Sistemas Operativos

2º Licenciatura em Ciências da Computação

Grupo 20 - Trabalho Prático

SDStore: Relatório de Desenvolvimento

Alef Keuffer	Alexandre Baldé	Ivo Lima
(A91683)	(A70373)	(A90214)

19 de maio de 2022

Resumo

Neste relatório explicar-se-á a abordagem utilizada para construir o serviço de armazenamento seguro de ficheiros SDStore, utilizando a linguagem C e os conhecimentos práticos desenvolvidos nas UC de *Sistemas Operativos*.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este relatório contém a descrição do projeto realizado pelo Grupo 20 para o Trabalho Prático de Sistemas Operativos, para o ano letivo de 2021/2022.

Estrutura do Relatório

A estrutura do relatório é a seguinte:

- No capítulo 2 faz-se uma análise do comportamento dos programas cliente e servidor desenvolvidos para a aplicação SDStore.
- No capítulo explicam-se alguns aspetos mais técnicos e concretos da implementação.
- No capítulo 6 termina-se o relatório com as conclusões e o trabalho futuro.

Para mais informações sobre LATEX consultar o livro.

Capítulo 2

Arquitetura da aplicação

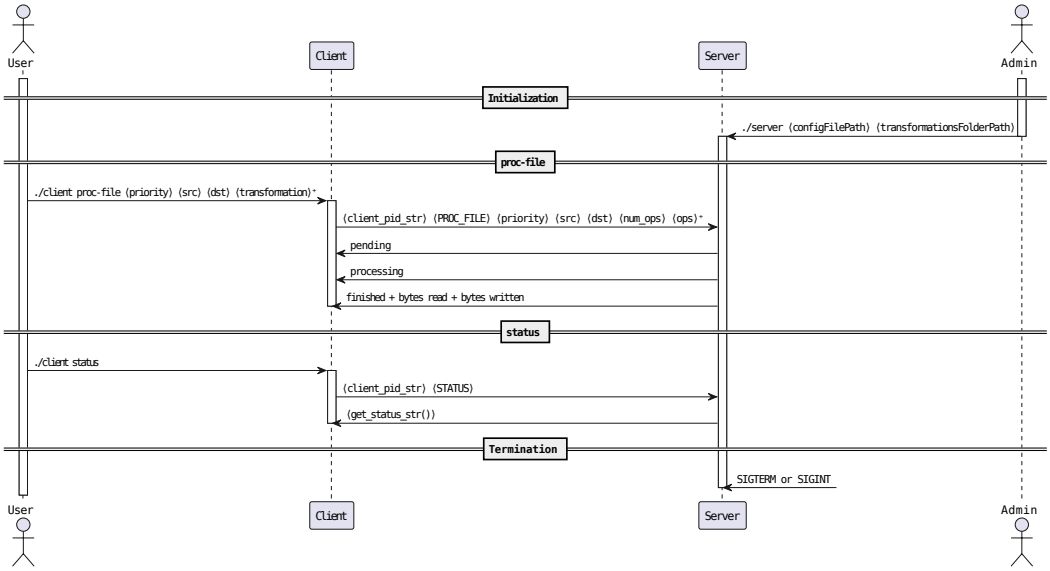


Figura 2.1: Interação entre Cliente e Servidor via terminal

2.1 Notas sobre Cliente, Servidor

- Servidor:
  - O servidor lê um ficheiro de configuração igual àquele descrito no enunciado, cuja informação ditará os limites de concorrência para cada transformação que pode correr.
  - O servidor também precisa receber como argumento a pasta onde se encontram os programas que efetuam as transformações. Ou seja, como pedia o enunciado: `./server etc/sdstored.conf bin/sdstore-transformations`.
  - A única forma de parar o servidor é enviar `SIGINT` ou `SIGTERM`, o que não terminará quaisquer processos em curso, ou que já tenham sido submetidos por clientes.
- Cliente:
  - A aplicação é composta por dois executáveis, `server` e `client` (que correspondem a `sdstore` e `sdstored`, resp.).
  - Tal como pedido no enunciado, o cliente pode submeter pedidos de transformação de ficheiros, com uma prioridade associada. A prioridade pode ser qualquer não negativo.
  - O cliente também pode, através do comando `./client status`, ver quais são as transformações atualmente em curso, e quais os limites de concorrência do servidor.

2.2 Comunicação entre Servidor e Cliente

- Servidor e clientes comunicam entre si através de `mkfifo()`s.
- Existe um só FIFO para comunicação de todos os clientes para o servidor, chamado `SERVER`
- Cada cliente envia o seu PID em cada pedido que faz, porque:
- Cada cliente tem um FIFO próprio para receber informação relativa ao seu pedido, identificado pelo seu PID.
- **Cada cliente**, ao comunicar com o servidor, cria uma estrutura de dados A com a informação relativa ao pedido (ou só `status` se for esse o caso), que depois envia ao servidor.
  O protocolo de comunicação entre cliente e servidor pode ser aproximado pela seguinte gramática:

$$
\begin{aligned}
\langle task\_message \rangle &::= \langle proc\_file \rangle \mid \langle status \rangle \mid \langle finished\_task \rangle \\
\langle proc\_file \rangle &::= \langle client\_pid\_str \rangle \text{ PROC\_FILE } \langle priority \rangle \langle src \rangle \langle dst \rangle \langle num\_ops \rangle \langle ops \rangle^{+} \\
\langle status \rangle &::= \langle client\_pid\_str \rangle \text{STATUS} \\
\langle finished\_task \rangle &::= \text{FINISHED\_TASK} \langle monitor\_pid\_str \rangle \\
\langle num\_ops \rangle &::= \langle int \rangle
\end{aligned}
$$

- O servidor não executa os pedidos que recebe dos clientes.
  Cria um **Monitor** por cada uma delas, que depois fica responsável por notificar o cliente e o servidor da sua terminação. Ver imagem 2.2.

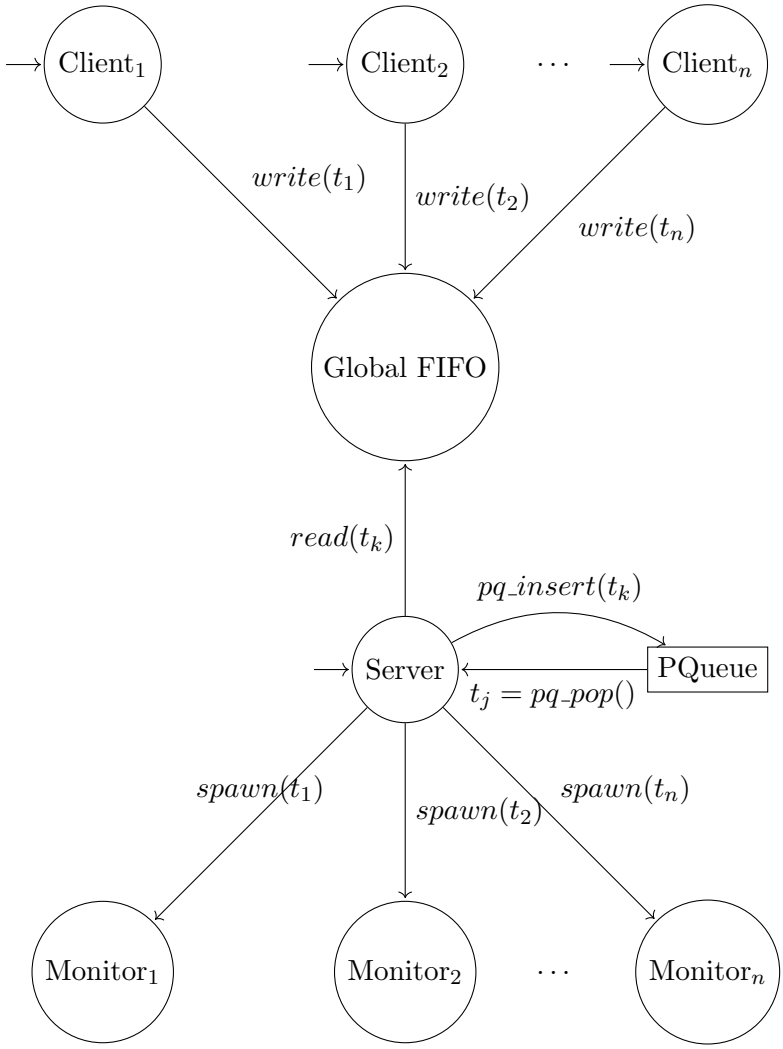


Figura 2.2: Arquitetura global da aplicação

Na figura 2.2, vê-se uma especificação informal do funcionamento global da aplicação.

- Vários clientes podem comunicar em simultâneo com o servidor
- O servidor lê os pedidos do FIFO, insere-os na sua "*priority queue*", e depois, de acordo com a sua lógica interna para controlar os limites de concorrência, cria monitores para as próximas tarefas, que ficam responsáveis pela "*pipeline*"de transformações.
- São os monitores que comunicam ao cliente a conclusão da tarefa, como referido anteriormente 2.2, assim como ao servidor, através de uma escrita no FIFO global de uma mensagem especial `FINISHED_TASK` 2.2. A figura seguinte ilustra melhor o funcionamento dos monitores.

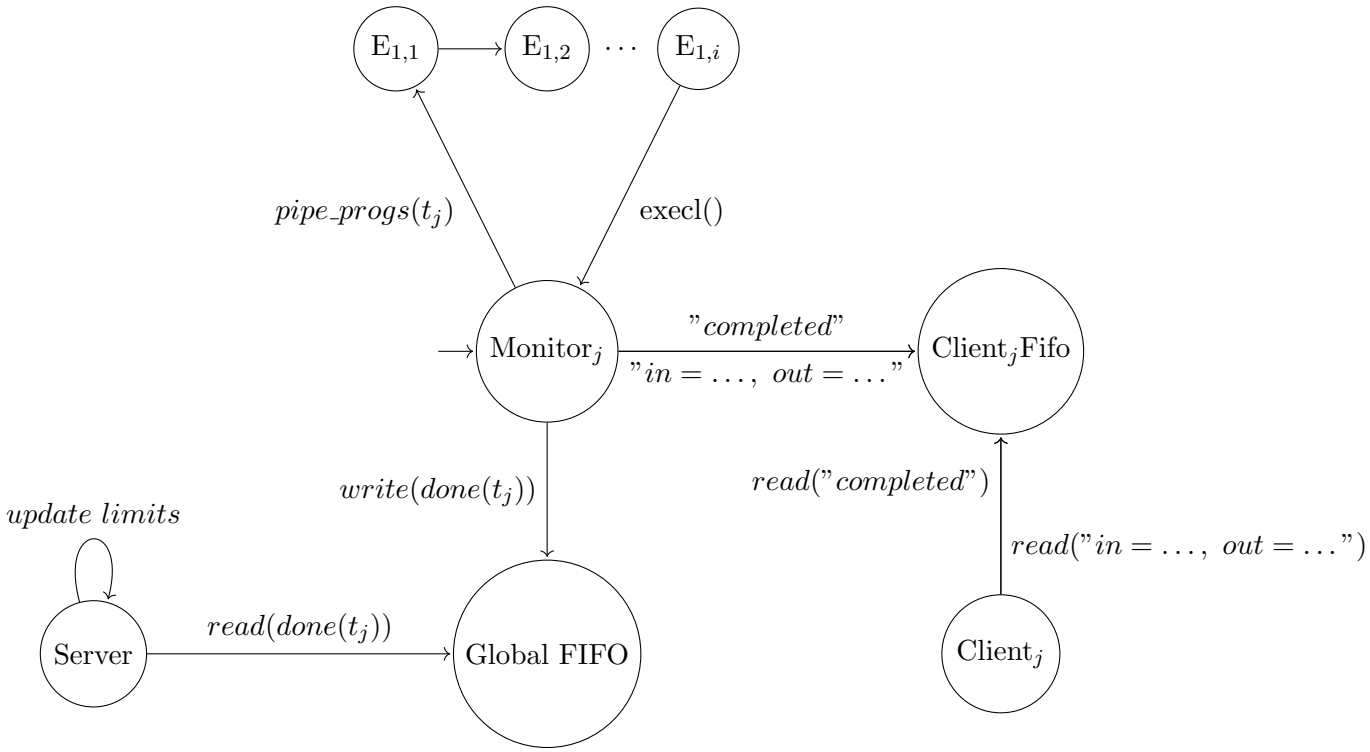


Figura 2.3: Relação entre Monitor, Cliente e Servidor

2.3 Detalhes sobre o servidor

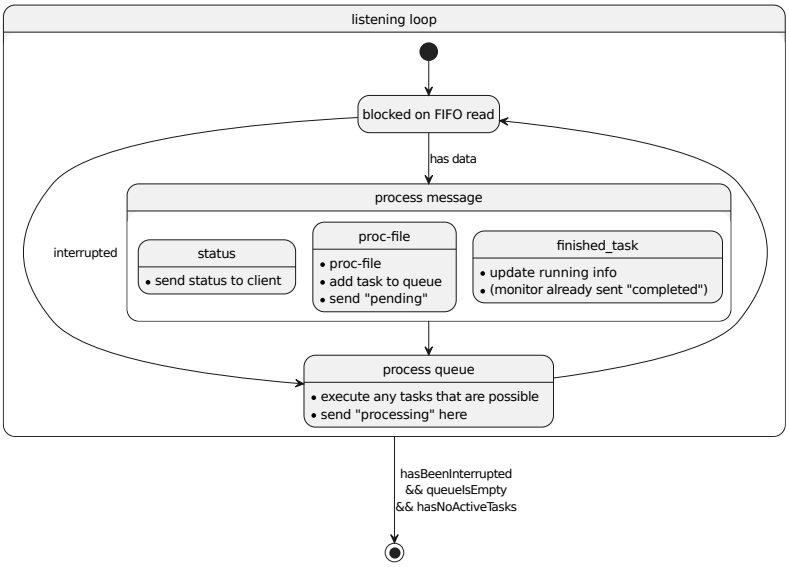
- O servidor faz, essencialmente, um ciclo while(1) read(fifo);.
- Durante o desenvolvimento do projeto, consideraram-se alternativas que fariam os monitores enviar sinais ao servidor aquando do término da sua execução, para ser poder fazer waitpid (...) , colher o processo monitor, e libertar os filtros em uso.

  No entanto, optou-se contra esta solução devido aos problemas de não-determinismo envolvidos no tratamento de sinais e.g. dois processos terminam exatamente ao mesmo tempo, só haverá um desses sinais na fila do servidor, só se fará wait uma vez, etc.
- O servidor usa uma "*priority queue*"[1] para guardar os pedidos que lê do FIFO global
- Após cada monitor terminar e escrever no FIFO global que já o fez, o servidor lê essa mensagem do FIFO e atualiza depois os limites de concorrência de cada tipo de transformação.
- O servidor obtém a próxima tarefa a executar através de pqueue_peek(). Se a próxima tarefa não puder ser executada, nunca se fazem esperas ativas: o servidor regressa à leitura do FIFO (que, recorde-se, é bloqueante), para esperar ou por novos pedidos de clientes, ou por monitores que terminam.

[1] implementação disponível em `https://github.com/vy/libpqueue`

A imagem seguinte tem mais detalhes sobre a lógica interna do servidor.

Figura 2.4: Funcionamento interno do Servidor



Capítulo 3

Análise e Especificação

3.1 Descrição informal do problema

3.2 Especificação do Requisitos

3.2.1 Dados

3.2.2 Pedidos

3.2.3 Relações

Capítulo 4

Concepção/desenho da Resolução

4.1 Estruturas de Dados

4.2 Algoritmos

Capítulo 5

Codificação e Testes

5.1 Alternativas, Decisões e Problemas de Implementação

5.2 Testes realizados e Resultados

Mostram-se a seguir alguns testes feitos (valores introduzidos) e os respectivos resultados obtidos:

Capítulo 6

Conclusão

Síntese do Documento [**?**, **?**].
Estado final do projecto; Análise crítica dos resultados [**?**].
Trabalho futuro.

Apêndice A

Excertos de Código Utilizado no Projeto

Listing A.1: Estrutura para pedidos de processamento de ficheiros

```
1    typedef struct task_t {
2      size_t pos; //private
3      pqueue_pri_t pri;
4      char *client_pid_str;
5      char *src;
6      char *dst;
7      char *ops;
8      int num_ops;
9      pid_t monitor;
10     char ops_totals[NUMBER_OF_TRANSFORMATIONS];
11   } task_t;
```

Lista-se a seguir UM TEXTO (COM O COMANDO VERBATIN)

```
aqui deve aparecer o código do programa,
tal como está formato no ficheiro-fonte "darius.java"
um pouco de matematica $\$$
caso indesejável $\varepsilon$
```

Listing A.2: Exemplo de uma Listagem

```
1      ou entao aparecer aqui neste sitio um pouco de matematica $
2      como alternativa ao anterior.
3      e aqui mais um teste ε
```