Relatório
Servidor Web Multi-Thread

Membros:

Alexandre Regalado 124572

Cristiane Moreno 128076

Introdução

Este relatório documenta a implementação de um servidor HTTP/1.1 concorrente em C, utilizando arquitetura master-worker com IPC baseado em memória compartilhada.

O servidor processa requisições de forma paralela através de múltiplos workers, cada um com thread pool dedicada.

Funcionalidades pretendidas implementadas:

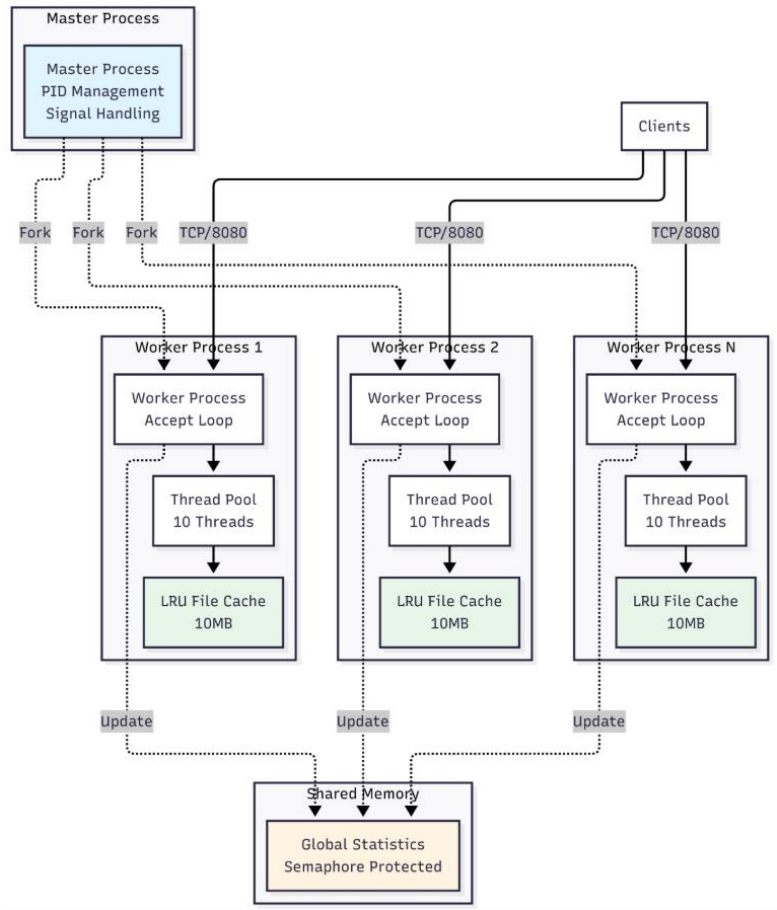
- Arquitetura Master-Worker
- Configuração por arquivo
- Fila partilhada Producer-Consumer
- Thread Pool por worker
- Cache LRU de arquivos
- Loggin thread-safe
- Estatísticas em memoria compartilhada

Funcionalidades adicionais implementadas:

- Endpoint '/health' para health checks
- Endpoint '/metrics' para estilo Prometheus
- Endpoint '/stats' com JSON detalhado
- Endpoints com prioridade
- Suporte Docker + Docker Compose
- Integração Prometheus + Grafana

Arquitetura Geral do Sistema

O servidor adota uma arquitetura *prefork master–worker*. No arranque, um processo master é responsável pela inicialização dos recursos globais e pela criação de um conjunto fixo de processos worker. Cada worker é independente e executa o processamento dos pedidos HTTP.



Componentes principais:

- Processo master, responsável pela coordenação do ciclo de vida do servidor;
- Processos worker, que aceitam conexões TCP e processam requisições;
- Pool de threads por worker, para paralelização do processamento;
- Fila partilhada para comunicação entre o accept loop e as threads;
- Memória partilhada para estatísticas globais

Modelo de Concorrência e IPC

Processos e Threads

O modelo de concorrência combina múltiplos processos com múltiplas threads por pro-cesso. Os workers são criados no arranque através da chamada *fork()*, evitando o custo de criação dinâmica de processos durante a execução.

Cada worker possui um pool fixo de threads, responsáveis por tratar as conexões aceites. Esta abordagem permite tirar partido de sistemas multi-core, distribuindo o trabalho de forma equilibrada.

Fila Producer-Consumer

A comunicação entre o *accept loop* e as threads é realizada através de uma fila circular limitada, implementando o padrão clássico producer-consumer. A sincronização é assegurada através de semáforos POSIX:

- Um semáforo para o número de slots vazios;
- Um semáforo para o número de slots ocupados;
- Um semáforo mutex para exclusão mútua.

Esta solução introduz um mecanismo natural de *backpressure*, prevenindo a aceitação ilimitada de conexões sob carga extrema.

Processamento de Requisições HTTP

O servidor suporta os métodos HTTP GET e HEAD. O processamento de uma requisição e segue os passos:

1. Aceita a conexão TCP
2. Lê e faz parsing da linha de pedido
3. Valida o método e o caminho solicitado
4. Verifica se é um endpoint especial. Caso seja processa-o.
5. Constrói o envio da resposta HTTP
6. Atualiza as estatísticas globais

Alguns endpoints, como */metrics /health /stats,* são tratados de forma prioritária, ignorando a fila. Isto garante o seu processamento. Esta regra foi criada para poder monitorizar o servidor quando este está sobrecarregado.

Gestão de Recursos e Sincronização

Cache de Ficheiros

Cada worker mantem uma cache local dos ficheiros, seguindo a regra LRU (Least Recentl Used). O acesso e protegido por read-write locks.

A implementação da cache demonstrou-se extremamente eficaz em testes onde os pedidos eram sempre os mesmos.

Estatísticas em Memoria Partilhada

As estatísticas globais do servidor, como o número total de pedidos e a contagem de cada código HTTP de resposta (20x, 4xx, 5xx), são armazenados em memoria partilhada criada com o **nmap()**. O acesso concorrente e gerido através de semáforos POSIX.

Funcionalidades Adicionais

Para alem das funcionalidades obrigatórias foram implementadas as seguintes funcionalidades adicionais:

- Endpoint **/health** para verificar se o servidor se encontra disponível usando o mínimo de recursos.
- Endpoint **/metrics** que nos dá as estatísticas globais em formato Prometheus. Útil para monitorizar o desempenho do servidor ao longo do tempo
- Endpoint **/stats** que nos dá as estatísticas globais, mas em formato JSON.
- Suporte para Docker e Docker Compose, útil para desenvolver em qualquer ambiente.

Estes extras ajudaram muito durante o desenvolvimento e na faze de testes, oferecendo uma visão global do sistema de forma fácil.

Metodologia de Testes

Testes Funcionais

Os testes funcionais foram realizados com comandos simples como o "**curl"** e browsers.

Testes de Carga

Embora o enunciado tenha recomendado o uso do Apache Bench, foi utilizada a ferramenta *k6* para testes de carga e stress, que nos permite definir para diferentes intervalos de tempo diferentes cargas.

Os pedidos vindos da ferramenta foram feitos em localhost, sendo que os teste não mostram problemas de rede nem latência. Foi testado puramente o servidor.

Resultados e Análise

Os resultados obtidos mostram que o servidor consegue acompanhar grandes quantidades de pedidos com um determinado número de workers e threads.

Sobre condições de stress, o servidor responde de forma adequada, devolvendo o código 503 sem ocorrência de falhas ou deadlocks.

A utilização da cache revelou-se o fator que mais melhorou o desempenho do servidor, aumentando significativamente o número de pedidos que conseguia responder e diminuir o tempo de resposta.

Com cache:

- Pedidos processados: 2.24 Milhões
- Tempo medio de resposta: 0 ms
- Pedidos processados com sucesso: 2,238,734
- Pedidos recusados: 4964

O tempo medio mostra-se **0** pois, não estamos a ler ficheiros (vem da cache) e não conta a latência de rede, apenas processamento do pedido.

Sem cache:

- Pedidos processados: 538 mil
- Tempo medio de resposta: 6 ms
- Pedidos processados com sucesso: 456,424
- Pedidos recusados: 81,912

Nota-se uma grande diferença entre o mesmo teste, com e sem o uso de cache. Tanto no número total de pedidos, quanto na percentagem de pedidos não processados.

Problemas Encontrados e Soluções

Durante o desenvolvimento do projeto foram identificados diversos problemas de concorrência como deadlocks, race conditions e memory leaks. Estes problemas foram resolvidos através da reorganização das regiões críticas e o uso de mecanismos de sincronização.

Conclusão

O desenvolvimento deste servidor HTTP permitiu aplicar os temas abordados nas aulas praticas da cadeira de Sistemas Operativos. Nomeadamente concorrência, memoria partilhada, semáforos, comunicação entre processos.

Nota

Todo o código fonte, scripts de teste e ficheiros de configuração encontram-se disponíveis no repositório entregue.

Os dashboards usados e configurações usadas na monotorização dos testes não se encontram no scope do projeto, por isso não foram entregues, mas encontram-se disponíveis em https://github.com/Alxit0/so-websocket-ipc

Foi usado inteligência artificial para:

- Discussão do projeto
- Geração de comentários e headers de funções
- Analise de código fonte e alerta de possíveis erros
- Geração de alguns ficheiros markdown de documentação