Sistemas Operativos
Trabalho Prático
2024/2025

Resumo

Foi proposto o desenvolvimento de um serviço de indexação e pesquisa sobre documentos de texto, constituído por duas partes: o servidor, responsável pela parte lógica, como a gestão da memória volátil e em disco, as estruturas usadas, e as funcionalidades fornecidas; e o cliente, que irá submeter os pedidos para o servidor. É explicada a arquitetura do software desenvolvido, não apenas a organização modular entre cliente e servidor, mas também a gestão de processos usada.

Grupo 25

Andreia Alves Cardoso	A106915
Cátia Alexandra Ribeiro da Eira	A107382
Filipa Cosquete Santos	A106837

Arquitetura modular

O programa foi desenvolvido com uma **divisão modular** entre o sistema e o cliente, estruturados em diretorias diferentes. Cada componente é isolado do outro, partilhando apenas informações relevantes, como definições comuns no módulo *protocol*, para permitir a comunicação entre si.

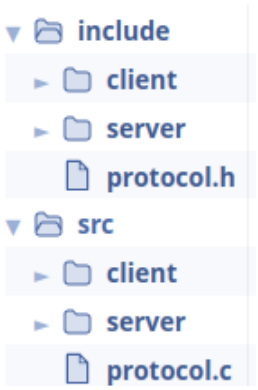
O **cliente** é o único responsável pela interação com o utilizador. Este recebe e encapsula os comandos do utilizador em duas estruturas guardadas em *protocol*, primeiro em *ClientRequest* e depois numa *Message*, que envia para o sistema. O cliente não tem nenhum conhecimento sobre a lógica interna ou que comandos o servidor suporta; apenas conhece a estrutura que precisa para formatar os comandos. Este design assegura que alterações na lógica do servidor não afetam o funcionamento do cliente, desde que o formato de comunicação se mantenha consistente.

O **servidor** nunca interage diretamente com o utilizador. Recebe os comandos já estruturados e processa-os, se forem válidos. Internamente, o servidor também segue uma abordagem modular: o ficheiro principal gere o ciclo de vida e controlo do servidor, enquanto outros módulos tratam da gestão da memória e do processamento dos pedidos dos clientes. Esta separação torna o código mais claro, mais fácil de testar, e simplifica a adição de novas funcionalidades. No entanto, é relevante notar que estes módulos internos não estão totalmente encapsulados entre si, uma vez que há uma certa quantidade de informação partilhada.

A **arquitetura modular** adotada foi fundamental para a robustez do software desenvolvido. Não só garante melhor organização e legibilidade, como minimiza erros que ocorrem devido à quebra do encapsulamento. O desenvolvimento do programa é então facilitado, como quando foi notada a necessidade de haver a estrutura adicional *Message*: a sua inclusão em *protocol* exigiu apenas a criação da estrutura e funções auxiliares, sem necessidade de grandes alterações em outras partes do código.

Esta abordagem modular foi escolhida desde o início do projeto para facilitar o desenvolvimento e manutenção, testes independentes e a possível extensão futura do sistema.

- Diretorias principais:



- Diretoria do servidor:

Então, para resumir, temos:

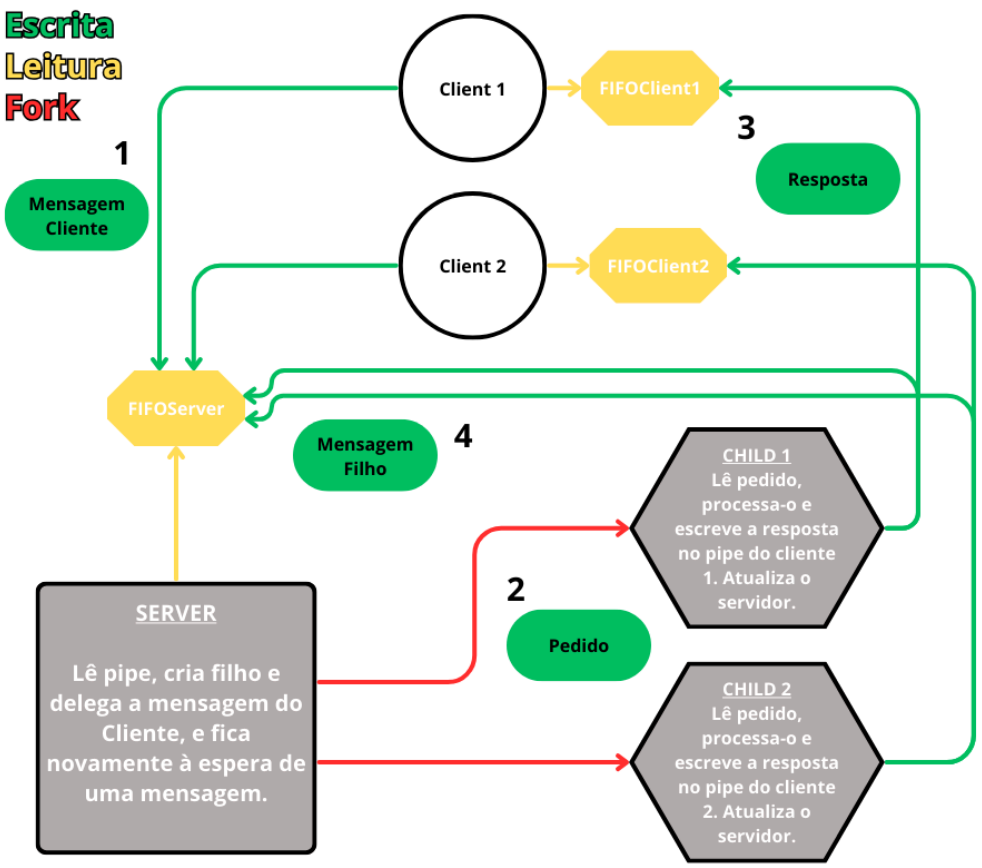
- **Cliente**: responsável pelo I/O do utilizador
- **Servidor**: responsável pela gestão de memória e processamento dos pedidos.
- **Protocol**: responsável pela conexão entre ambos.

Arquitetura multi-processo

O servidor foi concebido com uma arquitetura multi-processo, permitindo-lhe atender **múltiplos clientes simultaneamente**. No arranque, o servidor cria um *pipe* nomeado, designado por "fifoServer", através do qual irá ler todos os pedidos dos clientes. Cada cliente, por sua vez, possui o seu próprio *pipe* nomeado, nomeado "fifoClient_", seguido com o número *pid* do respetivo processo (de modo a ser um ID obrigatoriamente único!), onde irá receber a resposta do servidor.

O cliente escreve o seu pedido no *pipe* do servidor e espera pela resposta através do próprio *pipe*. Este modelo segue o algoritmo FCFS (*First Come First Serve*), e assim que o servidor recebe um pedido do cliente, imediatamente cria um novo processo filho através de um *fork* e delega-lhe o trabalho e responsabilidade de ler e processar o pedido. Este método garante que o pai permaneça **disponível** para ler pedidos sem bloqueios.

Após concluir o processamento do pedido, o filho escreve a resposta para o *pipe* do cliente correspondente e termina a sua execução. Nos casos em que o pedido do cliente implica alterações na memória do programa (inserção e remoção de documentos indexados ou atualização da cache) o processo filho envia uma mensagem para o *pipe* do servidor, a pedir que **atualize** a memória e os dados necessários. No fim do processo do filho, este finalmente envia uma mensagem a informar o servidor de que está pronto para ser colhido (*wait*), para evitar a acumulação de processos *zombie* e libertar os recursos do sistema associados.



Podemos então notar que seguindo este modelo, o servidor consegue escalar eficientemente e mantém-se responsivo mesmo com um elevado número de pedidos em simultâneo, uma vez que como delega a execução intensiva aos seus processos filhos, continua disponível para aceitar novos pedidos.

Mecanismos de Comunicação

O servidor recebe todas as comunicações no seu *pipe* através de estruturas do tipo ***Message***. Esta estrutura genérica encapsula tanto pedidos de clientes como comandos de processos filhos, permitindo que o servidor central aceite ambos de forma uniforme. Ela possui um campo para o remetente, que identifica quem enviou a mensagem (filho ou cliente) e um campo para os dados, que possui a informação específica. Todas as mensagens têm tamanho **fixo**, o que simplifica a leitura e escrita nos pipes nomeados e evita erros relacionados com os tamanhos das variáveis, ao custo de um maior desperdício de memória.

Uma mensagem de um **cliente** inclui:

- O caminho do seu próprio pipe nomeado (para onde a resposta será enviada).
- Os comandos introduzidos pelo utilizador (máximo de 5).
- O número real de comandos recebidos.

Uma mensagem de um **processo filho** inclui:

- Um identificador do tipo de comando (ADD, DELETE, LOOKUP, etc.).
- Uma estrutura *Document*.

A estrutura do filho indica diretamente o tipo de pedido, uma vez que este não precisa do encapsulamento necessário entre o cliente e o servidor, bem como um Documento, visto que geralmente todos os pedidos envolvem um.

- **Estrutura genérica de Mensagem:**

Message
- type : MessageType
- data (union)
  - **ClientRequest**
    - fifoPath : char[256]
    - command :char[5][64]
    - noCommand : int
  - **ChildRequest**
    - cmd : ChildCommand
    - doc : Document

- **Exemplo de mensagem recebida do cliente:**

Message
- type : CLIENT
- data
  - **ClientRequest**
    - fifoPath : “tmp/fifoClient2444”
    - command : [“-l”; “174”; “law”]
    - noCommand : 3

- **Exemplo de mensagem recebida do filho:**

Message
- type : CHILD
- data
  - **ChildRequest**
    - cmd : Add
    - doc : Document
      - Id: 174
      - Title: The Raven
      - Authors: Edgar Allan Poe
      - Path: Gdataset/1065.txt
      - Year: 2010

Inicialmente, a arquitetura do sistema previa que apenas o cliente se iria comunicar com o servidor através do seu *pipe*, enquanto os filhos usariam pipes anónimos com o processo pai. No entanto, esta abordagem implicava que o processo pai ficasse **bloqueado** à espera da resposta dos filhos, o que compromete o objetivo de atender vários clientes ao mesmo tempo. Assim surgiu a necessidade de criar a estrutura *Message*, que permitiu tratar todas as comunicações da mesma forma, mantendo o processo pai sempre **disponível**. Como mencionado antes, devido à arquitetura modular já existente, a integração desta estrutura foi simples.

Antes de enviar a resposta ao cliente, o processo filho envia primeiro o **tamanho** exato da mensagem que será enviada a seguir. Desta forma, o cliente não está limitado no tamanho da sua resposta e pode alocar **dinamicamente** apenas a quantidade necessária de memória, evitando tanto o corte prematuro da resposta como o desperdício de recursos ao reservar espaço excessivo, garantindo uma eficiente gestão de memória. Esta abordagem foi adotada para lidar com a variabilidade no tamanho das respostas, especialmente em comandos de pesquisa, onde o número de documentos encontrados pode variar drasticamente: desde apenas alguns resultados, até uma lista próxima do número total de documentos indexados. O programa certifica-se que a resposta enviada, no entanto, não é maior que a **capacidade mínima** possível de escrita num *pipe* nomeado (512 bytes), de modo a suportar todos os tipos de OS.

Funcionalidades e Decisões

O servidor organiza os documentos indexados com estruturas de acesso eficientes e gestão de dados. A cache assegura rapidez de operação, enquanto o armazenamento persistente garante a manutenção dos dados entre sessões. As operações disponíveis e as estruturas são descritas a seguir.

Começando com a estrutura ***Document***, já referida várias vezes neste relatório, é o elemento que representa um ficheiro indexado pelo sistema. Definida no módulo *protocol*, esta estrutura contém todos os metadados necessários de um documento, com um formato de tamanho fixo de modo a simplificar as operações de leitura e escrita, tanto em memória como em disco. Se os dados fornecidos excederem o limite, o excesso é truncado silenciosamente.

Cada *Document* tem os seguintes campos:

Public Attributes

int	**id**
char	**title** [200]
char	**authors** [200]
char	**path** [64]
short	**year**

Cache e Persistência

Para garantir rapidez de acesso e durabilidade dos dados, o sistema combina memória volátil (cache) com persistência permanente em disco. Isto permite otimizar a resposta a pedidos frequentes, enquanto assegura que a informação não se perde entre sessões de uso do programa. No início do programa o ficheiro com os dados é lido de modo a repopular a cache (os primeiros documentos indexados).

Estruturas usadas para a gestão da memória

A cache de documentos é gerida através de duas estruturas principais, fornecidas pela biblioteca **GLib**:

- *GHashTable*: utilizada para mapear os identificadores de documentos (*id*) para as respetivas instâncias de *Document*. Esta estrutura foi escolhida por permitir operações de inserção, remoção e pesquisa em tempo constante médio de O(1).
- *GQueue*: mantém a ordem de utilização dos documentos na cache. Sempre que um documento é acedido ou inserido, a sua posição na fila é atualizada, permitindo implementar de forma eficiente a política de substituição **LRU** (*Least Recently Used*).

Para a gestão da **persistência** de dados, foi criada uma *GHashTable* referida como *IndexSet* ou apenas *Index*, que guarda os documentos que já foram escritos no disco. Esta estrutura apenas guarda os respetivos IDs, daí funcionar como um *Set*.

Funcionamento geral da cache

Quando um documento é adicionado, é inserido tanto na tabela cache, como em último lugar na fila de utilização. Se a cache atingir o seu limite máximo de capacidade, o documento utilizado menos recentemente (na cabeça da fila) é expulso da cache e escrito para o disco, sendo também inserido no *Index*. Quando um documento é removido, no entanto, é retirado da cache e/ou do *Index*, mas a remoção no disco só ocorre no final da execução do servidor, para evitar espaços em branco no ficheiro ou operações de reescrita pesadas durante a execução do programa. Quando um documento é pesquisado, o programa começa por procurá-lo na cache e, se não o encontrar, ao ler o documento do disco, adiciona-o à cache como um recém-usado.

Esta abordagem minimiza o custo computacional durante o funcionamento normal do sistema, adiando operações pesadas de escrita para o momento em que o impacto no desempenho é menos relevante.

Funcionalidades Específicas

Estas funcionalidades são aquelas referidas no enunciado do projeto. Encontram-se todas definidas no módulo *services* e devolvem mensagens (*strings*) já estruturadas, sendo apenas necessário enviá-las para o cliente em seguida.

Adicionar ficheiros

O processo de indexação de um novo documento começa com a atribuição de um identificador único. Inicialmente, considerou-se utilizar apenas o número *pid* do processo filho responsável pelo pedido, assumindo que este seria sempre único. No entanto, com a adição da persistência em disco, verificou-se que, após um reinício do sistema operativo, os *pid* poderiam ser reutilizados, potencialmente originando colisões de identificadores. Assim, optou-se por utilizar um identificador sequencial, tendo o cuidado de se manter correto entre usos do programa, garantidamente único e com maior legibilidade.

O programa não verifica se o caminho do novo ficheiro já existe no sistema quando tenta indexar um novo ficheiro, sendo possível adicionar um mesmo ficheiro físico com metadados diferentes. Após a adição, o novo documento é adicionado à cache como um elemento recentemente utilizado.

Consultar ficheiros

Quando o servidor recebe um pedido de consulta a um documento, segue a seguinte sequência:

- Procura primeiro na cache.
- Se não encontrar o documento, procura no índice em memória.
- Se o documento existir apenas em disco, lê sequencialmente o ficheiro até localizar a entrada com o ID correspondente.

Após localizar o documento, caso tenha sido lido do disco, o documento é adicionado à cache. Em ambos os casos, o documento é movido para o final da fila de utilização, sendo considerado recentemente acedido.

Remover ficheiros

A remoção de um documento durante o programa é um processo simples; o servidor verifica se o ficheiro existe na cache ou no índice em memória e, se sim, remove-o dos da cache e/ou índice.

A eliminação física do documento no ficheiro persistente não é feita imediatamente. Em vez disso, é adiada para o encerramento do servidor, altura em que o ficheiro de indexação é completamente reescrito, de modo a eliminar entradas obsoletas sem incorrer em operações pesadas de reescrita a meio da execução.

Pesquisa sobre o conteúdo de um documento

Esta funcionalidade permite ao utilizador saber em quantas linhas uma palavra ou expressão específica aparece dentro do conteúdo de um documento individual. A pesquisa acontece como descrito em seguinte:

- Dados o ID do documento e a palavra a procurar, o programa localiza o caminho do ficheiro correspondente;
- É criado um processo filho que executa o comando "*grep -c -w*", que conta o número de linhas onde a palavra dada aparece como uma palavra completa no documento (e não como parte de outra).

Para capturar o resultado, um *pipe* é criado entre o pai e o filho e, utilizando *dup2*, o filho redireciona o seu *stdout* para o lado de escrita do pipe. O pai lê então o seu lado do *pipe* e tem finalmente o número de linhas devolvido pelo *grep*.

Pesquisa concorrente sobre o conteúdo de todos os ficheiros

Esta operação permite ao utilizador procurar uma palavra ou expressão em todos os documentos indexados, distribuindo o trabalho por vários processos de forma paralela. É importante notar que especificar o número de processos é opcional, e se nenhum for especificado, o programa corre de forma singular. No final, o cliente recebe os IDs de todos os documentos com essa palavra.

- Primeiramente, o servidor constrói um **GPtrArray** (um *array* dinâmico de pointers da biblioteca *glib*) com apontadores para todos os documentos atualmente indexados. Este conjunto é obtido iterando a cache e o disco, garantindo que apenas documentos válidos são considerados (quando lê o disco verifica que os documentos lidos estão no *index*, para prevenir guardar documentos que tenham sido eliminados durante o programa).
- Em seguida, é dividida a carga de trabalho entre os filhos de forma equitativa, com o número de processos filhos especificados no pedido do cliente. São estabelecidos *pipes* entre o processo pai e os filhos para a comunicação dos resultados.
- Para todos os documentos da sua carga, os filhos correm o comando "*grep -q -w*", que procura de forma silenciosa a existência da palavra procurada no ficheiro. Como não há

saída (*stdout*) com este comando, não é necessário o redirecionamento do *output* e é apenas necessário verificar se o comando terminou com sucesso, prova de que encontrou a palavra. Se tal acontecer, o processo filho escreve o ID do documento no *pipe* para o processo pai.

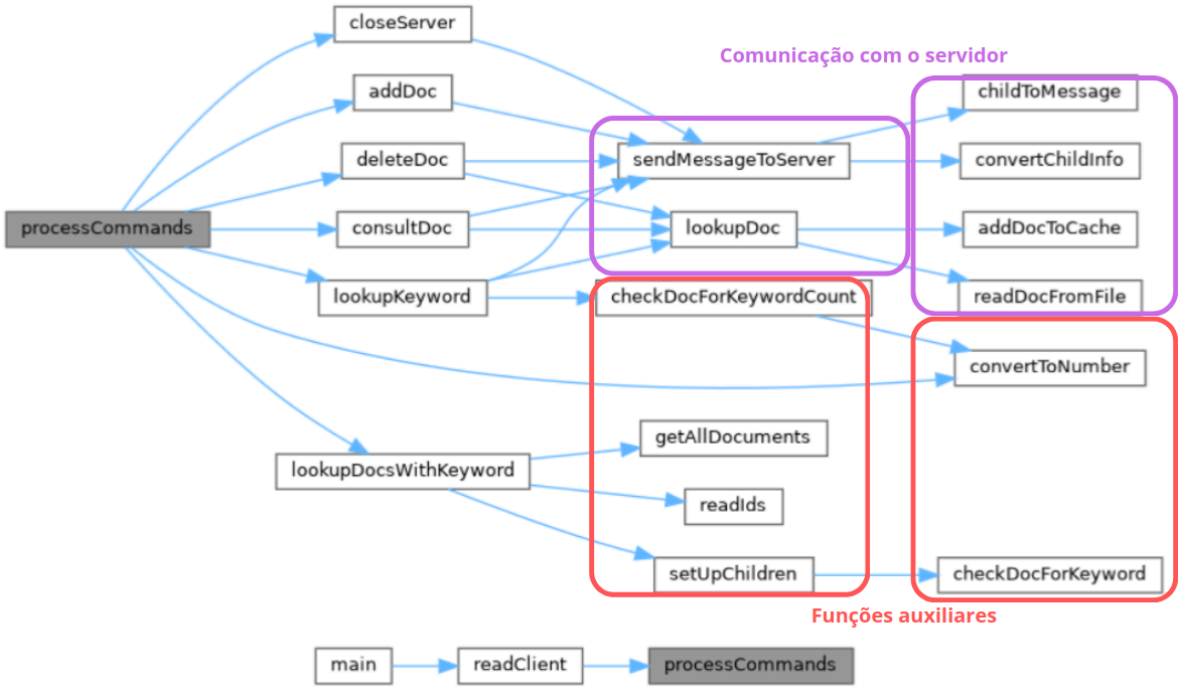
- Enquanto os filhos lêem os documentos, o pai lê continuamente do *pipe* e armazena os IDs dos documentos num **GArray**, um *array* dinâmico fornecido pela *glib*.
- Após todos os processos filhos terminarem, a mensagem é formulada com base nos resultados e enviada para o cliente.

Esta abordagem permite acelerar pesquisas em larga escala drasticamente, explorando o paralelismo do sistema operativo de forma eficiente.

Encerrar servidor

O processo filho envia uma mensagem ao servidor para encerrar, o que começa o processo de limpeza. Irá primeiro refazer o ficheiro em disco, criando um novo ficheiro e reescrevendo todos os documentos do ficheiro antigo para o novo, garantindo primeiro que estes se encontram no *Index*. Em seguida, escreve também os ficheiros guardados apenas em cache para o disco e liberta toda a memória volátil.

Gráfico de funções



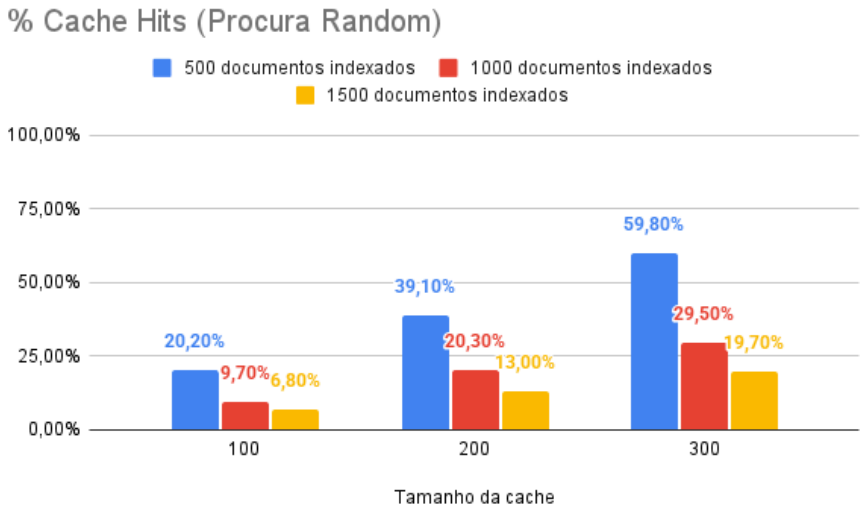
Testes executados

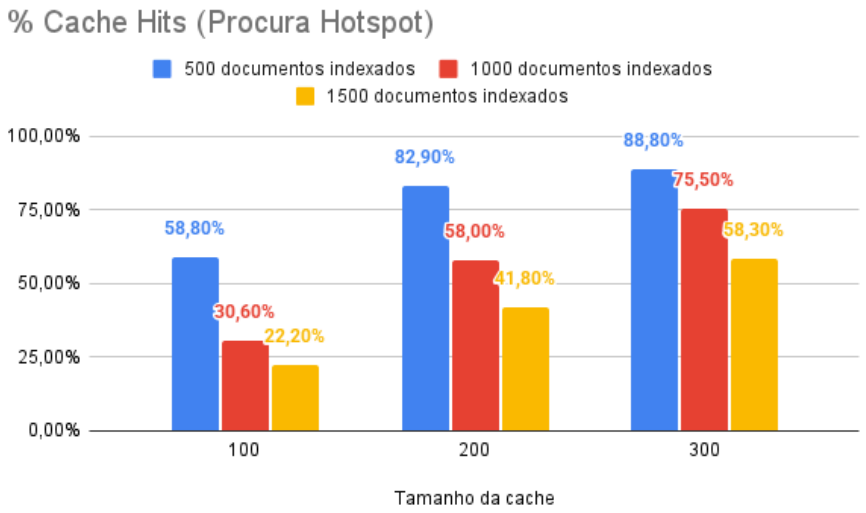
Foram criados *scripts* de teste para avaliar o impacto no desempenho de diferentes configurações de cache e também para avaliar o ganho de desempenho ao paralelizar a pesquisa de documentos. Os resultados aqui descritos foram obtidos da execução destes *scripts* no computador pessoal de um dos elementos do grupo.

O *script* ***test_cache.sh*** avalia o tempo médio que o programa demora a correr 10000 consultas de documentos (comando -c), bem como a percentagem de *cache hits/misses* observados.

O primeiro modo de geração de IDs para consulta criado foi simplesmente escolher um número aleatório entre 1 e o número de ficheiros indexados para teste. No entanto, foi posteriormente adicionada uma opção *hotspot* que segue o princípio de Pareto (regra do 80/20). Este princípio afirma que aproximadamente 80% dos efeitos vêm de 20% das causas. No nosso caso, isso traduz-se em 80% das consultas serem realizadas em 20% dos documentos. Esta opção tenta simular um ambiente mais realista para o funcionamento da cache.

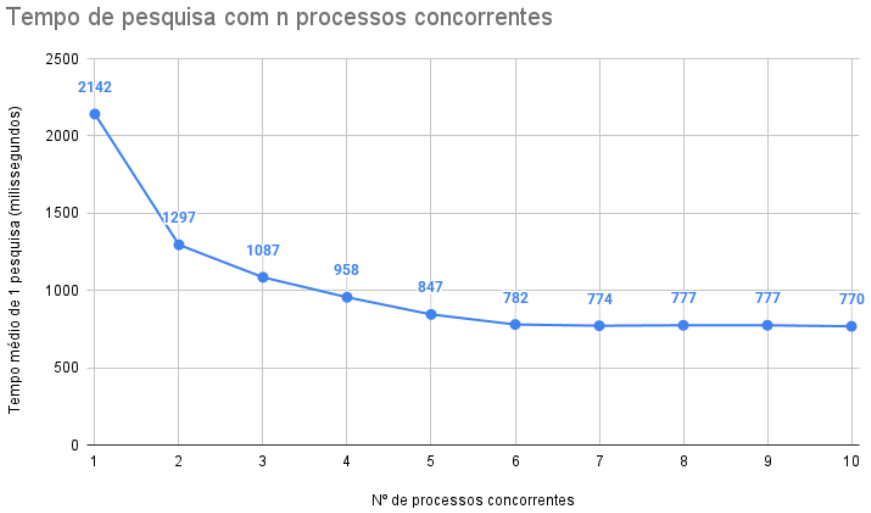
Foram testados os tamanhos de cache 0 (procura em disco), 100, 200, e 300. O tempo de consulta manteve-se estável em todas as diferentes configurações de cache, demorando cada consulta, em média, entre 1 e 1.2 milissegundos, e a totalidade do teste entre 10 e 12 segundos.





A nível de eficiência da cache, pode ser observado nos gráficos acima que o crescimento da percentagem de cache *hits* é bastante semelhante em todas as configurações de cache avaliadas, e que a procura *hotspot* mostra resultados ainda mais promissores. No entanto, uma vez que não houve diferença de tempo de execução entre a procura em disco e em cache, possivelmente devido ao tamanho pequeno da amostra de ficheiros, isto apenas nos diz que o algoritmo de substituição implementado, ***LRU***, é de facto eficiente quando aplicado em contexto real. Será de esperar que, na ordem dos milhões de ficheiros, e com um tamanho de cache apropriado, haja melhoria no tempo de execução.

O *script* ***test_concurrency.sh*** avalia o tempo médio que o programa leva para executar uma pesquisa (comando -s) utilizando diferentes quantidades de processos concorrentes. Os valores apresentados correspondem à média de 100 execuções.



A melhoria no desempenho é clara, porém observa-se que a redução no tempo de execução não é linear, e que por volta dos 8 processos concorrentes, o tempo estabiliza, indicando que o acréscimo de mais processos não significa ganhos relevantes de desempenho. Isto deve-se certamente a limitações de *hardware*, não podendo ser suportados mais processos concorrentes que *threads* lógicas disponíveis.

Conclusão

O sistema desenvolvido apresenta uma arquitetura modular e cumpre os requisitos estabelecidos no enunciado o melhor possível tendo em conta o tempo disposto. Existe uma separação clara entre cliente e servidor, uma utilização de processos paralelos para lidar com múltiplos pedidos ao mesmo tempo, e gestão eficiente de cache e persistência de documentos, que resulta num programa capaz de escalar de forma eficiente com o número de clientes e ficheiros.

A comunicação entre componentes usa estruturas de tamanho fixo para evitar inconsistências nas escritas em *pipes*, facilitando futuras expansões possíveis. A escolha cuidadosa das estruturas de dados, como *hashtables* para indexação rápida e filas para a gestão LRU, permite otimizar o desempenho tanto em operações de acesso frequente como em persistência de longo prazo. O uso destas estruturas fornecidas pela *glib* facilita a legibilidade não só por serem bem conhecidas, mas também testadas, sendo fáceis e rápidas de implementar, poupando-nos tempo e eventuais dificuldades caso escolhêssemos criá-las nós próprios.

Adicionalmente, de modo a garantir a qualidade, legibilidade e manutenção do código, todo o projeto foi documentado recorrendo ao **Doxygen**, permitindo gerar automaticamente uma documentação detalhada do programa. Isto não só torna mais fácil a manutenção futura, como também ajuda qualquer novo programador que precise de entender ou expandir o código.

No geral, acreditamos que o projeto apresenta uma boa base para futuros desenvolvimentos e melhorias. Se tivéssemos mais tempo para desenvolver o projeto, iríamo-nos certificar do encapsulamento dos módulos e de um melhor tratamento de erros, principalmente a prevenção dos mesmos.