

UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E
INFORMÁTICA

Multi-Threaded Web Server

Relatório Técnico de Implementação e Análise de Performance

Sistemas Operativos – 2025/2026

Autores:

Diogo Ruivo (NMec: 126498)

David Cálix (NMec: 125043)

Turma: P2 | **Grupo:** G7

Dezembro 2025

Conteúdo

1	Introdução	2
2	Arquitetura e Modelo de Concorrência	2
2.1	Processo Master	2
2.2	Processos Workers e Serialized Accept	2
2.3	Thread Pool	2
3	Sincronização e Gestão de Recursos	3
4	Implementações Adicionais (Bónus)	3
4.1	1. Dashboard de Estatísticas em Tempo Real	3
4.2	2. Suporte a HTTP Keep-Alive	4
4.3	3. Virtual Hosts (Hospedagem Virtual)	4
4.4	4. Execução de CGI (Common Gateway Interface)	4
4.5	5. Range Requests (HTTP 206 Partial Content)	4
5	Testes e Validação	5
5.1	Testes de Concorrência e Sincronização	5
5.2	Testes de Memória	5
5.3	Testes de Carga (Apache Bench)	5
6	Conclusão	5

1 Introdução

Este relatório descreve a engenharia e implementação do projeto *Multi-Threaded Web Server*, um servidor web de alto desempenho desenvolvido em C para a unidade curricular de Sistemas Operativos.

O objetivo principal foi criar um sistema robusto capaz de lidar com múltiplas conexões simultâneas, utilizando uma arquitetura híbrida de multiprocessamento e multithreading, suportada por primitivas de sincronização POSIX (Semáforos, Mutexes e Variáveis de Condição).

Para além dos requisitos funcionais base (servir ficheiros estáticos e gestão de cache), o projeto distingue-se pela implementação de um vasto conjunto de funcionalidades avançadas, como suporte a CGI, Virtual Hosts e HTTP Keep-Alive.

2 Arquitetura e Modelo de Concorrência

A arquitetura adotada segue o modelo **Master-Worker Pré-Forked** com **Thread Pools**.

2.1 Processo Master

O Master é responsável pela inicialização dos recursos globais (Memória Partilhada e Semáforos) e pela criação dos processos Worker através de `fork()`. O Master não processa pedidos HTTP; a sua função é monitorizar os sinais do sistema (SIGINT/SIGTERM) para garantir um encerramento limpo e recolher estatísticas globais.

2.2 Processos Workers e Serialized Accept

Optou-se por um modelo de *Serialized Accept* para a distribuição de carga. Todos os Workers herdam o socket de escuta, mas o acesso à chamada de sistema `accept()` é protegido por um semáforo (`queue_mutex`).

Vantagem Técnica: Esta abordagem mitiga o problema do "*Thundering Herd*", onde múltiplos processos acordam desnecessariamente para tratar uma única conexão, garantindo uma distribuição de carga eficiente pelo Kernel do Linux.

2.3 Thread Pool

Cada Worker mantém uma *Thread Pool* fixa (configurável em `server.conf`). O padrão Produtor-Consumidor é implementado internamente:

- **Produtor (Main Thread):** Aceita a conexão e coloca o descritor do socket numa fila ligada.

- **Consumidor (Worker Threads):** Dormem numa Variável de Condição (`pthread_cond_wait`) até haver trabalho, processando o pedido HTTP de forma assíncrona.

3 Sincronização e Gestão de Recursos

A integridade dos dados é garantida através de três níveis de sincronização:

1. **Inter-Processo (Semáforos Nomeados):** Utilizados para proteger recursos partilhados entre processos distintos.
 - `stats_mutex`: Garante a atomicidade na atualização das estatísticas globais em Memória Partilhada (`/dev/shm`).
 - `log_mutex`: Garante que as linhas no ficheiro `access.log` não ficam intercaladas.
 - `queue_mutex`: Serializa o acesso ao `accept()`.
2. **Intra-Processo (Mutexes):** Protegem a fila de tarefas da Thread Pool dentro de cada Worker, evitando *race conditions* entre a thread principal e as threads operárias.
3. **Leitura/Escrita (RWLocks):** Utilizados na **Cache LRU**. Como a leitura de ficheiros em cache é muito mais frequente que a escrita (inserção/remoção), utilizou-se `pthread_rwlock_t`. Isto permite que múltiplas threads leiam o mesmo ficheiro simultaneamente (`rdlock`), bloqueando apenas para escrita (`wrlock`) durante atualizações da cache.

4 Implementações Adicionais (Bónus)

Para além dos requisitos base, foram implementadas **5 funcionalidades extra** que aumentam significativamente a versatilidade e performance do servidor.

4.1 1. Dashboard de Estatísticas em Tempo Real

Descrição: Um painel web dinâmico acessível via `/stats` que mostra o estado interno do servidor. **Implementação:** No ficheiro `src/thread_pool.c`, o servidor deteta o caminho `/stats`. Em vez de procurar um ficheiro, acede à Memória Partilhada (protegida por semáforo), lê os contadores atómicos (uptime, conexões ativas, bytes transferidos) e injeta-os num template HTML construído em memória usando `snprintf`.

- *Benefício:* Permite monitorização sem necessidade de ferramentas externas.

4.2 2. Suporte a HTTP Keep-Alive

Descrição: Permite reutilizar a mesma conexão TCP para múltiplos pedidos. **Implementação:** A lógica de processamento foi alterada para um loop `while(1)`. Se o cliente enviar o cabeçalho HTTP/1.1 (ou não enviar `Connection: close`), o socket não é fechado. Foi aplicado um `SO_RCVTIMEO` (timeout) de 5 segundos no socket. Se o cliente enviar novo pedido dentro desse tempo, é processado imediatamente; caso contrário, a conexão encerra.

- *Benefício:* Reduz drasticamente a latência ao evitar o *handshake* TCP (SYN/ACK) em pedidos subsequentes.

4.3 3. Virtual Hosts (Hospedagem Virtual)

Descrição: Capacidade de alojar múltiplos sites (ex: `site1.local`, `site2.local`) no mesmo servidor/porta. **Implementação:** O parser HTTP extrai o cabeçalho `Host`. O sistema consulta a configuração carregada (`server.conf`) para verificar se existe uma diretória raiz específica para esse domínio (`VHOST_site1...`). Se encontrada, o `DOCUMENT_ROOT` é alterado dinamicamente apenas para esse pedido.

- *Benefício:* Permite alojamento partilhado eficiente.

4.4 4. Execução de CGI (Common Gateway Interface)

Descrição: Suporte para execução de scripts dinâmicos (Python). **Implementação:** Detetada em `src/cgi.c`. Se o ficheiro pedido terminar em `.py`:

1. O servidor cria um `pipe` e executa um `fork()`.
 2. O processo filho redireciona o `STDOUT` para o pipe (`dup2`) e substitui a sua imagem pelo interpretador Python (`execvp`).
 3. O processo pai lê o output do pipe e envia-o como corpo da resposta HTTP.
- *Benefício:* Permite criar páginas dinâmicas e APIs backend.

4.5 5. Range Requests (HTTP 206 Partial Content)

Descrição: Suporte para download parcial de ficheiros (essencial para streaming de vídeo ou pausar/retomar downloads). **Implementação:** O servidor analisa o cabeçalho `Range: bytes=X-Y`. Utiliza `fseek` para saltar para o byte `X` e lê apenas até `Y`. Responde com o código **206 Partial Content** e o cabeçalho `Content-Range`.

- *Benefício:* Grande eficiência na transferência de ficheiros grandes e suporte a media players.

5 Testes e Validação

A robustez do servidor foi validada utilizando uma suite de scripts automatizados (`tests/`).

5.1 Testes de Concorrência e Sincronização

Utilizou-se a ferramenta **Helgrind** (Valgrind) para detetar *data races*.

- **Resultado:** Nenhuma condição de corrida detetada nas estruturas críticas (Cache e Queue). O uso correto de `pthread_mutex` e `sem_t` foi confirmado.

5.2 Testes de Memória

Utilizou-se o **Memcheck** para verificar fugas de memória (*leaks*).

- **Resultado:** O servidor liberta corretamente toda a memória alocada (buffers HTTP, nós da cache, estruturas da thread pool) no encerramento (*Graceful Shutdown*).

5.3 Testes de Carga (Apache Bench)

Sob stress intenso (`ab -n 10000 -c 100`), o servidor manteve-se estável.

- **Keep-Alive:** Aumentou o throughput em cerca de 40% comparado com conexões *close*.
- **Cache:** O tempo de resposta para ficheiros em cache foi inferior a 1ms, validando a eficiência do `rwlock`.

6 Conclusão

O projeto *Multi-Threaded Web Server* resultou num servidor web altamente funcional e performante. A arquitetura escolhida permitiu explorar profundamente os conceitos de IPC e Sincronização.

A implementação das funcionalidades de bónus, nomeadamente o sistema de **CGI** e **Keep-Alive**, elevou a complexidade do projeto, exigindo um controlo rigoroso sobre os descritores de ficheiro e gestão de processos. O sistema final demonstra não apenas estabilidade sob carga, mas também flexibilidade para cenários de uso real.