Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

ConcurrentHTTP Server

Relatório Técnico de Implementação

Sistemas Operativos – 2025/2026

Autores:
Diogo Ruivo (NMec: 126498)
David Cálix (NMec: 125043)

Dezembro 2025

Conteúdo

1 Introdução

Este relatório descreve a implementação do *ConcurrentHTTP*, um servidor web de alto desempenho desenvolvido em C para ambientes Linux. O projeto visa demonstrar a aplicação prática de conceitos avançados de sistemas operativos, nomeadamente multi-processamento, multithreading e sincronização via IPC (Inter-Process Communication).

Os objetivos técnicos alcançados incluem:

1. **Arquitetura Híbrida:** Combinação de processos Worker (para estabilidade e uso de múltiplos cores) com Thread Pools (para concorrência massiva).
2. **Eficiência:** Adoção do modelo *Serialized Accept* para minimizar o overhead de gestão de conexões pelo Kernel.
3. **Robustez:** Gestão rigorosa de memória e recursos, garantindo a inexistência de *memory leaks*.
4. **Funcionalidades Extra:** Implementação de Dashboard em tempo real, Keep-Alive e Rotação de Logs.

2 Arquitetura e Modelo de Execução

O servidor adota uma topologia **Master-Worker** onde a gestão do ciclo de vida é separada do processamento de pedidos.

2.1 Visão Geral e IPC

O processo **Master** é responsável pela inicialização dos recursos partilhados e pela criação dos processos filhos. A comunicação e sincronização baseiam-se em:

- **Memória Partilhada (SHM):** Armazena as estatísticas globais do servidor, acessíveis por todos os Workers para leitura e escrita atómica.
- **Semáforos Nomeados:** Garantem a exclusão mútua em zonas críticas globais, como a escrita no ficheiro de log (`log_mutex`) e a aceitação de conexões (`queue_mutex`).

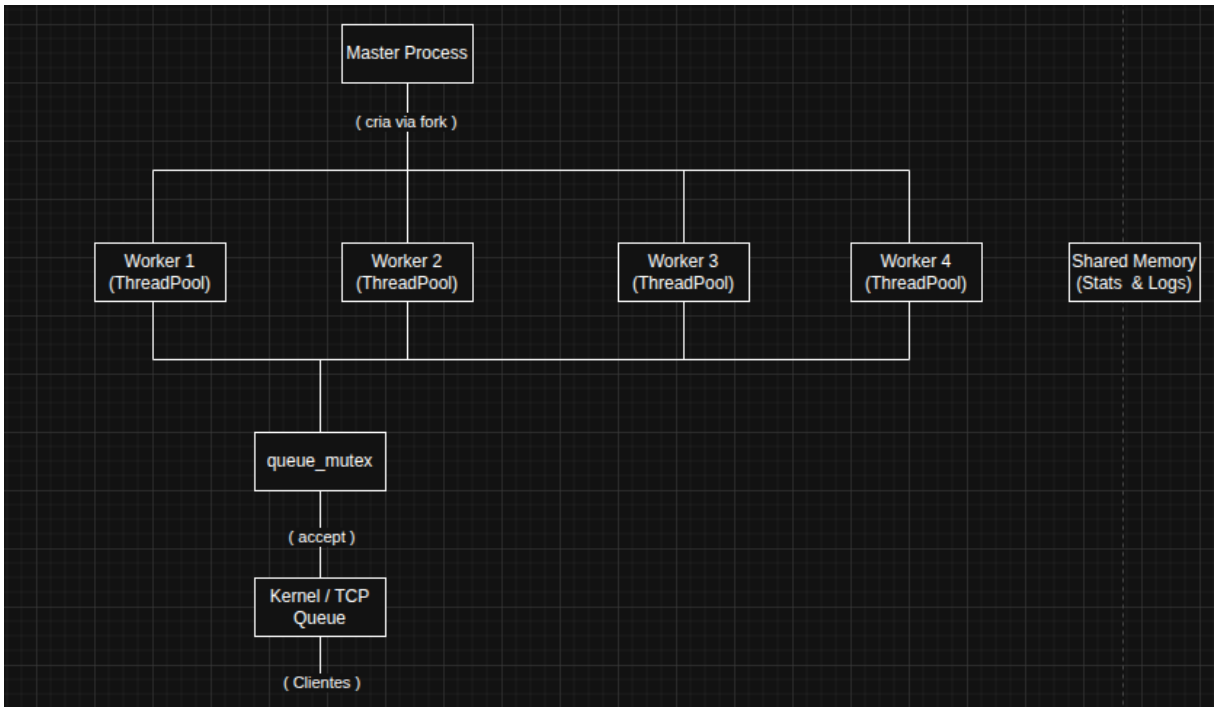

Figura 1: Arquitetura Master-Worker com Serialized Accept e Memória Partilhada.

2.2 Modelo "Serialized Accept"

Divergindo do modelo clássico de "Produtor-Consumidor" (onde o Master aceita e passa descritores), optou-se pelo modelo de **Serialized Accept**.

Nesta abordagem, todos os Workers herdam o socket de escuta. Para evitar o problema de *Thundering Herd* (onde todos os processos acordam desnecessariamente para uma única conexão), o acesso à chamada de sistema `accept()` é protegido por um semáforo. Apenas um Worker detém o lock num dado momento, delegando a gestão da fila de espera (*backlog*) inteiramente no Kernel do Linux, o que resulta numa maior eficiência e menor latência.

3 Detalhes de Implementação

A robustez do servidor assenta na gestão correta de concorrência dentro de cada processo Worker.

3.1 Thread Pool e Gestão de Tarefas

Cada Worker gere a sua própria *Thread Pool* de tamanho fixo. Esta estrutura evita o custo proibitivo de criar e destruir threads por cada pedido HTTP (`pthread_create`).

O funcionamento interno segue o padrão Produtor-Consumidor:

1. **Produtor (Main Thread):** Após o `accept()`, a thread principal insere o descritor do cliente na *Job Queue* (uma lista ligada protegida por Mutex).

2. **Sinalização:** É emitida uma notificação via Variável de Condição (`pthread_cond_signal`).
3. **Consumidor (Worker Thread):** Uma thread livre acorda, retira o descritor da fila e processa o pedido.

3.2 Cache LRU Thread-Safe

Para otimizar o tempo de resposta de ficheiros estáticos, implementou-se uma cache em memória com política de substituição LRU (*Least Recently Used*).

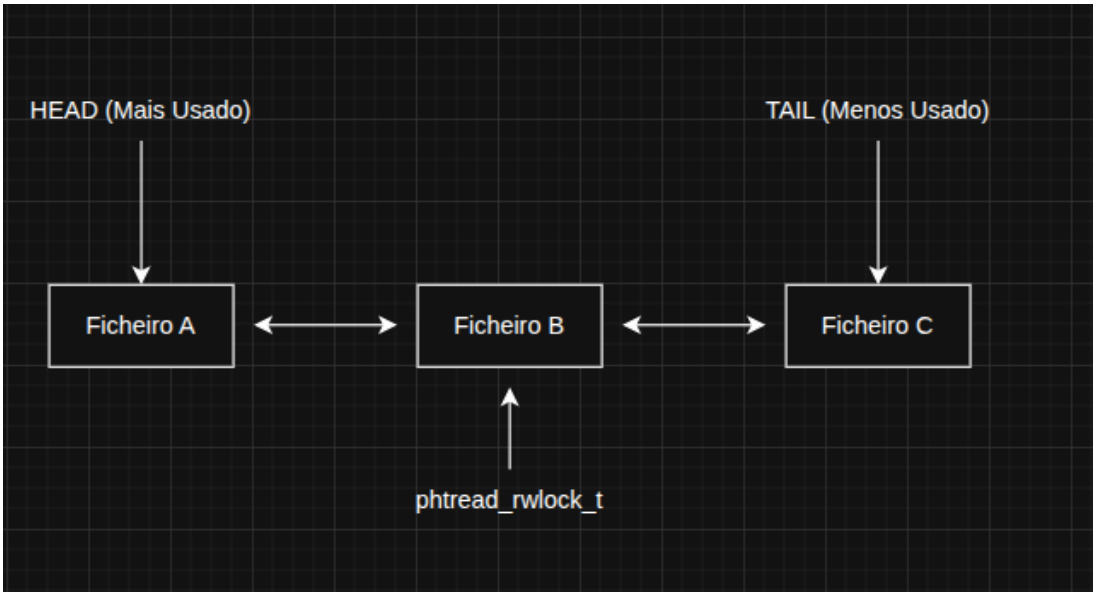

Figura 2: Estrutura interna da Cache LRU: Lista duplamente ligada com RWLock.

O desafio de concorrência foi resolvido utilizando **Reader-Writer Locks** (`pthread_rwlock_t`) em vez de Mutexes simples. Como ilustrado na Figura 2, isto permite que múltiplas threads leiam ficheiros populares simultaneamente (ex: `index.html`), bloqueando apenas quando é necessário inserir ou remover ficheiros da cache (operações de escrita).

3.3 Logging Atómico

O registo de acessos (`access.log`) é partilhado por todos os processos. Para evitar a corrupção de dados (linhas entrelaçadas), cada operação de escrita é protegida por um semáforo global (`log_mutex`), garantindo que cada linha de log é escrita de forma atómica.

4 Funcionalidades Avançadas

O servidor inclui funcionalidades extra que o aproximam de uma solução de produção.

4.1 Dashboard em Tempo Real

Foi implementado um endpoint dinâmico (`/stats`) que gera uma página HTML com o estado do servidor. Ao receber este pedido, o Worker lê a Memória Partilhada (protegida por semáforo) e apresenta métricas como *Uptime*, conexões ativas e total de bytes transferidos.

```
if (strcmp(req.path, "/stats") == 0) {
    sem_wait(sems->stats_mutex); // Leitura at mica
    // ... constr i HTML com stats ...
    sem_post(sems->stats_mutex);
    send_http_response(..., body, ...);
}
```

Listing 1: Geração do Dashboard em src/thread_pool.c

4.2 HTTP Keep-Alive

Para reduzir a latência em carregamentos de páginas complexas (com múltiplos recursos como CSS/JS), implementou-se o suporte a conexões persistentes. O servidor mantém o socket aberto após a resposta, aguardando novos pedidos do mesmo cliente até um limite de tempo (`TIMEOUT`), configurado através de `setsockopt(SO_RCVTIMEO)`.

4.3 Rotação de Logs

Para prevenir o consumo excessivo de disco, o sistema verifica o tamanho do ficheiro de log antes de cada escrita. Se exceder 10MB, o ficheiro é renomeado atomicamente para `access.log.1`, iniciando-se um novo ficheiro limpo.

5 Desafios de Engenharia

5.1 Sinais vs. Sincronização

Foi crucial distinguir o uso de sinais POSIX (`SIGINT`) para a gestão do ciclo de vida (shutdown gracioso) do uso de primitivas de sincronização (`pthread_cond`) para o fluxo de trabalho. A mistura destes conceitos levou inicialmente a *deadlocks*, resolvidos garantindo que os sinais apenas alteram flags atómicas globais.

5.2 Gestão de Memória

A prevenção de *memory leaks* num ambiente multithreaded é complexa. A utilização sistemática de ferramentas de análise dinâmica (Valgrind) permitiu identificar e corrigir fugas na destruição da cache e na limpeza das filas de tarefas durante o encerramento do servidor.

6 Conclusão

O projeto *ConcurrentHTTP* resultou num servidor web robusto, capaz de lidar com elevada concorrência graças à arquitetura de *Serialized Accept* e ao uso eficiente de *Thread Pools*.

A implementação das funcionalidades avançadas, como o Dashboard e Keep-Alive, adicionou valor prático significativo. Os testes realizados comprovaram a estabilidade do sistema e a ausência de fugas de memória, validando as opções de design tomadas ao longo do desenvolvimento.