Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Multi-Threaded Web Server

Relatório Técnico de Implementação, Desafios 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

No contexto da unidade curricular de Sistemas Operativos, foi proposto o desenvolvimento do projeto *Multi-Threaded Web Server*, um servidor web compatível com o protocolo HTTP/1.1, implementado integralmente em linguagem C.

O objetivo primordial deste projeto é a aplicação prática de conceitos fundamentais de sistemas operativos, nomeadamente a gestão de processos, manipulação de threads, comunicação entre processos (IPC) e sincronização de recursos partilhados. Ao contrário de servidores sequenciais simples, o *Multi-Threaded Web Server* foi desenhado para operar num ambiente de produção simulado, onde a concorrência elevada e a eficiência no uso de recursos são requisitos críticos.

Este relatório documenta detalhadamente as decisões de engenharia tomadas, a arquitetura híbrida implementada (Multiprocesso e Multithread), os mecanismos de sincronização utilizados para garantir a integridade dos dados e as funcionalidades avançadas desenvolvidas. Adicionalmente, apresenta-se uma secção dedicada aos desafios técnicos encontrados durante o desenvolvimento e as respetivas soluções, bem como uma análise de performance baseada em testes de carga.

2 Arquitetura e Modelo de Concorrência

Para garantir escalabilidade e robustez, a arquitetura do servidor baseia-se no modelo **Master-Worker Pré-Forked**, complementado internamente por **Thread Pools**. Esta abordagem híbrida permite tirar partido de sistemas multi-core (através de processos) enquanto mantém a leveza na troca de contexto (através de threads).

2.1 Processo Master

O processo Master atua como o gestor global do sistema. É iniciado no arranque da aplicação e tem as seguintes responsabilidades críticas:

- **Setup Inicial:** Leitura do ficheiro de configuração (`server.conf`), criação dos segmentos de Memória Partilhada (`/dev/shm`) e inicialização dos semáforos POSIX nomeados.
- **Gestão de Sockets:** Configuração do socket TCP de escuta na porta especificada.
- **Criação de Workers:** Execução da chamada de sistema `fork()` N vezes, onde N é o número de workers configurados.

- **Supervisão:** O Master não processa pedidos HTTP diretamente. Em vez disso, monitoriza sinais do sistema (`SIGINT`, `SIGTERM`) para coordenar um encerramento limpo (*Graceful Shutdown*), garantindo que nenhum processo "zombie" permanece ativo.
- **Estatísticas:** Periodicamente, o Master acede à memória partilhada para exibir o estado do servidor na consola.

2.2 Processos Workers e Serialized Accept

Cada processo Worker herda o descritor de ficheiro do socket de escuta do Master. Para evitar conflitos e ineficiências na aceitação de novas conexões, optou-se por um modelo de **Serialized Accept**.

2.2.1 O Problema "Thundering Herd"

Numa abordagem ingénua onde todos os processos bloqueiam no `accept()` simultaneamente, a chegada de uma única conexão acordaria todos os processos (dependendo da implementação do Kernel), mas apenas um conseguia a conexão. Isto gera um desperdício de ciclos de CPU conhecido como *Thundering Herd*.

2.2.2 Solução Implementada

Utilizou-se um semáforo (`queue_mutex`) para serializar o acesso. Apenas um Worker de cada vez pode tentar aceitar uma conexão. A lógica simplificada é apresentada abaixo:

```
while (atomic_load(&worker_running)) {
    // 1. Bloquear acesso ao accept (Exclus o M tua entre processos)
    if (sem_wait(sems.queue_mutex) != 0) {
        if (errno == EINTR) break;
        continue;
    }

    // 2. Aceitar a conex o (Kernel Queue)
    int client_fd = accept(server_socket, (struct sockaddr*)&client_addr
    , &len);

    // 3. Libertar IMEDIATAMENTE o mutex para outro worker avan ar
    sem_post(sems.queue_mutex);

    // 4. Despachar para a Thread Pool
    if (client_fd >= 0) {
        thread_pool_dispatch(pool, client_fd);
```

```
    }
}
```

Listing 1: Lógica de Serialized Accept no Worker

2.3 Thread Pool Interna

Dentro de cada Worker, existe uma *Thread Pool* fixa. Esta estrutura evita o custo computacional de criar (`pthread_create`) e destruir threads para cada pedido HTTP. O padrão utilizado é o **Produtor-Consumidor**:

- **Produtor (Thread Principal do Worker):** Aceita a conexão TCP e insere o descritor do socket (`client_fd`) numa fila ligada interna.
- **Consumidor (Worker Threads):** As threads estão em espera passiva numa Variável de Condição (`pthread_cond_wait`). Quando uma tarefa é inserida, uma thread acorda, retira o socket da fila e inicia o processamento do pedido HTTP.

3 Sincronização e Gestão de Recursos

A correção de um sistema concorrente depende da gestão rigorosa do acesso a recursos partilhados. Neste projeto, utilizámos três níveis distintos de primitivas de sincronização.

3.1 Sincronização Inter-Processo (IPC)

Recorreu-se a **Semáforos Nomeados POSIX** para coordenar ações entre processos distintos (Master e vários Workers). Estes semáforos persistem no sistema de ficheiros (geralmente em `/dev/shm`) e permitem acesso atómico a regiões de memória partilhada.

- `stats_mutex`: Protege a estrutura de estatísticas globais (`shared_data_t`). Garante que incrementos de contadores (ex: número de pedidos) não sofrem de *race conditions* quando múltiplos Workers tentam escrever simultaneamente.
- `log_mutex`: Fundamental para o sistema de logs. Garante que as linhas escritas no ficheiro `access.log` não ficam intercaladas ou corrompidas.
- `queue_mutex`: Como referido anteriormente, serializa a chamada `accept()`.

3.2 Sincronização Intra-Processo

Dentro de cada processo, a sincronização entre threads é feita com **Mutexes Pthread** e **Variáveis de Condição**.

- A fila de tarefas da *Thread Pool* é protegida por um `pthread_mutex_t`.
- A sinalização de "nova tarefa disponível" ou "shutdown" é feita via `pthread_cond_signal` e `pthread_cond_broadcast`.

3.3 Cache LRU com Reader-Writer Locks

Um dos componentes mais críticos para a performance é a Cache LRU (*Least Recently Used*). Dado que num servidor web o rácio de leituras é muito superior ao de escritas (inserção/remoção de ficheiros na cache), um Mutex simples seria ineficiente.

Implementou-se a sincronização com **Reader-Writer Locks (`pthread_rwlock_t`)**:

1. **Leitura com Atualização LRU:** Embora a estrutura utilize `pthread_rwlock_t`, optou-se por adquirir o *lock* de escrita (`wrlock`) também durante a leitura (`cache_get`). Isto é estritamente necessário para garantir a consistência dos ponteiros da lista duplamente ligada ao mover o elemento acedido para o topo (política LRU), sacrificando o paralelismo de leitura em prol da estabilidade da memória.
2. **Escrita (`wrlock`):** Quando é necessário inserir um novo ficheiro ou remover um antigo (eviction), a thread adquire um lock exclusivo, bloqueando temporariamente novas leituras até a operação terminar.

Esta estratégia maximiza o *throughput* na entrega de conteúdo estático popular.

4 Implementações Adicionais (Bónus)

Para além dos requisitos funcionais base, a equipa implementou **6 funcionalidades extra** que aumentam a robustez, eficiência e versatilidade do servidor, aproximando-o de uma solução real.

4.1 1. Dashboard de Estatísticas em Tempo Real

Descrição: Um painel web dinâmico acessível via rota `/stats`.

Detalhe Técnico: Ao detetar este caminho, o servidor não procura um ficheiro em disco. Em vez disso, adquire o semáforo da memória partilhada, lê o estado atual (uptime, conexões ativas, bytes transferidos, cache hits) e gera HTML dinamicamente em memória usando `snprintf`. Este mecanismo permite aos administradores monitorizarem a saúde do servidor sem necessitarem de acesso ao terminal.

4.2 2. Suporte a HTTP Keep-Alive

Descrição: Reutilização de conexões TCP para múltiplos pedidos.

Detalhe Técnico: A lógica de processamento foi alterada para um loop `while(1)`.

- Se o cliente enviar `Connection: keep-alive` (ou for HTTP/1.1 padrão), o socket não é fechado após a resposta.
- Foi aplicado um `SO_RCVTIMEO` (timeout) de 5 segundos no socket. Se o cliente não enviar novo pedido dentro desse tempo, o loop termina e a conexão encerra.
- **Impacto:** Redução drástica da latência e carga no CPU ao evitar o *handshake* TCP (SYN/ACK) repetido.

4.3 3. Virtual Hosts (Hospedagem Virtual)

Descrição: Capacidade de alojar múltiplos sites (ex: `site1.local`, `site2.local`) na mesma porta.

Detalhe Técnico: O parser HTTP extrai o cabeçalho `Host`. O sistema consulta a configuração carregada (`server.conf`) para verificar se existe uma diretoria raiz específica (`VHOST_...`) para esse domínio. Se encontrada, o `DOCUMENT_ROOT` é alterado dinamicamente apenas para o contexto desse pedido.

4.4 4. Execução de CGI (Common Gateway Interface)

Descrição: Execução de scripts dinâmicos (Python) no servidor.

Detalhe Técnico: Implementado 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 (`execlp`).

3. O processo pai lê o output do pipe e envia-o como corpo da resposta HTTP.

4.5 5. Range Requests (HTTP 206)

Descrição: Suporte para download parcial de ficheiros, essencial para streaming.

Detalhe Técnico: 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`.

4.6 6. Rotação Automática de Logs

Descrição: Gestão automática do tamanho do ficheiro de registo para não encher o disco.

Detalhe Técnico: Antes de cada escrita em `src/logger.c`, verifica-se o tamanho de `access.log`. Se exceder 10MB, o servidor adquire o `log_mutex`, renomeia o ficheiro atual para `access.log.1` e cria um novo, garantindo a continuidade do serviço.

5 Testes e Validação

A estabilidade do servidor foi validada através de uma suite de testes automatizados (`tests/`).

5.1 Testes Funcionais

O script `test_functional.sh` valida o cumprimento do protocolo HTTP: códigos 200, 404, 403, Content-Types corretos e integridade dos ficheiros descarregados.

5.2 Deteção de Race Conditions

Utilizou-se a ferramenta **Helgrind** (Valgrind) para análise dinâmica de concorrência.

- O script `test_sync.sh` lança o servidor sob Helgrind e bombardeia-o com pedidos concorrentes.
- **Resultado:** Confirmou-se a ausência de *data races* nas estruturas críticas (Cache e Queue), validando o uso correto dos locks.

5.3 Gestão de Memória

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

- **Resultado:** Todos os recursos (buffers, nós da lista ligada da cache, descritores de ficheiro) são libertados corretamente, tanto durante a execução como no encerramento (shutdown).

5.4 Testes de Carga (Apache Bench)

Para quantificar a performance do servidor sob stress intenso, utilizou-se a ferramenta `ab` com 10.000 pedidos e uma concorrência de 100 conexões simultâneas (`ab -n 10000 -c 100`). Os resultados obtidos demonstram a estabilidade e eficiência da arquitetura:

Tabela 1: Resultados dos Testes de Carga (4 Workers, 10 Threads/Worker)

Métrica	**Resultado**
Total de Pedidos	10000
Nível de Concorrência	100
Requests per Second (RPS)	**26913.55**
Time per Request (Mean)	3.716 [ms]
Time per Request (Atomic)	0.037 [ms]
Transfer Rate	27412.93 [Kbytes/sec]
Falhas de Conexão	0

Os dados da Tabela 1 confirmam que o servidor mantém um *throughput* elevado sem degradação do serviço ou falhas de segmentação, validando a robustez do modelo *Serialized Accept* na distribuição de carga.

O valor de 26.000 RPS demonstra que o servidor está provavelmente limitado pelo CPU (CPU-bound) na gestão de syscalls e trocas de contexto, dado que o tamanho dos ficheiros é pequeno e cabem todos na cache. O tempo médio atómico de 0.037ms sugere que a contenda nos semáforos (overhead de sincronização) é baixa.

6 Desafios de Implementação e Soluções

O desenvolvimento de um servidor concorrente em C apresenta desafios únicos, não apenas de lógica, mas de gestão de sistema operativo. Abaixo descrevem-se os problemas mais complexos encontrados e como foram resolvidos.

6.1 1. Processos Órfãos e Zombies

Desafio: Durante o desenvolvimento, ao terminar o servidor com `Ctrl+C`, o processo Master terminava, mas os processos Worker continuavam ativos ou ficavam em estado *Zombie* (`<defunct>`), ocupando entradas na tabela de processos do SO.

Solução: Implementação robusta de *Signal Handling*. O Master captura os sinais `SIGINT` e `SIGTERM`. No *handler*, altera uma variável atómica `keep_running`. O ciclo principal deteta a mudança e inicia a rotina de limpeza: envia `SIGTERM` a todos os PIDs dos filhos e, crucialmente, executa um ciclo de `wait()` para recolher o estado de saída de cada filho.

```
// Master Cleanup
for (int i = 0; i < config->num_workers; i++) {
    if (pids[i] > 0) kill(pids[i], SIGTERM);
}
// Recolher Zombies
for (int i = 0; i < config->num_workers; i++) wait(NULL);
```

Listing 2: Rotina de Cleanup no Master

6.2 2. Consistência da Cache LRU

Desafio: A estrutura de dados da cache é uma lista duplamente ligada com um *Hash Map* (simulado por procura linear neste caso) e gestão de tamanho máximo. Quando múltiplas threads tentavam atualizar a posição de um nó (para o marcar como "mais recente") simultaneamente, os ponteiros `next` e `prev` ficavam corrompidos, levando a *Segmentation Faults* ou ciclos infinitos.

Solução: Adoção estrita de locks de escrita (`pthread_rwlock_wrlock`). Qualquer operação que altere a topologia da lista (inserção, remoção ou reordenação LRU) exige exclusividade total. Além disso, ao ler um item da cache, fazemos uma **cópia profunda** dos dados antes de libertar o lock, para que a thread possa enviar os dados ao cliente sem bloquear a cache para outras threads.

6.3 3. Deadlocks na Rotação de Logs

Desafio: Implementou-se a rotação de logs. No entanto, se ocorresse um erro durante a rotação (ex: falha de disco) e o sistema tentasse registar esse erro no log chamando `log_request` recursivamente, o processo entrava em *Deadlock* porque já possuía o `log_mutex` e tentava adquiri-lo novamente.

Solução: Garantir que o código dentro das zonas críticas de log é "à prova de falhas" e nunca chama funções que possam tentar adquirir o mesmo semáforo. A gestão de erros dentro do sistema de log faz uso de `stderr` direto em vez de tentar usar o sistema de log partilhado. Como a leitura de um item implica a sua reordenação (mover para a cabeça), **as operações de leitura também adquirem exclusividade**, prevenindo a corrupção de ponteiros `next` e `prev`.

6.4 4. Depuração de Memória Partilhada

Desafio: Se o servidor crashasse abruptamente, os segmentos de memória partilhada e semáforos em `/dev/shm` persistiam. Ao tentar reiniciar o servidor, o `sem_open` falhava ou ligava-se a semáforos com estado corrompido.

Solução: Adicionou-se uma rotina de "limpeza preventiva" no início do `main()` do Master e no `Makefile` (target `run`). O servidor tenta desvincular (`shm_unlink`, `sem_unlink`) quaisquer recursos antigos com o mesmo nome antes de tentar criar novos, garantindo sempre um arranque limpo.

7 Conclusão

O projeto *Multi-Threaded Web Server* cumpriu todos os objetivos propostos na unidade curricular, resultando num sistema funcional, estável e eficiente.

A escolha da arquitetura híbrida revelou-se adequada para o problema: o uso de múltiplos processos permitiu robustez (a falha de um worker não derruba o servidor), enquanto o uso de threads permitiu uma concorrência elevada com baixo overhead de memória.

A implementação das funcionalidades de bónus, especialmente o **CGI** e o **Keep-Alive**, elevou a complexidade técnica do projeto, exigindo um domínio aprofundado da API POSIX e do funcionamento interno do protocolo HTTP. Os testes realizados comprovam que o servidor é capaz de lidar com cargas elevadas sem comprometer a integridade dos dados, validando a eficácia dos mecanismos de sincronização implementados.

Em suma, este trabalho permitiu consolidar conhecimentos teóricos sobre exclusão mútua, semáforos e gestão de memória, aplicando-os num cenário prático próximo da realidade da engenharia de software de sistemas.