

Universidade Do Minho
Escola de Engenharia

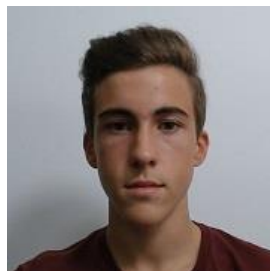
Universidade do Minho

SDStore: Armazenamento Eficiente e Seguro de Ficheiros
Sistemas Operativos
LCC 2º Semestre
Maio 2022

Grupo 7



Miguel Gonçalves
A90416



Paulo Costa
A87986



Rui Baptista
A87989

Introdução:

O intuito deste trabalho é implementar um serviço que permita aos utilizadores armazenar uma cópia dos seus ficheiros de forma segura e simultaneamente eficiente, sendo assim possível poupar espaço de disco. Assim, os utilizadores terão como opções a cifragem e a compressão dos ficheiros.

O utilizador irá escrever os comandos desejados no cliente, este vai enviar tais comandos ao servidor onde vão ser executados. Estes comandos podem ser tanto para processar e armazenar novos ficheiros como para recuperar conteúdo previamente guardado. É ainda possível consultar o estado atual das tarefas.

Estas funções serão explicadas detalhadamente no decorrer do relatório.

Estrutura e Decisões tomadas:

- **Cliente (sdstore.c)**

No cliente começamos por identificar o tipo do pedido do cliente, caso seja um **status** ou **proc-file** criados dois pipes nome para cada cliente que fazem a ligação com o servidor. Para tornar estes pipes únicos, utilizamos o pid do cliente. Desta forma, o *pipe_escrever* vai ter o nome “/tmp/wPidCliente” e serve para o enviar ao servidor o tipo do pedido do cliente. Em contrapartida, o *pipe_ler* cujo nome é “/tmp/rPidCliente” recebe a informação solicitada, que será escrita no stdout.

De forma que o servidor seja capaz de identificar a que cliente tem que mandar a informação, criamos um *pipePrincipal* no servidor. Este pipe serve apenas para os clientes enviarem o seu pid para o servidor identificar os seus respetivos pipes.

- **Servidor (sdstored.c)**

O servidor começa por abrir o primeiro argumento que recebe, que é a diretoria para o ficheiro de configuração, e retira deste ficheiro o número máximo possível para cada transformação, após isso guardamos o segundo argumento que é a diretoria para os executáveis das transformações.

De seguida, é criado o *pipePrincipal* que irá receber o pid dos clientes, para que o servidor possa identificar o *pipe_lercliente* que traz a indicação da tarefa a ser efetuada. Caso esta tarefa seja um pedido de verificação do estado, será chamada a função **status**, caso seja uma requisição de transformação de um ficheiro, será invocada a função **proc-file**. Este processo será repetido até que seja recebido um sinal de SIGINT ou SIGTERM que fechará de forma elegante o servidor. A seguir serão descritas as funções referidas:

- **Status:** Começa por criar um processo filho, que enviará através do *pipe_escrever* a informação do estado atual do programa ao cliente.

- **Proc-file:** No momento inicial verifica se o pedido pode ser executado utilizando a função `check_disponibilidadeMAX` e `check_disponibilidade`. Caso seja possível chamamos a função **monitor** que é responsável por abrir os ficheiros de input e output e fazer o encadeamento das transformações. Caso não seja possível, o pedido é adicionado a uma queue onde a sua posição depende no seu nível de prioridade. Caso o pedido não tenha prioridade é atribuído o valor de 1 (mais baixa).

Instruções:

O primeiro passo é compilar o programa usando o make fornecido, com o comando:

```
$ make
```

Agora com o programa compilado, estamos prontos a trabalhar nele, primeiro devemos ligar o servidor com o comando:

```
$/bin/sdstored etc/sdstored.conf bin/sdstore-transf
```

Com ele ligado podemos agora enviar comandos através do cliente, iremos então passar a explicar os diversos comandos disponíveis na aplicação.

Estruturas de dados:

Utilizamos duas *structs* de forma a guardar os dados, sendo uma delas a *process*, que irá guardar o PID do processo, o comando a executar, o pipe para escrita e a prioridade do pedido. Por outro lado, temos também a estrutura *queue* onde serão guardados os processos em espera, de forma a ser possível lidar com a questão da prioridade dos processos. Nesta iremos guardar variáveis de manipulação da *queue*, o tamanho da *queue* e um array com os processos.

```
// ----  
// Estruturas de dados  
// ----  
// Estrutura de um processo a ser guardado  
typedef struct process  
{  
    char *pid;           // PID do processo  
    char *comando;       // Comando a executar  
    int pipe_escrever;   // Pipe para escrever  
    int prioridade;      // Prioridade do processo  
} Process;  
  
// Queue onde serão armazenados os processos em espera  
typedef struct queue  
{  
    int first, last, size; // Primeiro, último e tamanho da queue  
    int capacity;          // Capacidade máxima da queue  
    Process *processes;    // Array de processos  
} Queue;
```

Figura 1- Structs

De forma a ser possível manipular as estruturas utilizámos as seguintes funções:

- **createQueue(int capacity)** : irá inicializar a queue
- **isFull(Queue *q)** : verifica se a queue está cheia
- **isEmpty(Queue *q)** : verifica se a queue está vazia
- **enqueue(Queue *q, Process p)** : adiciona um processo à queue
- **dequeue(Queue *q)** : remove um processo da queue
- **orderQueue(Queue *q)** : ordena a queue por prioridade
- **printQueue(Queue *q)** : imprime a queue

Comandos

Status

O comando para consultar as tarefas a serem efetuadas no momento é:

\$/bin/sdstore status

Desta forma é possível observar as tarefas a serem efetuadas, sendo ainda possível observar quais são passíveis de ser executadas e quais entrarão na fila de espera (queue), visto que cada transformação tem um número máximo de execuções simultâneas.

```
paulinhordc@MBP-de-Paulo S0-Project % ./bin/
sdstore status
Transf nop: 0/2 (Running/Max)
Transf bcompress: 0/3 (Running/Max)
Transf bdecompress: 0/3 (Running/Max)
Transf gcompress: 0/4 (Running/Max)
Transf gdecompress: 0/4 (Running/Max)
Transf encrypt: 0/2 (Running/Max)
Transf decrypt: 0/2 (Running/Max)
paulinhordc@MBP-de-Paulo S0-Project %
```

Figura 2 - Resultado Status

```
sdstored.conf
nop 2
bcompress 3
bdecompress 3
gcompress 4
gdecompress 4
encrypt 2
decrypt 2
```

Após o comando ser executado aparecerá, individualmente, uma linha com o nome da transformação seguido do número atual de execuções dessa mesma transformação e de seguida o seu valor máximo de execuções simultâneas.

Caso seja efetuado um pedido para executar uma transformação que já tenha excedido o seu limite, essa transformação entrará na queue, sendo executada quando for possível.

Com este comando vêm algumas funções, de forma a mantê-lo atualizado:

- **check_disponibilidade (char *command)**: irá retornar 1 caso existam filtros disponíveis para executar a transformação
- **updateSlots (char *arg)**: atualiza a informação sobre os filtros em uso, incrementando sempre que necessário
- **freeSlots (char *arg)**: atualiza a informação sobre os filtros em uso, decrementando quando o filtro deixar de ser usado.

Proc-File (sem Prioridade)

O comando para solicitar o processamento e armazenamento de um ficheiro é o seguinte:

```
$/bin/sdstore proc-file ficheiroInput ficheiroOutput transf
```

Para executar este comando é necessário fornecer alguns argumentos para além do comando em si, sendo eles a diretoria do ficheiro a ser processado (**ficheiroInput**), a diretoria onde será guardada a nova versão do ficheiro (**ficheiroOutput**) e um ou mais identificadores das transformações a aplicar (**transf**).

Nestes casos o nível de prioridade será definido como 0, por default.

Proc-File (com prioridade)

É possível priorizar a solicitação anterior executando o seguinte comando:

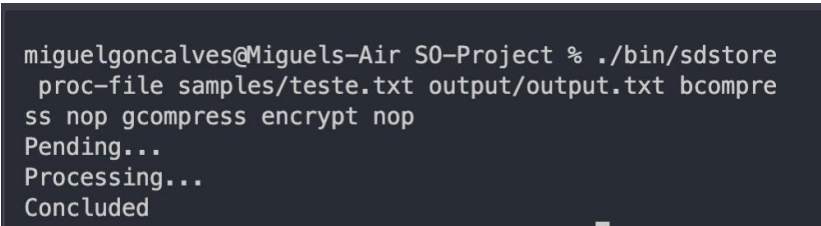
```
$/bin/sdstore proc-file -p X ficheiroInput ficheiroOutput transf
```

Neste caso é igualmente necessário fornecer tanto os caminhos para os ficheiros como as transformações desejadas. Para além disso é agora necessário escrever a *flag* de prioridade (**-p**) e ainda o seu nível de prioridade (**X**), sendo esta variável entre 0 e 5, inclusivé.

O sistema de prioridade será descrito com mais detalhe posteriormente no relatório.

Executar uma transformação

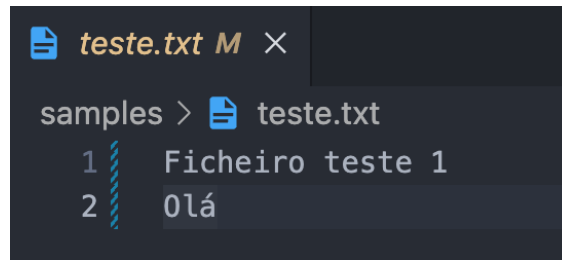
Como foi explicado na secção anterior (Comandos), e ilustrando com um pequeno exemplo, se executarmos na linha de comandos, algo como:



```
miguelgoncalves@Miguels-Air S0-Project % ./bin/sdstore  
proc-file samples/teste.txt output/output.txt bcompre  
ss nop gcompress encrypt nop  
Pending...  
Processing...  
Concluded
```

Figura 3- Execução proc-file 1

Esta opção vai executar, neste caso, as transformações “bcompress nop gcompress encrypt nop”, onde o ficheiro “input” está em *samples/teste.txt*.



```
teste.txt M ×
samples > teste.txt
1 Ficheiro teste 1
2 Olá
```

Figura 4- Conteúdo teste.txt

O ficheiro “output” vai ser guardado num ficheiro chamado *output.txt*.

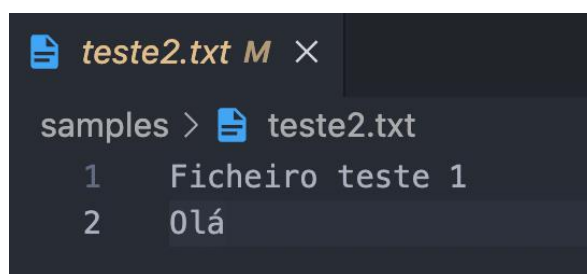
Como se verifica na *Figura 2* as transformações foram concluídas, passando pelos estados Pending e Processing, até chegarem ao Concluded.

Se posteriormente quisermos verificar se tudo foi concluído com sucesso, podemos, por exemplo, executar na linha de comandos as transformações contrárias, às anteriormente utilizadas e colocando agora com “input” o ficheiro *output.txt* e guardar o resultado, por exemplo, em *samples/teste2.txt*.

```
miguelgoncalves@Miguels-Air S0-Project % ./bin/sdstore proc-file output/output.txt samples/teste2.txt decrypt gdecompress nop bdecompress
Pending...
Processing...
Concluded
```

Figura 5- Execução proc-file 2

Agora, verificaremos se tudo correu como planeado verificando se o conteúdo do ficheiro *teste2.txt* é igual ao de *teste.txt*



```
teste2.txt M ×
samples > teste2.txt
1 Ficheiro teste 1
2 Olá
```

Figura 6- Conteúdo teste2.txt

Funcionalidades adicionais

- Sinal SIGTERM: (PAULO ALTERAR PRINT PARA OS DE BAIXO)

Para esta funcionalidade pediram-nos para que se fosse recebido o sinal SIGTERM, o servidor deveria terminar, mas com a subtilidade de que deveria deixar primeiro terminar os pedidos em processamento ou pendentes.

```
//Handler do sinal SIGTERM
void term_handler(int signum) {
    int status;
    pid_t pid;
    while((pid = waitpid(-1, &status, WNOHANG)) > 0) wait(NULL); //Termina todos os filhos.

    unlink("/tmp/main");
    if (unlink("/tmp/main") == -1) {
        perror("[tmp main] Erro ao eliminar ficheiro temporário");
        _exit(-1);
    }

    write(1, "\nA terminar servidor.\n", strlen("\nA terminar servidor.\n"));
    _exit(0);
}
```

Figura 7- Handler do sinal SIGTERM

Podemos verificar que a função espera que os filhos terminem, e posteriormente, apaga o pipe com nome main, para não aceitar novos pedidos, e transmite uma mensagem a informar que o servidor está a ser terminado.

```
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
Servidor iniciado com sucesso!
[DEBUG Comando Inicial] proc-file samples/teste.txt output/t.txt nop
[DEBUG Comando Fim] samples/teste.txt output/t.txt nop
[DEBUG input] samples/teste.txt
[DEBUG output] output/t.txt
^C
A terminar servidor.
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ child_handler inicio!
child_handler fim!
Queue size: 0
Queue first: 0
Filho fechado!

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 22)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$
```

Figura 8- Handler do sinal SIGTERM II

Podemos então verificar que esta funcionalidade foi conseguida com bastante sucesso.

• Prioridade

Nesta funcionalidade podemos definir a ordem pela qual queremos que uma Task avance, quando uma das transformações já encontra totalmente ocupada, como iremos demonstrar no exemplo seguinte.

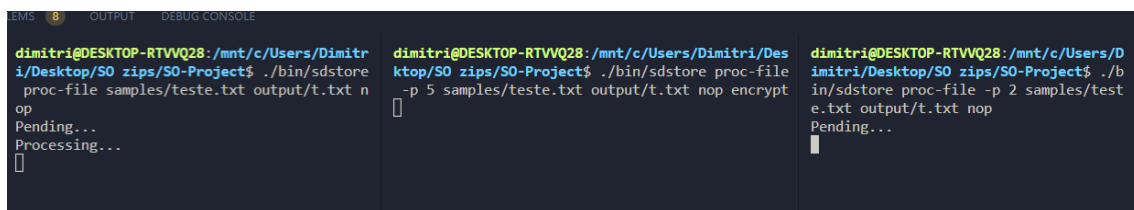
Começamos por atribuir o nível da prioridade, sendo 0 quando esta não existe (default) e um valor entre 0 e 5 que se encontra no 13º caractere:

```
// Verificar se o comando tem prioridade
if (comando[10] == '-' && comando[11] == 'p')
{
    hasPriority = 0;
    char *args = strdup(comando);
    strsep(&args, " ");
    auxComando = strsep(&args, "\\n");
    prio = atoi(&comando[13]);
}
```

Figura 9- Nível de prioridade

Vamos em seguida, demonstrar um caso prático para que seja mais fácil perceber o funcionamento desta funcionalidade.

Iniciamos então a primeira Task, ocupando assim por completo (neste exemplo) a transformação *nop*, obrigando a próxima Task com prioridade 2 a ficar pendente:



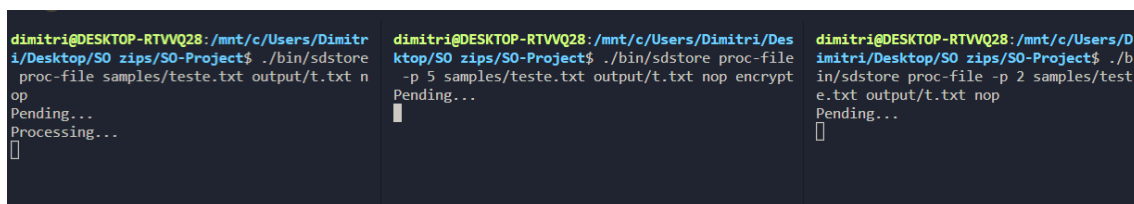
```
dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
op
Pending...
Processing...
█

dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file -p 5 samples/teste.txt output/t.txt nop encrypt
█

dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file -p 2 samples/teste.txt output/t.txt nop
Pending...
█
```

Figura 10 - Inicialização da primeira Task

De seguida, iniciamos a terceira Task com ordem de prioridade 5 (o mais elevado), passando assim para o início da *queue*, sendo idealmente inicializada logo que possível:



```
dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
op
Pending...
Processing...
█

dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file -p 5 samples/teste.txt output/t.txt nop encrypt
Pending...
█

dimitri@DESKTOP-RTVWQ28: /mnt/c/Users/Dimitri/Desktop/SO_zips/SO-Project$ ./bin/sdstore proc-file -p 2 samples/teste.txt output/t.txt nop
Pending...
█
```

Figura 11- Processos pendentes

Podemos observar que após a conclusão da Task inicial foi respeitada a prioridade, iniciando-se assim a Task com nível de prioridade 5:


```
LEMS 8 OUTPUT DEBUG CONSOLE

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 22)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 5 samples/teste.txt output/t.txt nop encrypt
Pending...
Processing...
█

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 2 samples/teste.txt output/t.txt nop
Pending...
█
```

Figura 12 - Início da segunda Task

Visto que a segunda Task acabou e a transformação *nop* se encontra novamente disponível, dá-se início à terceira task:

```
LEMS 8 OUTPUT DEBUG CONSOLE

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 22)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 5 samples/teste.txt output/t.txt nop encrypt
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 0)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 2 samples/teste.txt output/t.txt nop
Pending...
Processing...
█
```

Figura 13 - Conclusão da segunda Task

Por fim, é possível observar que as 3 Tasks foram concluídas com sucesso e pela ordem desejada:

```
LEMS 8 OUTPUT DEBUG CONSOLE

RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file samples/teste.txt output/t.txt nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 22)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 5 samples/teste.txt output/t.txt nop encrypt
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 0)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$ ./bin/sdstore proc-file -p 2 samples/teste.txt output/t.txt nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 22)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/S0-Project$
```

Figura 14 - Conclusão das Tasks

• Número de bytes recebidos e produzidos

Tal como era pedido, quando um pedido termina é reportado ao cliente o número de bytes que foram lidos do ficheiro de input e o número de bytes escritos no ficheiro de output. Para tal, abrimos primeiro os dois ficheiros mencionados, fazemos um read de byte a byte e incrementamos a variável. No fim escrevemos no stdout a informação pedida.

```
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ ./bin/sdstore proc-file -p 5 samples/test
e.txt output/t.txt bcompress nop
Pending...
Processing...
Concluded (bytes-input: 22, bytes-output: 70)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$
```

Figura 15- Número de bytes recebidos & produzidos

• Transformação impossível de realizar

Embora não tenha sido explicitamente pedido, achamos por bem impor esta restrição, no que tocava ao cliente tentar efetuar uma transformação que não tivesse espaço possível em algum dos slots de uma transformação em específico.

Como se pode verificar na figura a seguir apresentada:

```
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ ./bin/sdstore proc-file -p 2 samples/test
e.txt output/t.txt nop nop
Pending...
A transformação pedida excede algum dos limites estabelecidos do Servidor. Verifique o status!
Concluded (bytes-input: 22, bytes-output: 0)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$

dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$ ./bin/sdstore status
Transf nop: 0/1 (Running/Max)
Transf bcompress: 0/3 (Running/Max)
Transf bdecompress: 0/3 (Running/Max)
Transf gcompress: 0/4 (Running/Max)
Transf gdecompress: 0/4 (Running/Max)
Transf encrypt: 0/2 (Running/Max)
Transf decrypt: 0/2 (Running/Max)
dimitri@DESKTOP-RTVVQ28:/mnt/c/Users/Dimitri/Desktop/SO zips/SO-Project$
```

Figura 16- Transformação Impossível

Conclusão

Achámos este projeto de grande importância visto que abordou praticamente todos os conceitos lecionados ao longo deste semestre, onde tivemos de aplicá-los num desafio de maior escala.

Na nossa opinião todos os objetos foram alcançados, pois conseguimos cumprir todas as funcionalidades básicas e ainda praticamente todas as funcionalidades avançadas, isto é, o sinal SIGTERM e também as prioridades nas transformações. Foram intervenções que nos deram algum trabalho, mas no final valeram bastante a pena, pois tornaram o projeto não só mais eficiente como mais complexo, dando-nos assim a oportunidade de demonstrar ainda melhor os conhecimentos adquiridos.

Desta forma estamos contentes com o resultado final, no entanto temos noção que existe sempre espaço para melhorar.

Já fora deste contexto, queríamos apenas dar um agradecimento aos professores, mais em específico ao professor Vítor Fonte, que esteve sempre disponível para ajudar em certas dúvidas relativamente tanto ao trabalho prático como dos guiões realizados ao longo do semestre. E ainda ao professor Paulo Almeida, que também se demonstrou sempre disponível a atender as dúvidas existentes.