Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Multi-Threaded Web Server

Documento de Design e Arquitetura

Sistemas Operativos – 2025/2026

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

Dezembro 2025

Conteúdo

1 Introdução **2**

2 Arquitetura do Sistema **2**
2.1 Processo Master 2
2.2 Processos Worker 3
2.3 Fluxo de Processamento de Pedidos 3

3 Decisões de Design (Justificação) **5**
3.1 Estratégia de Conexão: Serialized Accept 5

4 Estruturas de Dados e IPC **5**
4.1 Memória Partilhada (Estatísticas) 5
4.2 Cache LRU (Least Recently Used) 5

5 Funcionalidades Adicionais (Bónus) **6**
5.1 Dashboard de Estatísticas 6
5.2 Suporte a HTTP Keep-Alive 6
5.3 Virtual Hosts (VHosts) 7
5.4 Execução de CGI 7
5.5 Range Requests (HTTP 206) 7
5.6 Rotação Automática de Logs 7

6 Mecanismos de Sincronização **7**

7 Conclusão **7**

1 Introdução

O presente documento descreve as decisões de arquitetura e design tomadas durante o desenvolvimento do servidor web *ConcurrentHTTP*. O objetivo principal foi criar um sistema robusto, escalável e tolerante a falhas, capaz de processar múltiplos pedidos HTTP simultaneamente.

A solução combina multiprocessamento (para isolamento e robustez) com multithreading (para concorrência eficiente), utilizando primitivas de sincronização POSIX para garantir a integridade dos dados partilhados.

2 Arquitetura do Sistema

O sistema segue o modelo **Master-Worker**, complementado por uma arquitetura interna de **Thread Pool** em cada Worker.

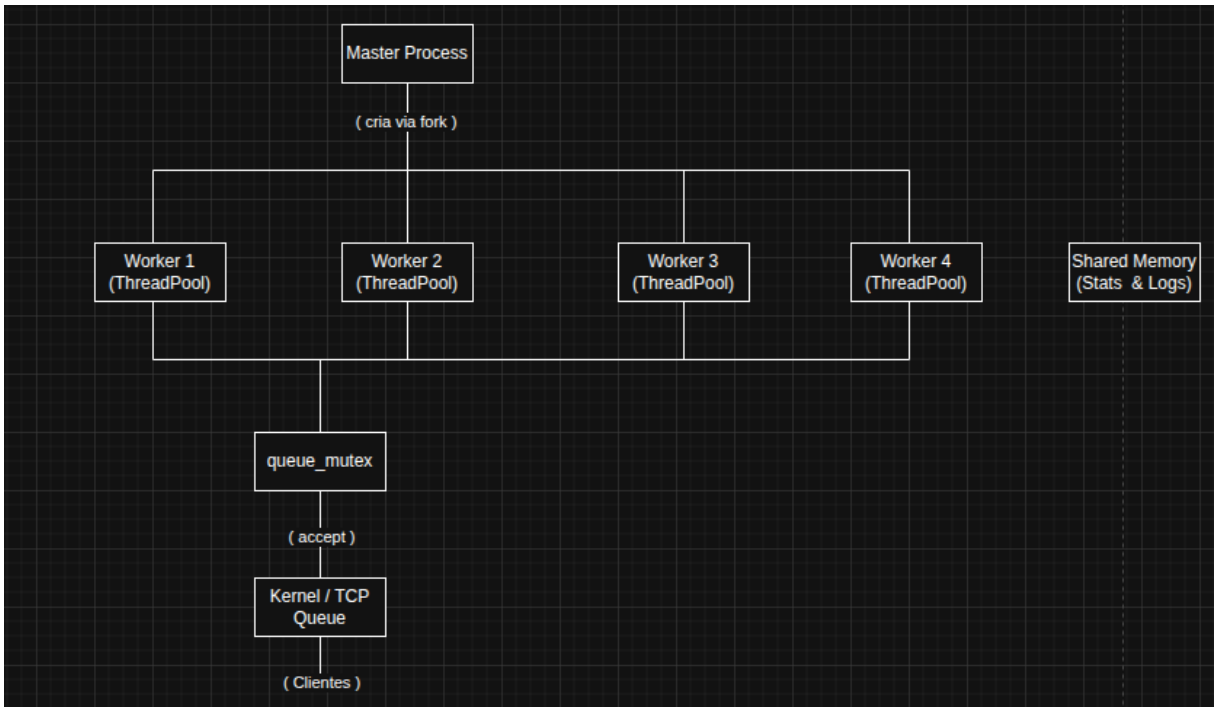

Figura 1: Arquitetura Master-Worker com Serialized Accept e Memória Partilhada.

2.1 Processo Master

O processo Master é responsável pela inicialização do sistema e gestão do ciclo de vida. As suas tarefas incluem:

- Criação e limpeza de recursos IPC (Memória Partilhada e Semáforos).
- Criação do *Listening Socket* na porta configurada.

- Criação dos processos Worker via `fork()`.
- Monitorização de sinais (`SIGINT`, `SIGTERM`) para encerramento gracioso.

2.2 Processos Worker

Cada Worker herda o descritor de ficheiro do socket de escuta. Internamente, cada Worker gere:

- Uma **Thread Pool** de tamanho fixo para processamento de pedidos.
- Uma **Cache LRU** local para ficheiros estáticos.
- A aceitação de novas conexões de forma sincronizada.

2.3 Fluxo de Processamento de Pedidos

A Figura 2 ilustra o ciclo de vida completo de um pedido HTTP dentro de um Worker, desde a aceitação da conexão até ao envio da resposta. Este fluxograma evidencia a lógica de decisão para as funcionalidades avançadas (Cache, CGI, Range Requests) e a gestão de conexões persistentes (Keep-Alive).

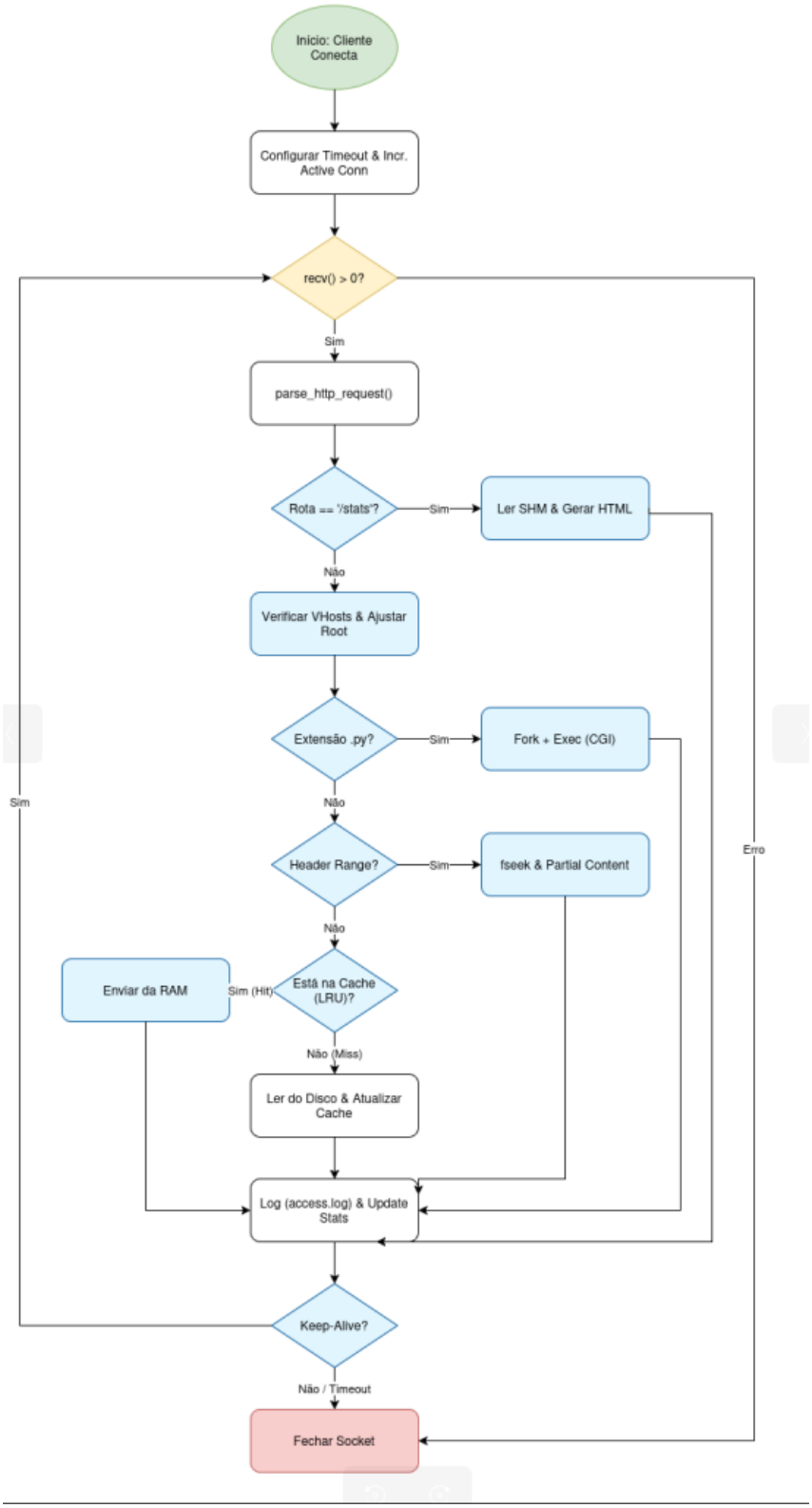

Figura 2: Fluxograma de processamento de um pedido HTTP na Thread Pool.

3 Decisões de Design (Justificação)

Esta secção detalha as escolhas técnicas mais importantes para a escalabilidade do sistema.

3.1 Estratégia de Conexão: Serialized Accept

Optou-se por desviar da sugestão de 'Master-Producer' do enunciado em prol da arquitetura 'Serialized Accept' para evitar o overhead de serialização de descritores de ficheiro e reduzir a latência.

Esta decisão baseou-se nos seguintes fatores técnicos:

1. **Eficiência:** Evita-se o overhead de copiar descritores entre processos via *Unix Domain Sockets*.
2. **Gestão de Fila:** Delega-se a gestão da fila de espera (*backlog*) ao Kernel do Linux, que é altamente otimizado.
3. **Prevenção de Thundering Herd:** Utiliza-se um semáforo (`queue_mutex`) antes do `accept()` para garantir que apenas um Worker acorda quando chega uma nova conexão.

4 Estruturas de Dados e IPC

As estruturas de dados foram desenhadas para minimizar o bloqueio e garantir a coerência.

4.1 Memória Partilhada (Estatísticas)

A estrutura `server_stats_t` armazena o estado global do servidor (pedidos totais, bytes, conexões ativas), acessível por todos os processos e protegida pelo semáforo `stats_mutex`.

4.2 Cache LRU (Least Recently Used)

A estrutura de dados subjacente combina uma lista duplamente ligada (para a ordenação temporal dos acessos) com um mecanismo de *lookup* linear para identificar os ficheiros em cache.

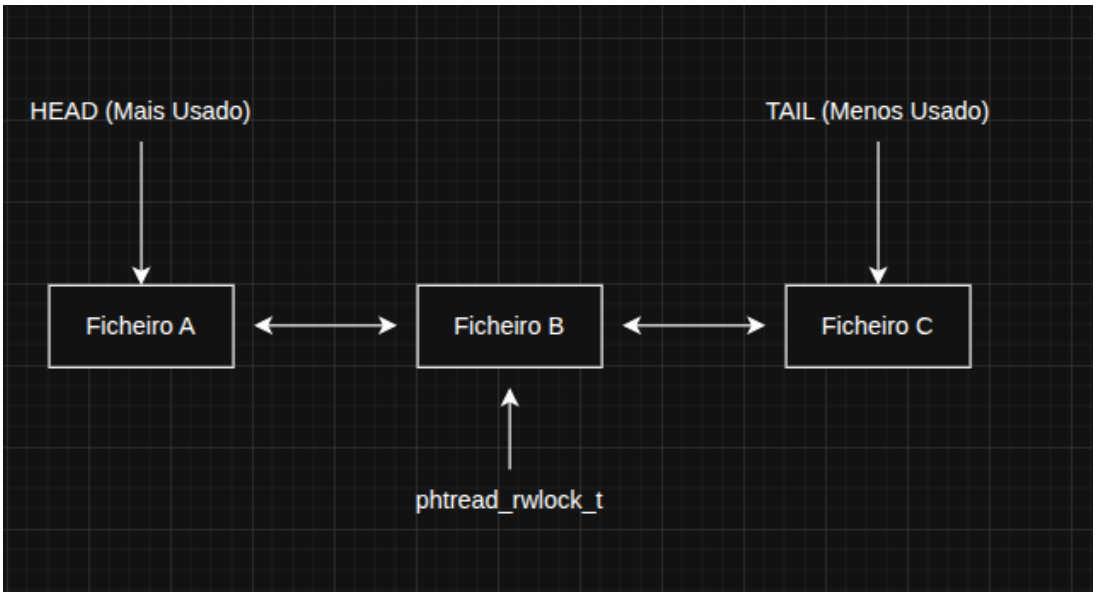

Figura 3: Estrutura interna da Cache LRU com lista duplamente ligada.

Estratégia de Sincronização: Utiliza-se um *Reader-Writer Lock* (`pthread_rwlock_t`) para proteger a estrutura. Embora este tipo de *lock* suporte leituras paralelas, a implementação rigorosa da política LRU exige que qualquer acesso a um ficheiro (mesmo de leitura) atualize a sua posição na lista duplamente ligada (movendo-o para a cabeça).

Por conseguinte, optou-se por adquirir o *lock* de escrita (`wrlock`) durante a operação de leitura (`cache_get`). Esta decisão de *design* prioriza a integridade estrutural da lista e a correção da política de substituição em detrimento do paralelismo total nas leituras, prevenindo corrupção de ponteiros em acessos concorrentes.

5 Funcionalidades Adicionais (Bónus)

Além dos requisitos base, o sistema integra 6 funcionalidades avançadas para maior robustez e versatilidade.

5.1 Dashboard de Estatísticas

Gera dinamicamente uma página HTML com métricas em tempo real (uptime, tráfego, cache hits), permitindo monitorização sem acesso à consola.

5.2 Suporte a HTTP Keep-Alive

Permite a reutilização de conexões TCP para múltiplos pedidos do mesmo cliente, reduzindo a latência do *handshake* TCP.

5.3 Virtual Hosts (VHosts)

Permite alojar múltiplos sites na mesma porta, servindo conteúdos diferentes com base no cabeçalho `Host` do pedido HTTP.

5.4 Execução de CGI

Suporta a execução de scripts dinâmicos (Python), criando um processo filho para executar o script e devolvendo o seu output ao cliente.

5.5 Range Requests (HTTP 206)

Implementa o suporte para downloads parciais de ficheiros (byte ranges), essencial para streaming de vídeo e retoma de downloads.

5.6 Rotação Automática de Logs

Gere automaticamente o tamanho do ficheiro de registo, rodando-o quando excede um limite (10MB) para evitar o enchimento do disco.

6 Mecanismos de Sincronização

O sistema utiliza uma abordagem híbrida de sincronização. É fundamental distinguir entre as primitivas **Inter-Processo (IPC)**, que residem em memória partilhada no sistema (`/dev/shm`) para coordenar os múltiplos processos Worker, e as primitivas **Intra-Processo**, que gerem a concorrência entre threads dentro do espaço de endereçamento de cada processo. O sistema utiliza uma combinação de primitivas POSIX para diferentes âmbitos:

Mecanismo	Nome	Função
Semáforo (IPC)	`queue_mutex`	Serializa o acesso ao `accept()` entre Workers.
Semáforo (IPC)	`stats_mutex`	Protege a escrita na Memória Partilhada.
Semáforo (IPC)	`log_mutex`	Garante escrita atómica no ficheiro de log.
Mutex (Thread)	`pool->mutex`	Protege a fila de tarefas da Thread Pool.
RWLock (Thread)	`cache->lock`	Garante exclusividade da lista LRU durante leituras.

Tabela 1: Resumo das Primitivas de Sincronização.

7 Conclusão

A arquitetura desenhada cumpre os requisitos de robustez e performance. A escolha pelo *Serialized Accept* simplificou a implementação sem comprometer a escalabilidade, enquanto o uso rigoroso de ferramentas de verificação como o *Valgrind* garantiu a ausência de fugas de memória.