



Escola de Engenharia

**Universidade do Minho – Departamento de Informática**  
**Mestrado Integrado em Engenharia Informática (MIEI)**  
**Licenciatura em Engenharia Informática (LEI)**

**2021/2022**

## **RELATÓRIO – Trabalho Prático**

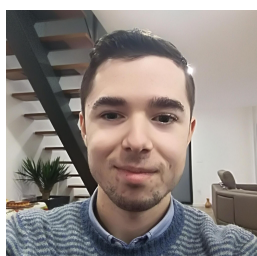
### **Sistemas Operativos (SO)**



### **SDStore: Armazenamento Eficiente e Seguro de Ficheiros**

### **Grupo TP-MIEI 8**

**A92847, Guilherme Sousa Silva Martins**  
**A94942, Miguel Velho Raposo**  
**A97777, Millena de Freitas Santos**



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Programas para Transformações</b>	<b>1</b>
<b>3</b>	<b>Comunicação entre o Cliente e o Servidor</b>	<b>2</b>
<b>4</b>	<b>Funcionalidades básicas</b>	<b>2</b>
4.1	Programa Servidor - <i>Sdstored</i> . . . . .	2
4.1.1	Processamento . . . . .	3
4.1.2	Execução de transformações . . . . .	3
4.1.3	Finalização . . . . .	4
4.2	Programa Cliente - <i>Sdstore</i> . . . . .	4
<b>5</b>	<b>Estruturas de dados e Variáveis auxiliares</b>	<b>5</b>
5.1	Variáveis auxiliares (Mais importantes) . . . . .	5
5.2	Request . . . . .	6
5.3	Reply . . . . .	6
5.4	Transf . . . . .	7
5.5	Process . . . . .	7
5.6	TProcess . . . . .	8
<b>6</b>	<b>Funcionalidades Avançadas</b>	<b>8</b>
<b>7</b>	<b>Conclusão</b>	<b>9</b>

## 1 Introdução

O trabalho prático foi realizado por base no sistema operativo Linux, na distribuição Ubuntu, como ambiente de desenvolvimento e de execução.

Este projeto consistiu na implementação de um serviço que permite aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e eficiente, poupando espaço de disco. Para tal, o serviço criado disponibiliza funcionalidades de compressão e cifragem dos ficheiros a serem armazenados (todas as funcionalidades foram disponibilizadas pela equipa docente).

O serviço permite a submissão de pedidos para processar e armazenar novos ficheiros bem como para recuperar o conteúdo original de ficheiros guardados previamente. Ainda, é possível consultar as tarefas de processamento de ficheiros a serem efetuadas num dado momento, bem como estatísticas sobre as mesmas.

Para tal, foi desenvolvido um cliente (programa *sdstore*) que oferece uma interface com o utilizador via linha de comando; e um servidor (programa *sdstored*), mantendo em memória a informação relevante para suportar as funcionalidades disponibilizadas. Durante a escrita do código para o programa, tivemos o cuidado de não criarmos ficheiros temporários, nem recorreremos à utilização das funções da biblioteca C de operações sobre ficheiros.

## 2 Programas para Transformações

Os diferentes programas de transformação de ficheiros estão enunciados a seguir:

- **bcompress / bdecompress**: Comprime / descomprime dados com o formato bzip.
- **gcompress / gdecompress**: Comprime / descomprime dados com o formato gzip.
- **encrypt / decrypt**: Cifra / decifra dados.
- **nop**: Copia dados sem realizar qualquer transformação.

O código-fonte (inalterado) das várias transformações contém que ser aplicada ao ficheiro original está presente na pasta */bin*. Para facilitar a organização e o acesso dos dados relacionados com as transformações, atribuímos os seguintes índices para as várias transformações:

0 – encrypt, 1 – decrypt, 2 – bcompress, 3 – bdecompress  
4 – gcompress, 5 – gdecompress, 6 – nop.

Para todo o outro tipo de string recebida, a função devolve -1.

No código-fonte do servidor (*sdstored.c*) criamos a função `hash_transf`, para que fosse possível converter a *string* da transformação recebida no seu respetivo índice.

### 3 Comunicação entre o Cliente e o Servidor

Tanto o cliente como o servidor comunicam entre si via pipes com nome (FIFO). As pipes com nomes que são criadas ao longo da execução do programa são guardadas na pasta *namedpipes* (criada inicialmente pelo Makefile). É usada a pipe com nome *CLIENT\_TO\_SERVER\_FIFO* para transferir informações do cliente para o servidor. Já para comunicação servidor-cliente o cliente cria uma pipe também salva na pasta *namedpipes* porém nomeada pelo seu pid. Fizemos desta forma pois já que um cliente só pode efetuar um pedido, então só precisamos de uma pipe cliente-servidor, mas como o servidor pode enviar mais que uma resposta ao cliente, por exemplo no caso de pedidos proc-file, então tornou-se necessário que estas respostas sejam enviadas pra pipes únicas específicas ao cliente, para que não tenha problema no acesso e leitura principalmente devido à concorrência.

Assim, estes são os únicos ficheiros temporários que são usados. Logo que os respetivos programas terminem, todas as pipes são fechadas e os seus ficheiros são imediatamente “apagados”, com recursos às funções `close_handler` no servidor e nos respetivos clientes.

Tanto para pedidos como para respostas limitamos o tamanho da mensagem ao utilizar uma matriz. Desta forma, não precisamos utilizar funções como `readln` ou até mesmo ler carácter por carácter a procura de quebra de linha o que tornaria muito mais ineficiente. Passamos a estrutura e lemos o tamanho da estrutura e isto facilita e também melhora o desempenho.

### 4 Funcionalidades básicas

O serviço implementado é composto por dois tipos de funcionalidades: básicas e avançadas. Foram aplicadas todas as funcionalidades básicas propostas no enunciado do trabalho. Elas serão enunciadas nos seus respetivos programas:

#### 4.1 Programa Servidor - *Sdstored*

O programa servidor deve receber 2 argumentos pelo terminal. O primeiro corresponde ao caminho para um ficheiro de configuração e o segundo para a pasta onde estão guardados os executáveis dos vários programas.

```
$ ./sdstored config-filename transformations-folder
$ ./bin/sdstored etc/sdstored.conf bin/
(exemplo para teste, não esquecer da barra depois do bin)
```

O ficheiro de configurações contém o número máximo de cada tipo de transformação que o servidor pode executar concorrentemente ao mesmo tempo. Um exemplo de ficheiro válido é o seguinte:

```
nop 3
bcompress 4
bdecompress 4
gcompress 2
gdecompress 2
encrypt 2
decrypt 2
```

Quando este recebe um comando do cliente através da pipe, antes de iniciar a sua (possível) execução, este verifica de que se trata o comando.

Caso seja do tipo *status*, a função `send_server_status` trata de enviar ao respetivo cliente os dados pedidos.

#### 4.1.1 Processamento

Por outro lado, caso seja um *proc-file*, as funções `verify_transfs_name` e `verify_transfs_names` verificam se o(s) nome(s) da(s) transformação/ões pedida(s) no comando são válido(s), respetivamente. Se forem válidas, o pedido é colocado na fila e a função `run_process` é chamada para percorrer a fila de pedidos prontos a serem executados e executá-los.

Esta função `run_process` percorre a fila dos pedidos prontos que é um array, e caso o pedido esteja no estado ready, ou seja, não está sendo executado, e caso o servidor tenha quantidade livre o suficiente das transformações necessárias para concluir o pedido, este então é marcado como running, é enviada uma mensagem ao cliente que o seu pedido está sendo processado, as transformações necessárias são reservadas e é executado.

Ao longo da execução, o utilizador é informado (via *standard output* do cliente) caso o pedido tenha ficado pendente, quando entra em processamento, e quando é concluído. No final, informa o número de bytes do ficheiro original e o número de bytes do ficheiro final.

#### 4.1.2 Execução de transformações

Há uma função para pedidos que contém apenas uma transformação e outra para pedidos que contém múltiplas transformações. Ambas utilizam redirecionamento do stdin e do stdout para que a chamada da system call `execl` seja corretamente feita para o ficheiro de entrada e o ficheiro de saída.

No caso da execução de vários pedidos, assim como aprendemos nas aulas práticas, foi necessária a criação de  $N - 1$  pipes e a divisão entre os três possíveis casos: primeira transformação, transformações intermediárias e última transformação. Quando esta última transformação é concluída,

queremos que o servidor prossiga a terminar o pedido. Para isto acontecer no momento correto, é preciso certificar que é apenas chamada após a última transformação ter sido executada. Portanto, a forma que encontramos foi criar um filho para a chamada da system call `exec` da última transformação e, o pai deste filho fica à espera e apenas quando o filho é concluído que chama a função que trata de finalizar o processo atual.

#### 4.1.3 Finalização

Para que o servidor possa concluir um pedido, ou seja, efetuar os cálculos dos tamanhos dos ficheiros, enviar a mensagem de conclusão para o cliente e libertar os recursos alocados ao pedido, o servidor usa a função `finish_process`, que recebe o *pid* do filho criado para executar as transformações do processo. Desta forma, é possível encontrar o processo no array de processos em execução.

Portanto, o servidor encarrega-se de libertar a quantidade de transformações reservadas para este processo, também liberta ao processo ao colocar `false` no `running` e, por fim, envia a resposta de conclusão ao cliente.

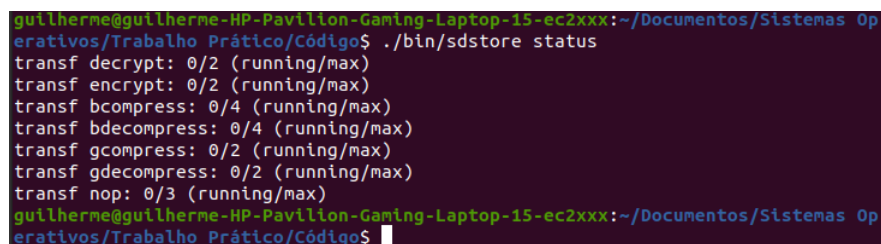
### 4.2 Programa Cliente - *Sdstore*

Um programa mais simples do que o do servidor, mas de extrema importância já que é o que recebe os pedidos dos clientes.

Os comandos que o cliente pode receber são de 2 tipos:

Para o caso de *status*, o *standard output* é usado pelo cliente para apresentar o estado do serviço ou o estado de processamento do pedido (`pending`, `processing`, `concluded`).

```
$ ./bin/sdstore status
```

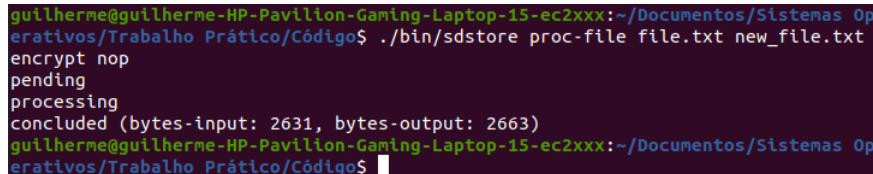


```
guilherme@guilherme-HP-Pavilion-Gaming-Laptop-15-ec2xxx:~/Documentos/Sistemas Operativos/Trabalho Prático/Código$ ./bin/sdstore status
transf decrypt: 0/2 (running/max)
transf encrypt: 0/2 (running/max)
transf bcompress: 0/4 (running/max)
transf bdecompress: 0/4 (running/max)
transf gcompress: 0/2 (running/max)
transf gdecompress: 0/2 (running/max)
transf nop: 0/3 (running/max)
guilherme@guilherme-HP-Pavilion-Gaming-Laptop-15-ec2xxx:~/Documentos/Sistemas Operativos/Trabalho Prático/Código$
```

Figura 1: Exemplo de resultado do comando *status* no cliente

Já quando o 1º argumento é "*proc-file*" a partir do *standart input* não podem conter o argumento `priority`, já que não implementámos a funcionalidade avançada de processos com prioridade (poderá ver mais pormenores no tópico "*Funcionalidades Avançadas*"). Assim, os comandos deverão ter sempre esta mesma estrutura:

```
$ ./bin/sdstore proc-file samples/file-a outputs/file-a-output bcompress
nop gcompress encrypt nop ...
$ ./bin/sdstore proc-file file.txt new_file.txt encrypt nop
(exemplo para teste)
```



```
guilherme@guilherme-HP-Pavilion-Gaming-Laptop-15-ec2xxx:~/Documentos/Sistemas Operativos/Trabalho Prático/Código$ ./bin/sdstore proc-file file.txt new_file.txt
encrypt nop
pending
processing
concluded (bytes-input: 2631, bytes-output: 2663)
guilherme@guilherme-HP-Pavilion-Gaming-Laptop-15-ec2xxx:~/Documentos/Sistemas Operativos/Trabalho Prático/Código$
```

Figura 2: Exemplo de resultado do comando *proc-file* no cliente

## 5 Estruturas de dados e Variáveis auxiliares

Durante a implementação do código do serviço, reparamos que seria necessário termos à disposição de variáveis globais e estruturas de dados que pudessem guardar dados sobre o estado dos processos que chegam ao servidor.

Assim, criamos as seguintes estruturas de dados e variáveis, enunciadas nos próximos tópicos:

### 5.1 Variáveis auxiliares (Mais importantes)

No código-fonte *sdstored.c*, podemos observar um conjunto de variáveis globais que guardam informações importantes para o funcionamento do servidor.

Apresentamos um resumo de cada uma delas nos seguintes pontos (as várias estruturas de dados aqui referenciadas que ainda não foram identificadas, sê-lo-ão nos subtópicos seguintes ao atual):

- **Transf transf[7]**: informações gerais sobre as 7 transformações em ficheiros possíveis. Como dito anteriormente, temos uma função que atribui um hashcode a cada transformação. Isto foi possível pois sabemos a quantidade exata de tipos de transformações existentes e, desta forma, não precisamos percorrer sempre todas para verificar se há transformações livres para executar pedidos ou até mesmo para verificar o nome.
- **Process processes[1024]**: dados dos processos a serem executados ao mesmo tempo (definimos que o servidor só pode executar no máximo 1024 processos ao mesmo tempo).

- **Process ready\_queue[4096]**: processos em fila-de-espera (pode-se guardar no máximo 4096 processos em fila-de-espera).
- **int ready\_queue\_total\_size; int process\_total\_size;** indicam, respetivamente, o número de processos em fila-de-espera e o número total de processos ATIVOS no servidor.

## 5.2 Request

Sempre que o programa *sdstore* (cliente) recebe um conjunto de argumentos a partir do terminal, a função **make\_request** do servidor transforma os argumentos recebidos na seguinte estrutura de dados:

---

```
typedef struct request {  
    int n_messages;  
    char message[64][512];  
    int pid;  
} Request;
```

---

Nela estão guardados os dados que o cliente precisa de enviar para o servidor, nomeadamente, o número de argumentos que o comando tem, as *strings* dos vários argumentos, bem como o *pid* do processo do cliente para que o servidor saiba para qual cliente responder. Caso o comando seja do tipo *proc-file*, **n\_messages** é superior a 1, enquanto que se for do tipo *status*, terá apenas um valor igual a 1. Isto além de facilitar na escrita do código também tornou-se mais eficiente visto que não é preciso utilizar *strtok* ou *strsep* para separar os argumentos, pois já vem separados por causa da escolha da matriz.

## 5.3 Reply

A estrutura *Reply* tem como objetivo fornecer ao cliente informações sobre o estado atual dos processos no servidor.

---

```
typedef struct reply {  
    int n_messages;  
    char message[64][512];  
    int status;  
} Reply;
```

---

Caso o valor da variável **status** seja igual a 0, deve-se fechar a FIFO que permite o envio de dados do servidor para o respetivo cliente. Assim não é possível receber mais informações do servidor. Ou seja, o pedido do cliente ao servidor foi devidamente concluído.

Por outro lado, se **status** for igual a 1, significa que o pedido ainda está a decorrer e o servidor enviará mensagens sobre o estado do pedido do cliente.



## 5.4 Transf

Esta estrutura não é aplicada a um processo em específico. Deve ser vista como uma estrutura auxiliar global que fornece informações sobre um tipo de transformação, como o nome, o número de transformações do tipo em causa que estão a ser executadas no momento atual; bem como, o número máximo de transformações deste tipo que o servidor pode executar ao mesmo tempo.

---

```
typedef struct transf {  
    char name[16];  
    int running;  
    int max;  
} Transf;
```

---

## 5.5 Process

Uma das estruturas mais importantes deste projeto, onde são guardadas todas as informações relevantes de um processo.

Uma estrutura *Process* contém o *pid* do cliente que enviou o request para executar o pedido; e o *pid* do processo-filho onde o processo está a ser executado. Pois é uma forma que conseguimos encontrar no array *process* o processo que já pode ser finalizado a partir da função que executa as transformações.

Para além disso, contém as *strings* dos *paths* dos ficheiros de *input* e de *output*, os booleanos **running** e **ready**, que indicam, respetivamente, se o processo está a ser executado pelo servidor, ou se ainda está à espera/em fila de espera.

Por fim, mas não menos importante, o número de TIPOS de transformações que o processo terá de realizar (**tp\_size**), bem como informações sobre a quantidade necessária de cada uma das transformações do pedido no array *tp*[7].

---

```
typedef struct process {  
    int client_pid;  
    int fork_pid;  
    TProcess tp[7];  
    int tp_size;  
    char transf_names[64][64];  
    char name_input[512];  
    char name_output[512];  
    int number_transfs;  
    bool running;  
    bool ready;  
} Process;
```

---

## 5.6 TProcess

Pertencente à estrutura de dados *Process*, a estrutura auxiliar *TProcess* fornece ao servidor informações sobre uma transformação em específico, nomeadamente sobre o nome da transformação e o número total de transformações que o processo tem de realizar deste tipo em específico. Também beneficia-se da função de hashcode.

Isto foi criado para facilitar a avaliação da disponibilidade das transformações para a tomada de decisão sobre executar o pedido ou mantê-lo em fila de espera. Achamos que seria mais fácil assim do que ter que percorrer o array da estrutura processo que guarda as transformações na ordem de execução visto que as transformações repetidas poderiam aparecer intercaladas com outras o que dificultaria a contagem e tornaria menos eficiente.

---

```
typedef struct transfs_process {  
    int n;  
    char name[16];  
} TProcess; // TP
```

---

## 6 Funcionalidades Avançadas

Decidimos não aplicar todas as funcionalidades avançadas propostas no enunciado. Aplicámos apenas as seguintes funcionalidades avançadas:

Quando um pedido termina, é reportado ao cliente o número de bytes do ficheiro recebido e o número de bytes produzidos no ficheiro final resultante de cada operação *proc-file*. Para isso, utilizámos a função `get_file_size`, que, recebendo o file descriptor do ficheiro em causa, devolve o número total bytes do ficheiro através da `lseek` ao redirecionar o offset para o final do ficheiro.

Se o servidor receber um sinal `SIGTERM`, via `< Ctrl + C >` ou `< Ctrl + 'barra' >`, deve terminar de forma graciosa. Portanto, para estes sinais há o um handler que percorre a fila dos processos pendentes enquanto tiver e, no final, faz `unlink` da pipe do cliente para o servidor de forma que o servidor não irá receber mais pedidos.

Para o nosso projeto, decidimos que não iríamos aplicar as operações *proc-file* com diferentes prioridades. Desta forma, todos os argumentos dos comandos recebidos pelo cliente têm todos a mesma prioridade e são executadas pela ordem com que estão escritos no comando.

## 7 Conclusão

Concluindo, o grupo pode afirmar que está satisfeito com o trabalho feito. Com este trabalho prático, foi possível desenvolver um projeto mais complexo e próximo da realidade, dentro da área da Engenharia de Sistemas Operativos. Também permitiu consolidar todos os conteúdos lecionados nas aulas práticas da unidade curricular de Sistemas Operativos. Particularmente, ficamos muito contentes com a implementação do conceito de hash mesmo que de uma forma simples e, também, com a fixação e limitação do tamanho dos pedidos e respostas.

Gostávamos de ter implementado estruturas mais eficientes para a fila, pois sabemos que percorre a fila toda desta forma mesmo que não tenha o total de transformações disponíveis para o processo. Talvez uma matriz ou mesmo uma minheap teria sido mais aconselhável. Porém optamos por seguir pelo mais seguro e, ainda assim, de certa forma eficiente devido à localidade.

Apesar de não termos implementado a funcionalidade avançada de prioridade de pedidos, podemos dizer que conseguimos implementar com sucesso o serviço de cópia eficiente de ficheiros e as funcionalidades que descrevemos ao longo do relatório.