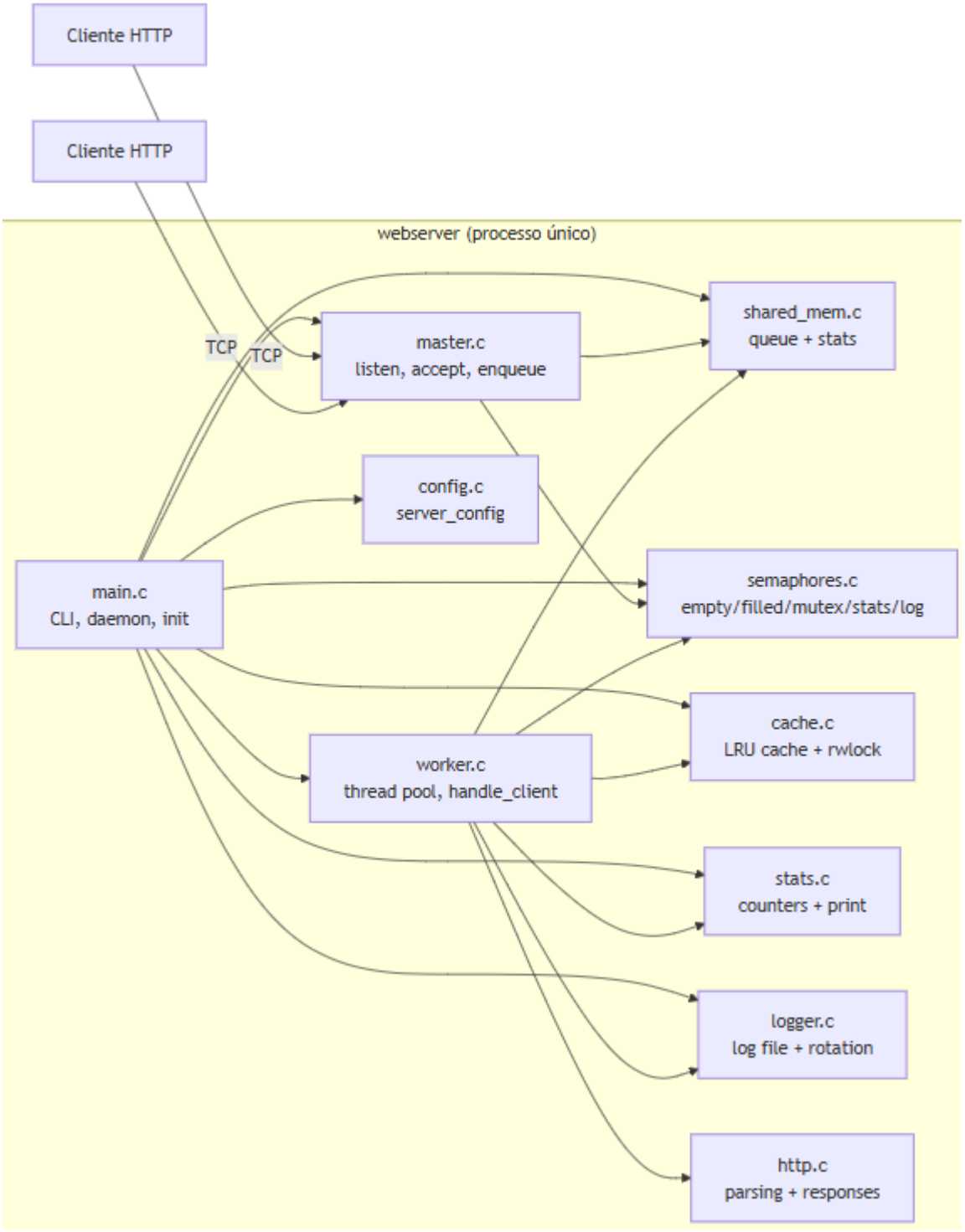
Visão Geral da Arquitetura

A arquitetura do ConcurrentHTTP Server (webserver) é baseada num único processo principal que gerencia um pool fixo de threads de worker, através d'um design de produtor-consumidor para lidar com as conexões. O processo principal é iniciado pela main.c, que configura o ambiente (leitura do server.conf, inicialização de memória partilhada, semáforos, cache, logger e estatísticas) e inicia o master.c. O módulo master.c é responsável pela gestão do socket de escuta e atua como o produtor, enfileirando as conexões recebidas numa fila circular de sockets (bounded queue), trata as recusas com respostas 503 quando a fila está cheia. Os threads worker.c funcionam como consumidores: cada um retira ligações da fila, processa o ciclo de pedido/resposta HTTP completo (http.c para parsing, cache.c para LRU, stats.c para métricas e logger.c para registo), com suporte a funcionalidades avançadas como Keep-Alive e Range Requests (Features Bonus). A concorrência e a segurança de thread/processo são garantidas pelos módulos shared_mem.c (para a fila e estatísticas globais) e semaphores.c, que fornece os mecanismos de sincronização (empty_slots, filled_slots, queue_mutex, stats_mutex, log_mutex). Componentes auxiliares como config.c gerem as definições e tests/ são usados para validação de concorrência e carga.

Diagrama de arquitetura



Modelo de Concorrência e Sincronização

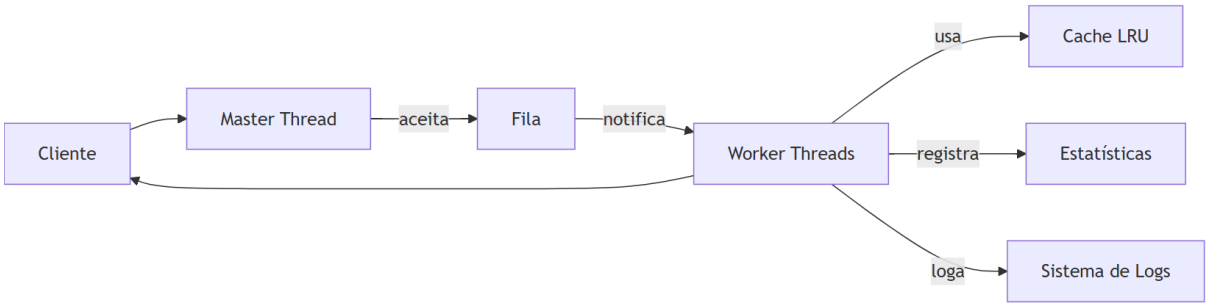
O ConcurrentHTTP Server emprega um modelo robusto de concorrência baseado no padrão Produtor-Consumidor para a gestão de conexões e utiliza primitivas de sincronização POSIX para garantir a integridade dos dados partilhados. A Feature 1, a Fila de Conexões, é uma fila circular bounded localizada em memória partilhada (shared_data_t), definida pelos campos sockets[], front, rear, count e capacity. A sincronização é obtida através de dois semáforos POSIX, empty_slots e filled_slots, que controlam, respetivamente, a capacidade livre e o trabalho pendente, e um queue_mutex para exclusão mútua durante as manipulações dos índices da fila. O Produtor (master) tenta adquirir espaço com sem_trywait(empty_slots); em caso de falha (EAGAIN), o servidor responde imediatamente com 503 Service Unavailable e descarta a conexão, garantindo a resiliência contra sobrecarga. O Consumidor (worker) bloqueia em sem_wait(filled_slots) e, após processar a conexão via handle_client_connection, liberta o slot com sem_post(empty_slots).

A Feature 2, o Thread Pool, é um conjunto fixo de threads de worker criadas em main.c. Cada thread executa um ciclo de dequeue_connection seguido por handle_client_connection para processar um ou mais pedidos HTTP (incluindo o suporte a Keep-Alive). O shutdown ordeiro é acionado pelo SIGINT, que define a flag keep_running = 0, seguido pelo master que desbloqueia os workers com posts adicionais em filled_slots, permitindo que main.c utilize pthread_join para a terminação limpa de todas as threads.

A Sincronização dos Subsistemas Auxiliares é crucial: as Estatísticas (stats.c), residentes em shared_data_t, são protegidas por um stats_mutex, garantindo atualizações atómicas de métricas como total_requests, códigos de status, bytes_transferred e tempos de resposta, além da contabilização de cache hits/lookups através de stats_cache_access. O Cache LRU (cache.c) é um cache por processo, protegido por um reader-writer lock (pthread_rwlock_t), permitindo alta concorrência em cache hits (modo leitor) e serialização apenas durante modificações estruturais (inserção, evicção LRU) no modo escritor. Por fim, o Logger (logger.c) garante segurança de acesso ao ficheiro por meio de um log_mutex, utiliza um buffer interno para otimizar o I/O de disco e implementa log rotation (a cada 10MB) para gestão eficiente do espaço em disco.

Ciclo de Vida de um Pedido HTTP

Fluxo geral



Handle_client_connection

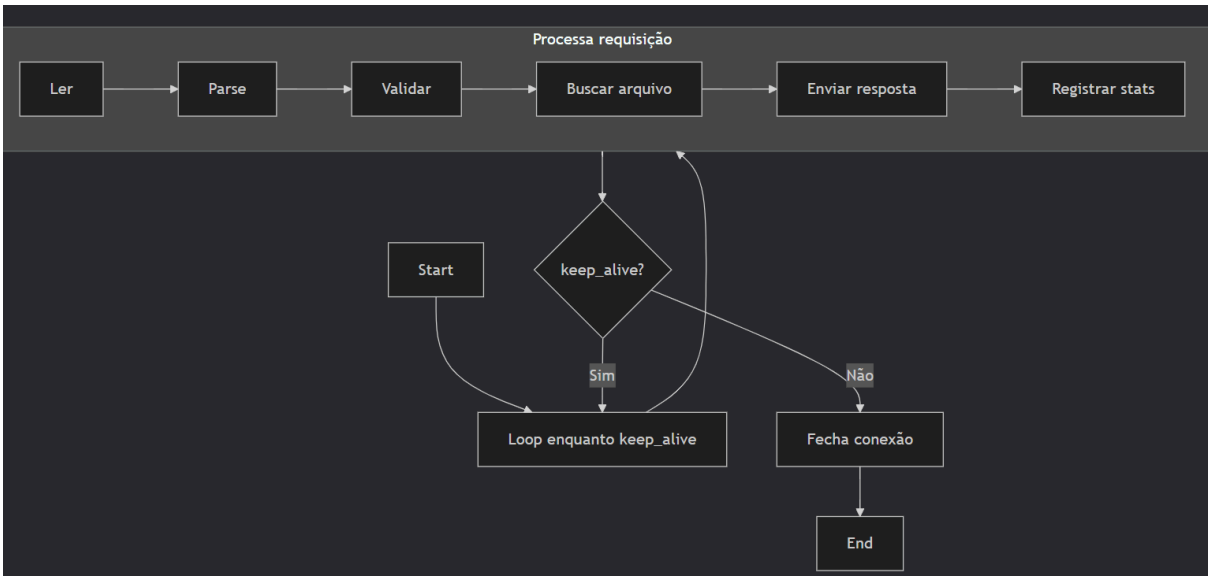
A função handle_client_connection implementa o ciclo de processamento de pedidos para cada client_fd, suportando o mecanismo HTTP Keep-Alive através de um loop interno. Inicialmente, um timeout (SO_RCVTIMEO) é configurado no socket com base em TIMEOUT_SECONDS, e a métrica active_connections é incrementada via stats_request_start. O loop interno inicia com a leitura do pedido utilizando recv_http_request. Em caso de timeout ou EOF (retorno ≤ 0), o ciclo termina, e a conexão é fechada.

Em seguida, o pedido é analisado por parse_http_request, extraindo o método, o caminho e cabeçalhos relevantes, como Connection e Range. A decisão de Keep-Alive é tomada com base nesses cabeçalhos e na versão HTTP (HTTP/1.1 mantém por omissão, HTTP/1.0 fecha por omissão). Apenas o método GET (ou potencialmente HEAD) é suportado; outros métodos resultam numa resposta 405 Method Not Allowed.

O caminho do URI é normalizado (prevenindo sequências .. e mapeando / para /index.html) para construir o full_path sob o DOCUMENT_ROOT. O subsistema de Cache LRU é acedido através de cache_get_file, e o resultado (hit ou miss) é registado por stats_cache_access. Se o recurso não for encontrado, é retornada uma resposta 404 Not Found. O servidor suporta Range Requests: se um cabeçalho Range: bytes=start-end válido estiver presente, a resposta 206 Partial Content é preparada, com o header Content-Range apropriado e o envio exclusivo do segmento de dados solicitado; intervalos inválidos resultam em 416 Range Not Satisfiable.

O resultado final é enviado através de send_http_response, que encapsula o status code (200, 206, 404, etc.), headers relevantes (Content-Type, Connection) e os dados do ficheiro. Após o envio, o tempo de resposta é calculado, e stats_request_end é invocado para atualizar todas as métricas globais, incluindo total_requests, bytes transferidos e a média de tempos de resposta, além de decrementar active_connections. Finalmente, logger_log_request serializa o evento no log no formato Apache Combined. O loop é repetido apenas se o Keep-Alive for verdadeiro e não tiver ocorrido um erro fatal.

Fluxograma interno do handle_client_connection



Rationale

O design do ConcurrentHTTP Server baseia-se em decisões estratégicas que otimizam a concorrência e a eficiência de recursos. A Fila de Conexões (Feature 1) adota rigorosamente o padrão Bounded Buffer utilizando Semáforos POSIX (empty_slots, filled_slots, queue_mutex), que fornecem um modelo explícito de capacidade e simplificam o shutdown ordenado através da sinalização de workers bloqueados. Em vez de um modelo thread-per-connection, a arquitetura emprega um Pool Fixo de Threads de Worker (Feature 2), minimizando a sobrecarga de criação/destruição de threads e limitando a concorrência real para garantir um desempenho previsível sob alta carga.

A monitorização é centralizada através de Estatísticas em Memória Partilhada (shared_data_t), protegidas por stats_mutex, que permite a atualização consistente de métricas globais (e.g., Total Requests, Average Response Time, Cache Hit Rate) por todas as threads, enquanto o processo master as imprime periodicamente (stats_print). A Cache LRU é implementada por processo e usa um Reader-Writer Lock (pthread_rwlock_t), permitindo que múltiplos workers leiam dados em cache em paralelo, serializando apenas operações de modificação (inserção, evicção). A política do cache restringe a entrada de ficheiros a $< 1 \text{MB}$ para evitar a poluição do limite de memória (e.g., 10MB) por recursos grandes.

O Logger (logger.c) garante a atomicidade do registo através do log_mutex e otimiza o I/O ao agrupar entradas num buffer interno antes de escrever em disco. A inclusão da rotação de log (a 10MB) assegura a gestão sustentável do espaço em disco. Em termos de protocolo, o suporte a HTTP Keep-Alive reduz significativamente o overhead de estabelecimento de conexões TCP, melhorando o throughput. Por fim, as Range Requests (206) permitem o suporte a funcionalidades avançadas, como downloads parciais e

streaming, enviando apenas o segmento de dados solicitado do buffer do ficheiro (ou cache) e garantindo a resposta 416 para intervalos inválidos.