ConcurrentHTTP Server – Technical Report

1. Organização do Código e Build

1.1 Estrutura de Ficheiros

Módulo	Descrição
`src/main.c`	Ponto de entrada, inicialização e shutdown
`src/master.c`	Socket listener e enqueue de conexões
`src/worker.c`	Thread pool e processamento de clientes
`src/cache.c/.h`	Cache LRU thread-safe
`src/logger.c/.h`	Logging com rotação e sincronização
Makefile	Build e targets auxiliares

1.2 Fluxo Principal

Inicialização (`main.c`):

- Parse de argumentos: `-c`, `-p`, `-w`, `-t`, `-d`, `-v`, `-h`, `--version`
- Carregamento de config com `load_config`
- Daemonização opcional (`-d`)
- Inicialização de: shared memory, semáforos, cache, logger, stats
- Criação de thread pool
- Loop de `accept()` + `enqueue_connection`

Shutdown (SIGINT):

- Define `keep_running = 0`
- Acorda workers com `sem_post`
- `pthread_join` de todas as threads
- Destruição de recursos (cache, logger, semáforos, shared memory)

---

2. Implementação por Feature

2.1 Connection Queue (Bounded Producer–Consumer)

Estrutura (`shared_mem.h`):

```
shared_data_t {
    queue {
        int sockets[MAX_QUEUE_SIZE];
        int front, rear, count, capacity;
```

```
    }
}
```

Semáforos:

- `empty_slots` – espaços vazios disponíveis
- `filled_slots` – posições com socket
- `queue_mutex` – exclusão mútua

Produtor (`enqueue_connection`):

1. `sem_trywait(empty_slots)` → se `EAGAIN`, responde 503 e fecha
2. `sem_wait(queue_mutex)`
3. Coloca socket em `sockets[rear]`
4. Atualiza `rear` e `count`
5. `sem_post(queue_mutex)` → `sem_post(filled_slots)`

Consumidor (`dequeue_connection`):

1. `sem_wait(filled_slots)`
2. `sem_wait(queue_mutex)`
3. Lê de `sockets[front]`
4. Atualiza `front` e `count`
5. `sem_post(queue_mutex)` → `sem_post(empty_slots)`

2.2 Thread Pool Management

Configuração:

- Nº threads = `num_workers × threads_per_worker`
- Overrides via `-w` (workers) e `-t` (threads/worker)

Ciclo de Worker:

```
while (keep_running) {
    client_fd = dequeue_connection(...);
    if (client_fd >= 0) {
        handle_client_connection(...);
    }
}
```

Encerramento:

- `keep_running = 0` acorda todas as threads
- `pthread_join` aguarda finalização

2.3 Shared Statistics

Contadores Protegidos (`stats_mutex`):

- `total_requests`, `bytes_transferred`
- `successful_2xx`, `client_errors_4xx`, `server_errors_5xx`
- `cache_hits`, `cache_lookups`
- `active_connections`, `total_response_time`
- `start_time` (para uptime)

Operações:

- `stats_request_start()` – incrementa `active_connections`
- `stats_request_end()` – atualiza contadores e tempo médio
- `stats_cache_access()` – regista lookups e hits
- `stats_print()` – exibe resumo com uptime, taxa cache, erros

2.4 Thread-Safe File Cache (LRU)

Estrutura:

- Lista duplamente ligada com `head` (MRU) e `tail` (LRU)
- Entradas: `path`, `data`, `size`

Sincronização:

- `pthread_rwlock_t` global
- `rdlock` para buscas, `wrlock` para inserções/evicções

Política:

- Apenas ficheiros `< 1MB` entram no cache
- Limite total em bytes via `CACHE_SIZE_MB`
- Evicção do `tail` quando necessário

Integração Range:

- Cache armazena ficheiro completo
- Pedidos Range usam janela do buffer
- Sem estados específicos por range

2.5 Thread-Safe Logging

Características:

- Sincronização com `log_mutex` (semáforo POSIX)
- Macros `SEM_WAIT_SAFE`/`SEM_WAIT_SAFE_RET` para tratar `EINTR`
- Formato Apache-like: `ip - - [timestamp] "METHOD PATH HTTP/x.y" status bytes`

Rotação:

- Acionada quando ficheiro > 10MB

- Renomeia log atual para `.1` e reabre novo

Buffer:

- Flush automático quando cheio, a meia capacidade, ou antes de rotação

---

2.6 HTTP Keep-Alive

Implementação:

- `SO_RCVTIMEO` configurado com `TIMEOUT_SECONDS`
- Loop por conexão em `handle_client_connection`

Política:

- `Connection: close` → fecha imediatamente
- `Connection: keep-alive` → mantém aberta
- Ausência: HTTP/1.0 fecha; HTTP/1.1 mantém (por omissão)

2.7 Range Requests (HTTP 206)

Validação e Resposta:

- Header `Range` válido → calcula `start/end` vs `file_size`
- Intervalo válido → HTTP 206 + header `Content-Range` + corpo parcial
- Inválido/out-of-range → HTTP 416

Com Cache:

- Buffer completo armazenado
- Apenas fatia solicitada enviada ao cliente

---

3. Limitações e Trabalho Futuro

Limitação	Impacto
Modelo single-process + threads	Não escalável a múltiplos cores com partilha eficiente
Cache por processo	Sem partilha global em multi-processo
Range requests (um intervalo)	Múltiplos ranges não suportados
Parâmetros não otimizados	Performance dependente de hardware

Possíveis Melhorias:

- Migração para multi-processo com IPC distribuído
- Cache distribuído entre processos
- Suporte a múltiplos ranges (RFC 7233)
- Tuning automático baseado em hardware

---

4. Conclusões

Requisitos Cumpridos:

- Fila bounded com respostas 503 em overload
- Thread pool de tamanho fixo
- Estatísticas globais thread-safe
- Cache LRU com proteção
- Logging com rotação segura
- Keep-Alive e Range requests

Validação:

- Testes com curl, Apache Bench, concorrência
- Valgrind/Helgrind sem problemas críticos
- Estabilidade sob carga comprovada