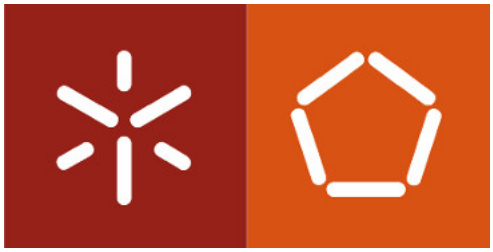

Universidade do Minho
Licenciatura em Engenharia Informática

Sistemas Operativos
Serviço de Indexação e Pesquisa de Documentos

Grupo 3
Luis Ferreira - A98286

16 de maio de 2025

Índice

1 Introdução

O presente relatório representa detalhadamente a implementação de um serviço projetado para permitir a gestão e consulta eficiente da meta-informação de documentos de texto armazenados localmente. Este sistema permite adicionar, remover e consultar documentos indexados, bem como realizar pesquisas sobre o seu conteúdo com base em palavras-chave.

A implementação do sistema foi realizada em C, utilizando conceitos avançados de programação de baixo nível, como manipulação de processos, comunicação entre processos através de pipes e uso de chamadas ao sistema operacional Unix.

O relatório detalha a arquitetura do sistema, destacando as decisões tomadas durante o processo de desenvolvimento. Incluem-se ainda excertos de código relevantes que ilustram em prática as funcionalidades do sistema.

Por fim, são discutidos os resultados obtidos, incluíndo a eficiência do sistema na indexação e pesquisa de documentos, possíveis limitações e sugestões para melhorias.

2 Arquitetura do Sistema

A arquitetura do sistema é composta por três componentes principais: o cliente, o servidor e o *manager* de documentos, sendo também suportada por uma componente auxiliar, a *cache*.

O cliente é responsável por construir e enviar pedidos ao servidor. Antes de cada envio, cada pedido é validado na parte do cliente, de modo a garantir que todos os campos obrigatórios estão preenchidos, poupando trabalho ao servidor. O servidor recebe estes pedidos e trata-os de acordo com a complexidade do pedido solicitado, devolvendo posteriormente a resposta ao cliente através do seu *FIFO*.

O servidor, ao receber pedidos do tipo consulta ou pesquisa (operações potencialmente demoradas), cria processos filhos através da chamada ao sistema *fork()*, delegando-lhes a execução do trabalho. Esta estratégia evita que o processo principal fique bloqueado, permitindo o processamento e execução de pedidos de múltiplos clientes em simultâneo. Os pedidos mais simples, como indexação ou remoção, são tratados diretamente no processo principal, uma vez que envolvem a manipulação de estruturas de dados globais partilhadas.

O *manager* de documentos tem como principal função armazenar e gerir a meta-informação dos documentos indexados, título, autores, ano e caminho do ficheiro. Esta estrutura de dados é construída sobre uma *Hash Table* que permite o acesso rápido dos documentos através do seu *ID*.

A cache funciona como uma camada intermédia para acelerar o acesso aos documentos mais frequentemente consultados. É implementada com suporte a políticas de substituição (*FIFO* ou *LRU*), e reside exclusivamente na memória do processo servidor principal, garantindo consistência e integridade dos dados.

3 Comunicação Servidor - Cliente

A comunicação entre cliente e servidor é feita por meio de *FIFOs* nomeados, *dclient_fifo* e *dserver_fifo* respetivamente, garantindo isolamento entre sessões de clientes distintos. Esta comunicação é baseada num modelo de soliticação onde o cliente envia um pedido para o servidor e a resposta posterior para o respetivo *FIFO* do cliente.

3.1 Cliente - Servidor: Pedido

Ao executar o cliente, este começa por criar um *FIFO* próprio com o nome *"CLIENT_PID"*, onde o *PID* é o identificador do processo, através do getpid(), garantindo que cada cliente tem um canal de resposta exclusivo.

Por sua vez, é construído o pedido numa estrutura *"Request"* que contém o *PID* do cliente, o tipo do pedido e a informação do pedido, no caso de uma operação de indexação, o tipo do pedido é "-a" e a informação engloba toda a meta-informação. O envio de um pedido decorre quando o cliente abre o *FIFO* do servidor, no modo *O_WRONLY* e escreve a estrutura que representa o pedido.

Ao executar o servidor, é inicializada a estrutura de dados que armazena todos os documentos a serem indexados *DocumentManager*, são carregados documentos anteriormente indexados (da sessão anterior) através do ficheiro correspondente, se existir, e é inicializada a *Cache*. O servidor começa também por criar o seu *FIFO* próprio com o nome *"SERVER"*. Para garantir que o servidor não receba um sinal de fim de ficheiro (*EOF*) quando um cliente termina a escrita no *FIFO* do servidor, é aberto um descritor de escrita adicional (*dummy_fifo*). Este descritor é mantido aberto pelo próprio servidor, assegurando que o *read()* sobre o *FIFO* principal não termine prematuramente entre pedidos.

O servidor que por sua vez se encontra à escuta, lê cada pedido recebido no seu *FIFO* através da chamada ao sistema *read()*, com base no tipo de pedido, o servidor processa no seu processo principal se o pedido não for do tipo consulta/pesquisa, caso contrário é criado um processo filho através da chamada ao sistema *fork()* para que fique encarregue de processar esses pedidos mais complexos.

Segue-se os seguintes anexos para ilustrar a execução do servidor e do cliente para enviar um pedido, é de se notar que o servidor possui dois argumentos, o *path* onde os documentos estão organizados localmente e o tamanho da *cache*

Figura 1: *Run SERVER*

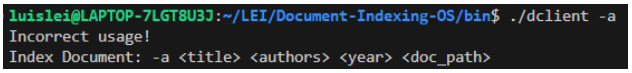

Figura 2: *Run CLIENT*

3.2 Servidor - Cliente: Resposta

Após processar o pedido, o servidor responde através do *FIFO* exclusivo do cliente, de acordo com o PID do cliente incluído na estrutura *"Request"*, a resposta encontra-se no campo *"response"* desta estrutura

O cliente abre o seu *FIFO*, no modo *O_RDONLY*, e lê a estrutura *"Request"* devolvida pelo servidor que contém a resposta e escreve-a para o seu *"stdout"*. Em seguida é fechado o *FIFO* do cliente e é feita a limpeza do mesmo através de *"unlink(CLIENT_PID)"*

4 Estrutura *Request*

A estrutura *"Request"* representa cada pedido solicitado por um Cliente ao Servidor. Esta estrutura é composta por diversos campos que contém informações importantes sobre o pedido a ser executado.

Entre esses campos, o *cmdType* representa o tipo de pedido a ser processado pelo servidor, os tipos de pedidos podem ser, "-a" "title" "authors" "year" "path" para indexar um documento, "-c" "key", para consultar a meta-informação de um documento, onde *key* representa a chave do documento indexado, "-d" "key" para remover a meta-informação do respetivo documento correspondente à chave *key*, "-l" "key" "keyword", para devolver o número de linhas que um dado documento identificado pela *key* contêm a dada palavra-chave (*keyword*), "-s" "keyword" para devolver uma lista de identificadores (que representam documentos) que contêm a dada palavra-chave, o mesmo comando pode ser executado com uma *flag* adicional "nr_processes" que representa o número de máximo processos a executar simultâneamente.

O campo *info* é pré-processado no cliente e representa todos os campos necessários de cada comando, separados com uma barra vertical "|", o campo *PID* já explicado anteriormente, representa o identificador do processo do cliente, o campo *duration* utilizado apenas para operações do tipo consulta ("-c") é incluído na mensagem de resposta do servidor para informar ao cliente quando tempo demorou a consulta da meta-informação, O campo *response* com um tamanho fixo de 1024 *bytes* representa a resposta a ser escrita e lida pelo servidor e cliente, respetivamente, por fim o campo *start_time* é utilizado para medir o tempo que o servidor demorou para consultar a meta-informação.

```c
// Structure to represent a command to run
typedef struct request{
    CommandType cmdType;
    char info[512];
    int pid;
    int duration;
    char response[1024];
    struct timeval start_time;
}Request;
```

Figura 3: Estrutura *Request*

5 Estrutura *Documents/DocumentManager*

De modo a armazenar a meta-informação indexada de cada documento, tem-se a estrutura *Documents*, os tamanhos dos campos *title*, *authors* e *path* são pré-definos com um tamanho fixo de 200 *bytes*, 200 *bytes* e 64 *bytes*, respetivamente, como se pode verificar no seguinte anexo:

```c
// Defines the max size for the following fields
#define TITLE_SIZE   200
#define AUTHORS_SIZE 200
#define PATH_SIZE    64

// Struct to represent a document
typedef struct{
    char title[TITLE_SIZE];
    char authors[AUTHORS_SIZE];
    char path[PATH_SIZE];
    int year;
    int id; // document's id used as key
}Document;
```

Figura 4: Estutura *Documents*

Em seguida, a estrutura responsável por armazenar todos os documentos indexados foi implementada com auxilio da *Glib* para o uso de *Hash Tables* de modo que a indexação, consulta e remoção de documentos seja o mais eficiente possível, esta abordagem é fundamental para manter a escalabilidade do sistema à medida que o número de documentos cresce.

```c
// Struct to store all documents indexed in a GHashTable
typedef struct{
    GHashTable* documentTable;
}DocumentManager;
```

Figura 5: Estutura *DocumentManager*

6 Estrutura *Cache*

A estrutura *Cache* tem como objetivo acelerar o acesso a documentos frequentemente consultados, evitando leituras repetidas da estrutura principal de documentos. Para isso, combina duas estruturas da *GLib*. Uma *HashTable* (*docEntry*), que representa os documentos recentemente consultados encapsulados na estrutura *CacheEntry* que armazena cada documento. Uma *GQueue* (*order*), que mantém a ordem dos documentos armazenados, utilizada para implementar a política de substituição: *FIFO* (*First-In-First-Out*) ou *LRU* (*Least Recently Used*). Além disso, a estrutura conta com os campos auxiliares, *capacity* que representa a capacidade máxima da cache (número de documentos), *current_size* que representa a capacidade atual, *policy* representa a política de substituíção, *hits/misses* representam contadores estatísticos usados para avaliar a eficiência da cache.

```c
/**
 * Structure to represent the Cache
 */
typedef struct{
    GHashTable* docEntry;   // key: doc_id, value: CacheEntry*
    GQueue* order;          // FIFO or LRU order tracking
    int capacity;
    int current_size;
    int policy;             // 0 -> FIFO | 1 -> LRU
    int hits;
    int misses;
}Cache;

/**
 * Structure that stores a Document in the Cache
 */
typedef struct{
    Document* metaInfo;
}CacheEntry;
```

Figura 6: Estutura *Cache*

7 Comandos

7.1 Indexação -a

Para solicitar um pedido do tipo indexação, o cliente terá que ser inicializado com a *flag* -a e os campos *"title" "authors" "year" "path"*, quando processado por parte do Servidor, é recebida uma resposta de *feedback* sobre o pedido. Em anexo podemos verificar um exemplo de indexação de um documento.

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dclient -a "Romeo and Juliet" "William Shakespear" "1996" "1112.txt"
Document 1648 indexed with success.
```

Figura 7: Indexar Documento - Cliente

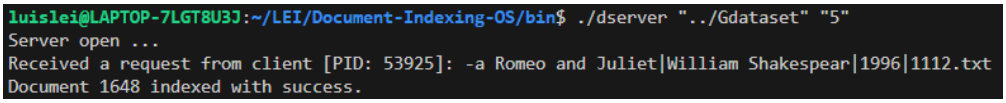

Figura 8: Indexar Documento - Servidor

7.2 Consultar -c

Para solicitar um pedido do tipo consulta, o cliente terá que ser inicializado com a *flag* -c e o campo *key* que representa a chave do documento a ser consultado, quando processado por parte do servidor, é recebida uma resposta de *feedback* sobre o pedido, juntamente com o tempo que demorou para encontrar a meta-informação do documento solicitado.

O servidor, quando recebe um comando do tipo consulta é criado um processo filho através da chamada ao sistema *fork()* e é criada uma pipe para que este processo filho possa comunicar com o processo pai, o processo filho começa então por encontrar o documento, se ele se encontrar na *Cache* é porque se trata de um *hit*, caso contrário é um *miss*, é calculado o tempo que custou a cada estrutura de dados para encontrar tal documento, no fim, desta procura, o processo filho escreve numa estrutura auxiliar se ocorreu um *hit* ou *miss* na *Cache*.

O processo pai lê na pipe a informação da estrutura auxiliar e fica então encarregue de fazer alocações de memória e/ou mudanças na cache conforme necessário. No caso da política de substituição *LRU* é atualizada a *Queue* quando ocorre um *hit* para atualizar a posição do documento acedido para que seja o mais recente, no caso de um *miss* é indexado o documento na *CacheEntry* para que posteriormente, caso seja pedido o mesmo documento, seja possível encontrá-lo na *Cache*.

Em anexo podemos verificar um exemplo de consulta de um documento.

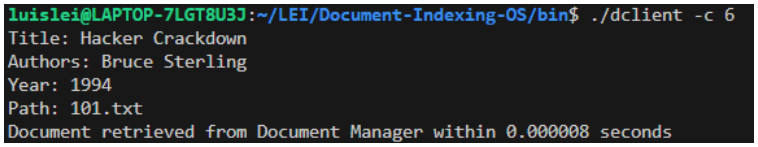

Figura 9: Consultar Documento - Cliente

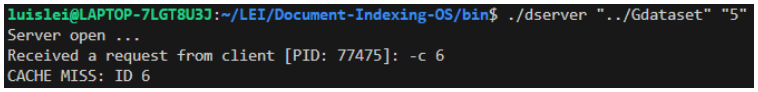

Figura 10: Consultar Documento - Servidor

7.3 Remover -d

Para solicitar um pedido do tipo remoção, o cliente terá que ser inicializado com a *flag* -d e o campo *key* que representa a chave do documento a ser removido, quando processado por parte do servidor, é recebida uma resposta de feedback sobre o pedido.

O servidor, quando recebe um comando do tipo remoção, remove o documento da estrutura *Document-Manager* e da estrutura *CacheEntry* se existir.

Em anexo podemos verificar um exemplo de remoção de um documento.

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dclient -d 1648
Index entry 1648 deleted with success.
```

Figura 11: Remover Documento - Cliente

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dserver "../Gdataset" "5"
Server open ...
Received a request from client [PID: 81785]: -d 1648
Received Key: 1648
Tried to remove doc_id 1648 from cache, but it was not found.
Index entry 1648 deleted with success.
```

Figura 12: Remover Documento - Servidor

7.4 Número de Linhas -l

Para solicitar um pedido do tipo pesquisa número de linhas que um dado documento tem face a uma palavra-chave, o cliente terá que ser inicializado com a *flag* -l e os campos *key "keyword"* que representam a chave do documento e a palavra-chave, respetivamente, quando processado por parte do servidor, é recebida uma resposta de feedback sobre o pedido. Este pedido tem como objetivo contar o número de linhas do documento que contêm a palavra-chave fornecida.

O servidor, quando recebe um comando do tipo número de linhas que se trata de uma operação de pesquisa, cria um processo filho através da chamada ao sistema *fork()* para que fique responsável por executar o respetivo comando. Neste processo filho, é em primeiro lugar procurado o documento com a *key* recebida, se existir, é criada uma pipe e um outro processo para que fique encarregue por executar o comando *grep* para o documento com o seu respetivo *path*, antes da execução deste comando, é redirecionado o resultado dele pelo *stdout* para o fim da escrita da pipe, o primeiro processo filho, aguarda pela execução do seu processo filho para que possa ler na pipe o resultado do comando *grep* e escreve a mensagem de resposta conforme o número de linhas que contêm a dada palavra-chave no documento dado.

Em anexo podemos verificar um exemplo de pesquisa de número de linhas que um documento contêm uma dada palavra chave.

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dclient -l 32 "love"
Keyword "love" found in 58 lines of document ID 32.
```

Figura 13: Número de Linhas Documento - Cliente

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dserver "../Gdataset" "5"
Server open ...
Received a request from client [PID: 90075]: -l 32|love
```

Figura 14: Número de Linhas Documento - Servidor

7.5 Listagem de Identificadores -s

Para solicitar um pedido do tipo listagem de identificadores que contêm uma dada palavra-chave, o cliente terá que ser inicializado com a *flag* -s e o campo *keyword* que representa a palavra-chave, caso o cliente pretender a mesma operação só que recorrendo a multíplos processos, terá então de incluir um outro campo *nr_processes* que representa o número de processos a executarem simultâneamente o pedido, quando processado por parte do servidor, é recebida uma resposta de *feedback* sobre o pedido. Este pedido tem como objetivo listar todos os identificadores de documentos que contêm a palavra-chave fornecida.

O servidor, quando recebe um comando do tipo listagem de identificadores de documentos que se trata de uma operação de pesquisa, cria um processo filho através da chamada ao sitema *fork()* para que fique responsável por executar o respetivo comando. Neste processo filho, tanto como um único processo ou vários *nr_processes* a executar o comando, é feita uma iteração pela estrutura de dados *DocumentManager* que contêm todos os documentos indexados no servidor e é feita a mesma análise no Número de Linhas -l. Para cada resultado do comando *grep* que o número de linhas seja superior a 0, é escrito o *ID* do documento em questão, caso o cliente solicite um número de processos a executar o pedido simultâneamente, é criada uma estrutura auxiliar, que contêm o *ID* do documento a ser iterado, o *PID* do processo e a *pipe*, para que o segundo processo filho que executa o comando *grep* possa escrever escrever o resultado do comando *grep*, enquanto isso o primeiro processo filho aguarda pela execução de cada processo filho para que possa atualizar o número de processos ativos. No fim, é percorrida a estrutura auxiliar para que possa listar os identificadores dos documentos como resposta ao cliente.

Em anexo podemos verificar um exemplo da listagem de identificadores de documentos que contêm a dada palavra-chave.

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dclient -s "abdicate"
[928, 187, 188, 931, 375, 746, 1117, 376, 747, 1489, 194, 1494, 753, 386, 387, 573, 388, 1130, 1323, 26, 27, 954, 398, 1141, 586, 1143, 1514, 219, 777, 1527, 1160, 605, 1165,
 426, 1354, 242, 1170, 431, 621, 66, 252, 254, 999, 259, 1557, 1558, 261, 1559, 1560, 819, 1561, 264, 640, 826, 832, 91, 1581, 99, 470, 1216, 846, 291, 662, 295, 667, 672, 10
43, 1416, 1417, 676, 863, 678, 309, 496, 1239, 1056, 871, 874, 876, 135, 877, 322, 1435, 879, 1250, 1436, 1437, 511, 1438, 1068, 1257, 1629, 890, 891, 1634, 893, 1638, 712, 7
13, 1455, 1642, 530, 1279, 355, 916, 361, 918, 733, 363, 1290, 736, 185]
```

Figura 15: Listagem de Identificadores - Cliente

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dserver "../Gdataset" "5"
Server open ...
Received a request from client [PID: 106852]: -s abdicate|1
```

Figura 16: Listagem de Identificadores - Servidor

7.6 Paragem -f

Para solicitar um pedido do tipo paragem do servidor, o cliente terá que ser inicializado com a *flag* -f, quando processado por parte do servidor, é recebida uma resposta de *feedback* tal que o servidor foi parado com sucesso.

O servidor, quando recebe um pedido do tipo paragem, deixa de correr o ciclo *while* passando a variável *isRunning* a 0, após o ciclo, é fechado os *FIFOs "server_fifo"* e *"dummy_fifo"*, são escritos todos os documentos na estrutura *DocumentManager* para o ficheiro de documentos para ser lido na próxima sessão e é libertada a memória da estrutura. Ainda antes do servidor ser fechado, é escrito no *stdout* do servidor, o resumo da cache da sessão atual, dando a conhecer os *hits/misses/miss rate*, posteriormente é libertada a memória dedicada à *Cache* e é removido o *FIFO* do servidor através de "*unlink(SERVER)*".

Em anexo podemos verificar um exemplo de uma paragem do servidor.

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dclient -f
Server is shutting down ...
```

Figura 17: Paragem do Servidor - Cliente

```
luislei@LAPTOP-7LGT8U3J:~/LEI/Document-Indexing-OS/bin$ ./dserver "../Gdataset" "5"
Server open ...
Received a request from client [PID: 111571]: -f
Server is shutting down ...
Cache stats: 0 hit(s), 0 miss(es), hit rate: 0.00%
```

Figura 18: Paragem do Servidor - Servidor

8 Discussão dos Resultados

8.1 Análise Listagem de Identificadores

Foi criado um script que permite correr vários clientes a pedirem sequencialmente uma listagem de identificadores que contêm uma dada palavra-chave, com um número de processos variado, desde a um único processo, até 32 processos em simultâneo. Este mesmo script mede o tempo desde que o comando é recebido pelo servidor até que o cliente escreva no seu *stdout*.

```
command,arg,processes,time_ms
search,war,1,16579
search,war,2,3233
search,war,4,1828
search,war,8,1639
search,war,16,1794
search,war,32,1834
search,science,1,3348
search,science,2,2125
search,science,4,1734
search,science,8,1714
search,science,16,1642
search,science,32,1508
search,artificial,1,3090
search,artificial,2,2284
search,artificial,4,1582
search,artificial,8,1803
search,artificial,16,1724
search,artificial,32,1777
search,data,1,3847
search,data,2,2520
search,data,4,1891
search,data,8,2098
search,data,16,1750
search,data,32,1778
```

De acordo com os seguintes resultados, é possível verificar, um ganho significativo ao correr o comando para 2 processos em simultâneo em comparação com um único processo, já para 4 processos em simultâneo,

o ganho já não é tão significativo, em comparação com 2 processos em simultâneo, mas mesmo assim, demonstra uma melhor *performance*. Tal pode ser justificado, pelo simples facto do trabalho que estaria a ser realizado por um único processo, ser realizado por vários.

Contudo, tal não se parece comprovar para um número de processos em simultâneo superior a 8, para alguns dos resultados obtidos, é possível observar que o ganho comparado com 4 processos em simultâneo é insignificativo ou pior. A razão para este fenómeno pode ser justificado pela limitação de hardware, dado que o dispositivo usado para testar este sistema possui 4 *cores* o que significa que o dispositivo só pode executar até 4 processos verdadeiramente em parelelo, os processos acima desse número entram em competição pelos mesmos *cores*, levando a um *context switching* (troca de contexto entre processos), o que tem custo de tempo considerável para estas operações.

Também se deve ao *overhead* de criação e sincronização de processos, cada chamada ao sistema *fork()* e *pipe* custa tempo e memória, o processo pai necessita de gerir as *pipes* através da leitura do conteúdo e precisa de esperar pelos processos filhos através da chamada ao sistema *wait()*. Um outro motivo que sustenta esta falta de ganho, reside no *I/O* e no disco (neste caso *HDD*), a operação *grep* envolve leitura de documentos no disco, que por sua vez, não é capaz de responder a muitos acessos em paralelo de forma eficiente

8.2 Análise Consulta de Documento - *FIFO/LRU*

Para uma *Cache* inicializada com uma capacidade de até 5 documentos foi feito um teste para verificar quanto tempo em média a estrutura *DocumentManager* levava até encontrar e devolver documento solicitado, em média este resultado foi de 20,8μs.

Testando novamente os mesmos 5 documentos solicitados para uma política de substituição *FIFO* o tempo médio foi de 15,6μs, o seguinte padrão favorece esta política de substituição, uma vez que estamos sempre a aceder aos mesmos 5 documentos na *Cache* o que faz com que nenhum documento saia da nossa *Queue*.

Para a política de substituição *LRU* foi testada uma sequência 5 documentos que favorecia a *LRU*, mesmo em situações em que um novo documento fosse solicitado e caso voltasse a pedir um documento que ainda estivesse na *Queue* continuava a ter uma ótima *performance* com um tempo médio de 7μs

Para esta análise podemos concluir que a política *LRU* apresenta um desempenho superior em padrões de acesso repetido com reutilização recente de documentos, enquanto a política*FIFO* comporta se adequadamente apenas em padrões lineares e previsíveis.

9 Dificuldades Sentidas e Conclusões

O seguinte projeto foi elaborado com alguma dificuldade principalmente ao resolver problemas que envolviam vários processos em simultâneo, contudo, a gestão de tempo para o dado projeto permitiu a realização de todas as tarefas com eficácia. Uma das limitações que o projeto enfrenta reside na maneira como o servidor escreve a mensagem de resposta ao cliente, a resposta é representada pela a estrutura *Request* no campo *response*, este mesmo campo possui um tamanho fixo de 1024 *bytes* o que parece ser suficiente, contudo, não é maneira mais correta de cuidar deste problema. Em casos cuja a listagem de identificadores de documentos é muito grande, o servidor só consegue escrever até esse dado tamanho fixo, o que leva a perda de informação.

Foi pensada numa solução para que o servidor podesse escrever a resposta toda para o cliente, utilizando pipes entre os processos filhos que são responsáveis por comandos do tipo consulta/pesquisa. Porém, foi encontrado um outro problema sobre o *read()* do servidor estar bloqueado, uma vez que, o servidor estaria a aguardar até que o processo filho terminasse o comando e escrevesse na *pipe* para que o servidor pudesse ler, o que fez com que outros clientes ficassem bloqueados, o servidor continuava a executar em prática os comandos, mas não seria capaz de responder a todos os clientes.

Apesar desta limitação sobre o tamanho fixo da resposta, o servidor continua a ser capaz de receber vários pedidos de clientes e responder aos vários pedidos sem que um cliente fique bloqueado devido a outro cliente ter pedido um comando de pesquisa/consulta.

Em suma, o presente trabalho prático permitiu aplicar de forma concreta os conceitos abordados na unidade curricular, permitindo uma compreensão mais aprofundada do comportamento dos processos e da comunicação entre eles. O sistema de indexação e pesquisa de documentos desenvolvido ao longo deste projeto revelou-se uma solução funcional, abrangente e eficaz face aos objetivos propostos.