined error template located in `www/errors/404.html`.

6.7 Web Dashboard (Bonus Feature)

The project includes a fully self-contained Web Dashboard, implemented in `www/dashboard.html`. This optional component provides a modern browser-based interface for real-time server monitoring. It uses **TailwindCSS** for layout and design, **Chart.js** for visualization, and periodically fetches live metrics from the backend endpoint `/api/stats` in JSON format.

Accessing the Dashboard. After starting the server, open any web browser and navigate to:

`http://localhost:8080/dashboard.html`

The page is automatically refreshed every two seconds to display current server statistics.

Displayed Information. The dashboard presents an overview of runtime metrics including:

- **Total Requests** – cumulative number of handled HTTP requests.
- **Data Transferred** – total outgoing bandwidth.
- **Average Response Time** – mean latency in milliseconds.
- **Active Connections** – number of concurrent client sessions.
- **HTTP Status Distribution** – counters for 200, 404, and 500 responses.
- **Cache Performance** – hit/miss counts and calculated hit rate.
- **Requests per Second Chart** – dynamic time-series graph showing recent load levels.

Technology Stack.

- **TailwindCSS** is loaded via CDN to provide the responsive dark interface without additional assets.
- **Chart.js** (also via CDN) renders a continuously updated line chart.
- The dashboard logic is implemented directly in inline JavaScript, without external dependencies or build steps.

Data Source (`/api/stats`). The page periodically issues HTTP `GET` requests to `/api/stats`. The server must expose this endpoint and return a JSON object with the following fields (example):

```
{
  "total_requests": A,
  "bytes_transferred": B,
  "avg_response_time_ms": C,
  "active_connections": D,
  "status_codes": { "200": X, "404": Y, "500": Z },
  "cache": { "hits": E, "misses": F, "hit_rate": E/(E+F) * 100
     }
}
*the letters are number placeholders that can change depeding
   on the number of requests and their volume
```

Troubleshooting.

- **404 Not Found:** Ensure `dashboard.html` is present in the document root defined in `server.conf`.
- **Disconnected Status:** The indicator will turn red if `/api/stats` is unreachable or returns invalid JSON. Verify the backend implementation with:

```
curl http://localhost:8080/api/stats
```

- **Chart Not Updating:** Clear browser cache or refresh the page.

Summary. The Web Dashboard provides an intuitive and visually rich way to monitor the HTTP server's real-time performance. It requires no installation or additional dependencies and is accessible at:

`http://localhost:8080/dashboard.html`

6.8 Graceful Shutdown

To terminate the server, use:

```
Ctrl + C
```

The server handles the `SIGINT` signal gracefully by:

- Closing all open sockets.
- Releasing shared memory and semaphores.
- Flushing log buffers to disk.

After shutdown, you can verify resource cleanup using:

```
ipcs -m     # Shared memory segments
ipcs -s     # Semaphores
```

No remaining entries should appear under the server's process user.

7 Troubleshooting

This section lists common issues that may occur during compilation, configuration, or runtime execution of the Concurrent HTTP Server, together with their corresponding causes and recommended solutions. Each problem is derived from real test and validation scenarios observed during the system's development and evaluation.

7.1 Compilation Issues

1. Missing Libraries or Header Files. Symptom:

```
fatal error: pthread.h: No such file or directory
fatal error: semaphore.h: No such file or directory
```

Cause: Required POSIX development libraries (`libpthread`, `librt`) are missing.

Solution: Install the development toolchain for POSIX libraries:

```
sudo apt-get install build-essential
sudo apt-get install manpages-posix-dev
```

Then recompile using:

```
make clean && make all
```

2. Undefined Reference Errors at Link Time. Symptom:

```
undefined reference to ‘sem_open’ or ‘shm_open’
```

Cause: The real-time library (`-lrt`) is not linked properly.

Solution: Ensure the Makefile includes:

```
LDFLAGS += -lrt
```

(Already included by default in this project.)

7.2 Configuration and Startup Issues

3. “Port Already in Use” Error. Symptom:

```
Error: Failed to bind socket on port 8080
```

Cause: Another process (possibly a previous server instance) is already listening on the same port.

Solution:

- Terminate any running instance of the server:

  ```
  sudo lsof -i :8080
  kill <PID>
  ```

- Alternatively, change the port in `server.conf` to an unused value (e.g., 8081).

4. “Permission Denied” on Low Ports. Symptom:

```
bind(): Permission denied
```

Cause: Non-root users cannot bind to privileged ports (below 1024).

Solution: Either use a non-privileged port (e.g., 8080) in `server.conf`, or start the server with elevated privileges:

```
sudo ./bin/webserver
```

5. Missing or Invalid Document Root. Symptom:

```
Error: Failed to open document root directory 'www'
```

Cause: The directory specified by `DOCUMENT_ROOT` does not exist.

Solution: Ensure the directory exists and contains at least one valid file:

```
mkdir -p www
echo "<h1>Server is running!</h1>" > www/index.html
```

6. Invalid Configuration Entries. Symptom:

```
Warning: Invalid line in server.conf: "THREADS_PER_WORKER = -2"
```

Cause: Numeric parameters cannot be negative or zero.

Solution: Edit `server.conf` to ensure all values are positive integers:

```
THREADS_PER_WORKER = 4
```

7.3 Runtime and Concurrency Issues

7. No Response to HTTP Requests. Symptom:

```
curl: (7) Failed to connect to 127.0.0.1 port 8080: Connection
    refused
```

Cause: The server process is not running, or the master process failed to spawn workers.

Solution: Run the server manually in the foreground to view debug messages:

```
make run
```

Check for errors such as "Failed to fork worker" or "Invalid socket descriptor."

8. Interleaved or Corrupted Logs. Symptom: Multiple log entries overlap in the same line of `logs/server.log`.

Cause: This may occur if the logging semaphore was removed or not initialized properly.

Solution:

```
pkill webserver
sem_unlink /ws_log_sem
make run
```

The `master.c` process recreates the semaphore on startup.

9. Shared Memory Not Released After Shutdown. Symptom:

```
ipcs -m
------ Shared Memory Segments --------
0x00000000  12345  user  666  4096
```

Cause: The server terminated abnormally (e.g., via `kill -9`), skipping cleanup routines.

Solution: Manually remove IPC objects:

```
ipcrm -m 12345   # Shared memory
ipcrm -s 54321   # Semaphore
```

Then restart the server using `make run`.

10. High CPU Usage or Slow Response. Cause: Cache size too small or thread count too low for the incoming request rate.

Solution: Increase `THREADS_PER_WORKER` and `CACHE_SIZE_MB` in `server.conf`, or reduce concurrency levels during testing:

```
ab -n 100 -c 10 http://127.0.0.1:8080/
```

7.4 Testing and Validation Problems

11. Test Scripts Fail to Execute. Symptom:

```
./tests/test_suite.sh: Permission denied
```

Cause: The test scripts are missing execution permissions.

Solution:

```
chmod +x tests/*.sh
make test
```

12. Stress Test Hangs Indefinitely. Cause: Thread pool exhaustion or blocked request queue due to oversized `MAX_QUEUE_SIZE` and insufficient consumer threads.

Solution:

- Decrease `MAX_QUEUE_SIZE` to balance producer–consumer throughput.
- Monitor shared memory statistics using `./bin/stats_reader`.

```
make debug
./bin/webserver
```

This mode provides detailed trace logs that can help identify configuration or synchronization errors during runtime.