SDStore

Sistemas Operativos 2021/2022
Universidade do Minho

Luís Pereira	Rui Oliveira	Tiago Pereira
A96681	A95254	A95104

28 de maio de 2022

1 Introdução

O presente relatório refere-se ao trabalho prático proposto na unidade curricular de Sistemas Operativos na Licenciatura em Engenharia Informática da Escola de Engenharia da Universidade do Minho no ano de 2022.

O objetivo do trabalho consiste no desenvolvimento de uma aplicação cliente/servidor cujo objetivo consiste em processar ficheiros, aplicando-lhes sucessivas transformações, como, por exemplo, encriptação / desencriptação e compressão / descompressão.

A seguir, são expostas às decisões tomadas pelo grupo, bem como o porquê das mesmas e as suas vantagens e eventuais desvantagens. Também é explicada a implementação de forma breve, aprofundamento apenas as partes que o grupo considera as mais relevantes para a compreensão do funcionamento do *software* desenvolvido. Finalmente, é analisado o desempenho da solução em diferentes cenários, assim como potenciais entraves à sua melhoria.

2 Arquitetura

Como mencionado anteriormente, a aplicação segue o modelo cliente/servidor, pelo que existem dois programas a desenhar: o servidor, responsável pelo processamento dos pedidos; e o cliente, que envia esses mesmos pedidos.

2.1 Servidor

O servidor é constituído essencialmente por três componentes, cada uma correspondendo a um processo filho do processo principal do servidor. O fluxo dos pedidos pelo servidor é apresentado de forma genérica através da figura 1.

De seguida explica-se em detalhe, de forma agnóstica relativamente à sua implementação, como cada um destes componentes funciona.

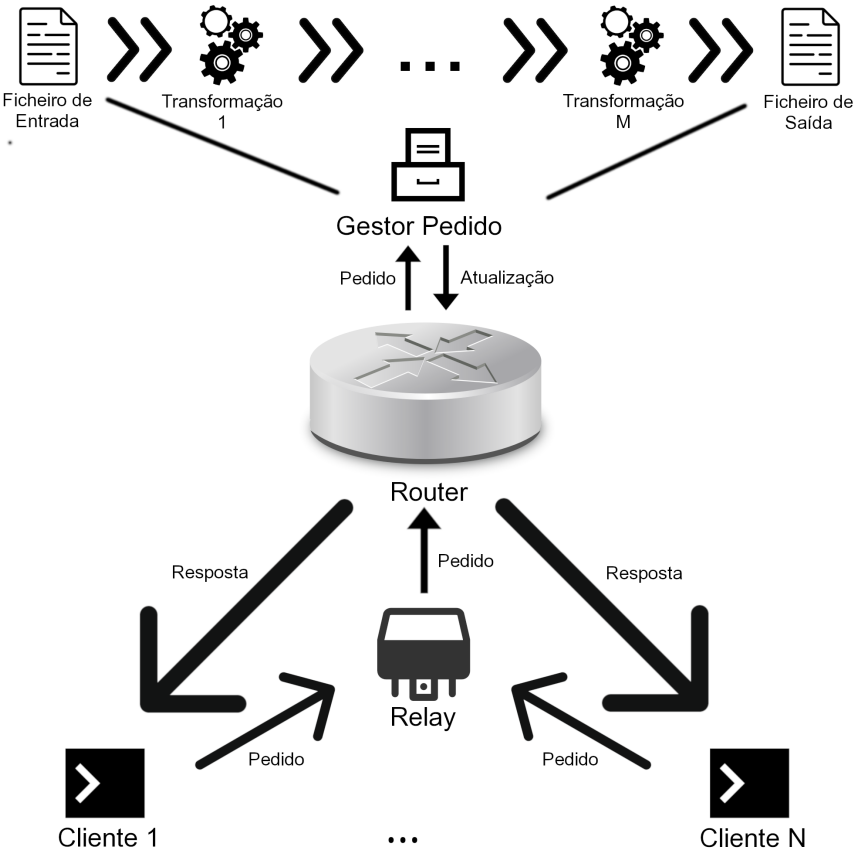

Figura 1: Esquema da arquitetura do servidor

2.1.1 Relay

O *relay* é responsável por receber os pedidos diretamente dos clientes e encaminhá-los para o *router*. A criação deste componente deveu-se à necessidade de fechar o canal de leitura do servidor, mas permitindo na mesma a leitura de atualizações relativamente ao estado dos pedidos e transformações ainda a correr. A maneira mais fácil de resolver este problema foi, exatamente, a separação entre o canal de leitura direto dos clientes e o router. Assim, quando o servidor fecha o seu canal de leitura, o relay para de executar, mas o canal de leitura de atualizações do router continua ativo.

2.1.2 Router

O *router* é o principal componente do servidor. A função deste consiste em processar todos os pedidos dos clientes, colocá-los em fila de espera e mandar

executar os pedidos assim que houver disponibilidade de recursos, isto é, houver instâncias de transformações livres para o pedido ocupar.

Assim sendo, o *router* recebe, de uma forma genérica, atualizações vindas quer do *relay* quer dos diferentes gestores de pedidos (que são seus subprocessos). Atualizações podem corresponder ao fim de uma instância de transformação, pelo que passa a existir mais uma disponível para executar os próximos pedidos; ao fim da execução de um dado pedido, pelo que o cliente é notificado da conclusão do seu pedido; de que o servidor vai encerrar, pelo que o *router* deve terminar mal a sua fila de espera fique vazia[1];ou, finalmente, à chegada de um novo pedido de um cliente. Por sua vez, os pedidos dos clientes podem tratar-se de pedidos para processar um ficheiro, que são colocados em fila de espera (cf. 2.1.3), ou pedidos de *status*, aos quais a resposta é imediata.

Após cada atualização, o *router* procura o pedido com maior prioridade que pode ser executado (cf. 2.1.3). Caso não exista, volta a escutar por atualizações. Caso exista, é criado um gestor do pedido - um subprocesso do router responsável pela execução do pedido dado - e o pedido sai da lista de espera.

2.1.3 Ordenação de Pedidos

Um dos requisitos a cumprir é a implementação de pedidos com prioridades diferentes, logo não basta processar os pedidos pela sua ordem de chegada (*FIFO*). Há, por isso, a necessidade de uma estrutura de dados que permita armazenar os pedidos que chegam em função da sua prioridade.

Mais concretamente, a prioridade de um pedido é definida por dois valores: o primeiro corresponde ao valor numérico da sua prioridade (um inteiro entre 0 e 5, 5 é o mais prioritário), o segundo corresponde ao tempo de chegada ao servidor, cuja função é desempatar pedidos com o mesmo valor de prioridade - entre estes, o que chegar primeiro é atendido primeiro.

Outro aspeto a considerar é o facto de haver pedidos que são independentes entre si. Por exemplo, se em espera no servidor estiverem os pedidos

```
proc-file 5 ... bcompress nop
```

```
proc-file 4 ... encrypt
```

e houver apenas uma instância da transformação encrypt disponível, este pedido deve ser processado, dado que isso não influenciará o tempo de espera do pedido de maior prioridade, visto que estes não partilham transformações.

Por outro lado, se em fila de espera estiverem os pedidos

```
proc-file 5 ... encrypt encrypt
```

```
proc-file 4 ... encrypt
```

e só houver uma instância da transformação encrypt disponível, apesar de ser possível executar o pedido de prioridade 4, isso atrasaria o pedido mais prioritário, pelo que o servidor deve esperar que outra instância fique disponível.

[1]De notar que, quando o *router* recebe esta atualização, é garantido que não irá receber mais pedidos de clientes.

2.1.4 Gestor de Pedido

Quando há um pedido para ser executado, o *router* cria um sub-processo, o gestor do pedido, cuja função é criar todas as instâncias das transformações e fazê-las comunicar entre si. Depois da configuração feita, o gestor espera que todos os seus processos filhos terminem. Sempre que um termina, o gestor envia uma atualização ao *router* a informar da disponibilidade de mais uma instância dessa transformação.

Quando todas as transformações acabam, e o pedido está cumprido, o gestor notifica o *router* do mesmo, para que o cliente seja notificado, antes de terminar a sua execução.

2.2 Cliente

O cliente é um processo completamente independente do servidor - aliás é possível vários clientes estarem ligados ao servidor simultaneamente. O cliente deve receber o seu pedido através dos seus argumentos, processá-los para o formato utilizado pelo servidor, criar um canal de comunicação (*FIFO*) para o qual o servidor deve enviar a resposta, e escutar esse mesmo canal até o pedido ser terminado, imprimindo a resposta do servidor para o *standard output*.

3 Implementação

Com a arquitetura da aplicação detalhada, são apresentados de seguida os detalhes de implementação considerados relevantes para a compreensão do funcionamento do projeto.

A filosofia da implementação foi tornar o programa o mais genérico possível, isto é, tentar que este não esteja limitado apenas às restrições do enunciado (nomeadamente no número e nome dos programas que é possível executar), ou a outras limitações, tais como o tamanho dos caminhos e nomes de ficheiros ao utilizar strings de comprimento fixo ao invés de comprimento variável. Além disso, procurou desenvolver-se uma aplicação com um grau de escalonamento relativamente alto, ou seja, que consiga suportar um elevado número de pedidos simultaneamente.

Isto levou a desafios adicionais de desenvolvimento, nomeadamente a garantia de que o programa não tem fugas de memória, o que foi verificado através da ferramenta Valgrind. Atualmente, o programa não apresenta qualquer fuga de memória.

3.1 Pedidos

Para uniformizar a implementação de pedidos entre o servidor e o cliente, e de forma a facilitar o processamento do pedido no lado do servidor, foi implementada a estrutura Request que armazena a informação de um pedido. A definição da estrutura é a seguinte:

```
typedef struct request {
    RequestType type; ///< The type of the request
    int priority; ///< The priority of the request (from 0 to 5)
    char* sender;   ///< The name of the input fifo of the client
    PipeWritter senderWriter; ///< The writter to the client
    char* inputFile; ///< The name of the input file
    char* outputFile; ///< The name of the output file
    int operationCount; ///< The number of operations requested
    char** operations; ///< The request operations
    int timeOfArrival; ///< Time of arrival in the server
} REQUEST, * Request;
```

Dos atributos da estrutura, de destacar que o tipo do senderWriter será explicado com detalhe em 3.4. O type do pedido pode ser ou STATUS ou PROC_FILE.

O timeOfArrival é preenchido pelo servidor à chegada do pedido, correspondendo à ordem de chegada ao servidor (0 corresponde ao primeiro pedido a chegar, 1 ao segundo, .etc).

3.2 Ordenação de Pedidos

A partir do referido em 2.1.3, é possível inferir a seguinte propriedade: *Um pedido pode ser satisfeito se e só se for o mais prioritário para cada uma das transformações que utiliza.* Dito por outras palavras, um pedido pode ser executado se não houver nenhum pedido com prioridade superior que utilize uma transformação em comum com este.

Esta observação motiva a criação de diferentes filas de espera, uma para cada tipo de transformação suportada pelo servidor. Para implementar uma fila de espera foi utilizada uma *max heap*, que permite inserção e remoção de pedidos em tempo logarítmico, bem como descobrir o elemento com mais prioridade em tempo constante (uma vez que a comparação de pedidos é feita em tempo constante). A estrutura que corresponde à implementação de uma fila com prioridade é a PQueue. O conjunto de todas as filas de espera e as operações de inserção global de um pedido e procura do próximo a executar são tratadas através da estrutura RequestSorter pelas funções enqueue e nextInLine.

Quando um pedido chega para ser colocado em espera, este é adicionado às filas de espera de todas as transformações que utiliza. [2] Isto é feito em $O(Tlog(P))$, onde P é o número de pedidos em espera no servidor, e T o número de transformações suportadas pelo servidor: assumindo este valor como sendo constante, a complexidade de inserção é apenas $O(log(P))$.

Quando se pretende descobrir qual o pedido a executar a seguir, são percorridas todas as filas de espera à procura do elemento em primeiro lugar nas mesmas.

[2] Um cuidado a ter com esta implementação é a possibilidade de um pedido utilizar duas vezes a mesma transformação, o que implica a necessidade de fazer mais verificações para evitar a duplicação do pedido numa fila de espera.

De seguida, calcula-se qual destes (se algum) é o que deve ser executado de seguida. Inicialmente considera-se o resultado nulo (NULL). Percorre-se todos os elementos que estão no topo das filas de espera. Para cada pedido, verifica-se se este corresponde também ao topo das filas de espera das transformações que usa. Caso afirmativo, este pedido passa a ser o a executar. Caso haja mais do que um pedido que possa ser executado, é retornado o mais prioritário. O pedido a executar é, por fim, eliminado de todas as filas de espera a que pertence. Isto executa em $O(Tlog(P) + P^2)$ - o que, assumindo T como constante, resume-se a $O(P^2)$. Como este cálculo é efetuado para cada atualização, mais concretamente cada pedido e cada nova instância disponível, é garantido que nunca é possível executar mais do que um pedido após cada atualização.

3.3 Updates

Da mesma forma que se criou o Request, foi necessário criar uma estrutura que definisse as atualizações que os gestores de pedidos enviam ao *router*. Esta estrutura designa-se por Update e tem uma declaração bastante mais simples que a de um pedido.

De forma geral, uma atualização tem um tipo, que pode ser um pedido vindo de um cliente, o fim de execução de um pedido,o fim de uma transformação, ou, finalmente, que o servidor vai encerrar. Os tipos são declarados na seguinte enumeração

```
/**
 * @brief The different types of #Update
 *
 */
typedef enum updateType
{
    U_REQUEST, ///< Incoming #Request
    U_REQUEST_FINISHED, ///< #Request has been executed
    U_FINISHED_OP, ///< A transformation has finished
    U_SERVER_DISCONECTED ///< The server will terminate
} UpdateType;
```

Além do tipo, um Update leva também informação sobre o pedido a que se refere, ou a que operação se refere. As operações são identificadas por um número inteiro, como explicado em 3.6.

3.4 Comunicação entre processos

De forma a otimizar as operações de leitura e escrita em pipes, foram criados os tipos PipeReader e PipeWritter, que permitem a utilização de buffers de forma transparente às suas chamadas de funções. Estes tipos suportam a leitura e escrita, respetivamente, de um número fixo de bytes (inteiros, por exemplo) ou de strings terminadas por um caracter nulo. A única diferença notável entre

as duas é que o PipeWritter apenas escreve o seu conteúdo quando chamada a função específica de flush, enquanto que a leitura do PipeReader para o buffer pode ser feita quando necessário sem perturbar a comunicação. Como esperado, caso não haja informação a ler nem no buffer nem no pipe, a operação bloqueia até nova informação ser disponibilizada.

As funções de leitura e escrita podem depois ser chamadas por funções de leitura e escrita de Requests e Updates, facilitando incomensuravelmente a comunicação entre os diferentes processos ao mudar o foco de dados primitivos para objetos. Isto permite diminuir em muito a reutilização de código e mantém o projeto modular e a comunicação elegante.

3.5 Logging

Para permitir ao servidor dar algum feedback sobre a sua execução, nomeadamente a chegada e processamento de pedidos e eventuais erros que possam acontecer, foi desenvolvida uma forma de imprimir mensagens padrão para o *standard output* e *standard error*. Existem diferentes níveis de severidade das mensagens (informativas, avisos, erros e erros fatais). As diferentes mensagens foram definidas com uma X macro no ficheiro logging.h. Para imprimi-las, é necessário invocar a função printMessage.

3.6 Configuração do Servidor

A configuração do servidor deve ser carregada a partir de um ficheiro fornecido como argumento para o servidor. Ao ler esse ficheiro, o servidor guarda a informação numa estrutura Config. Essa estrutura guarda:

- O número de transformações: corresponde ao número de linhas do ficheiro de configuração. Deste modo é possível criar uma configuração com um número praticamente ilimitado de transformações
- Nomes das transformações: são guardados num array dinâmico de strings de comprimento variável, o que aumenta a flexibilidade da configuração
- Número máximo de instâncias de cada transformação: um array de inteiros, em que a posição i corresponde ao número máximo de instâncias da transformação cujo nome é o i-ésimo elemento do array de nomes

Seguindo esta lógica, podemos criar um índice de transformações. Mais concretamente, o índice de uma transformação é a posição nos arrays de nome e número máximo de instâncias dessa mesma transformação. Assim sendo, é possível referir-se a operações usando apenas um inteiro, o que diminui a quantidade de dados a transmitir.

3.7 Pedidos de *Status*

Ao chegar ao *router*, os pedidos de *status* não são postos em fila de espera, pelo contrário, são respondidos imediatamente.

Para saber as instâncias disponíveis de cada transformação, o *router* armazena um array com esse número para cada transformação, que é atualizado sempre que um pedido é executado ou uma operação termina. Para saber que pedidos existem ainda no servidor, é guardado um array dinâmico de pedidos, no qual é adicionado um pedido sempre que este chega ao servidor. Quando o pedido termina a sua execução, o valor correspondente no array é colocado a nulo. Deste modo, para obter todos os pedidos, basta iterar sobre esse mesmo array e filtrar os valores não nulos.

Com esta informação constrói-se a *string* que é enviada de seguida ao cliente. Estes pedidos são respondidos em $O(P)$, onde P é o número de pedidos no servidor. Uma otimização futura seria eliminar a necessidade de filtrar os elementos não nulos, diminuindo o número de operações a realizar.

3.8 Fim Gracioso do Servidor

O fim gracioso do servidor é feito a partir do *relay*. Este no início inicia o *router* como sendo um processo independente para os sinais não se propagarem, e depois, antes de se desligar envia ao *router* um *update* a dizer para se desligar quando acabar todos os seus pedidos. No final fecha o *FIFO* e o *pipe* de comunicação ao *router* deixando de poder receber mais pedidos.

3.9 Cliente

A implementação do cliente é relativamente simples: o programa lê os seus argumentos, converte-os para um Request e envia-o para o servidor. Entretanto, abre um *FIFO* para ler as respostas do servidor. Quando esse *FIFO* fechar, significa que o pedido terminou de executar e o cliente termina normalmente.

4 Testagem e Avaliação de Desempenho

4.1 Software Auxiliar Desenvolvido

Para executar testes sobre o desempenho da aplicação foi construidos *scripts bash* para fazer execução de um pedido igual ao *SDStore* utilizando *piping*, criar ficheiros com informação aleatória de certos tamanhos e também uma forma de cronometrar o tempo que demora a execução de cada pedido. Para além disso, foi criado uma aplicação em *python* para fazer grandes volumes de testes automaticamente de dimensões variáveis e guardar essa informação num ficheiro *.json* para depois ser possível a geração de gráficos com a informação.

4.2 Resultados

Utilizando a sequencia de operações *bcompress nop gcompress encrypt nop* e ficheiros de tamanho 2^n bytes, $0 \leq n < 30$ obtemos os seguinte tempos:

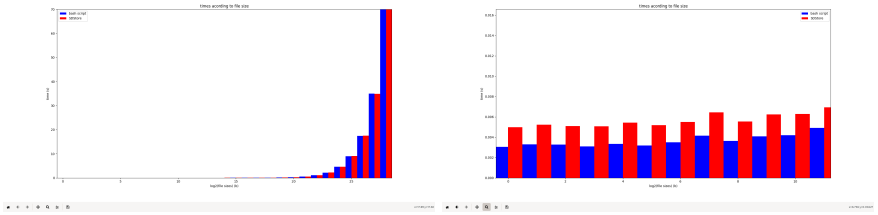

Figura 2: utilizar pipes no terminal vs sdstored

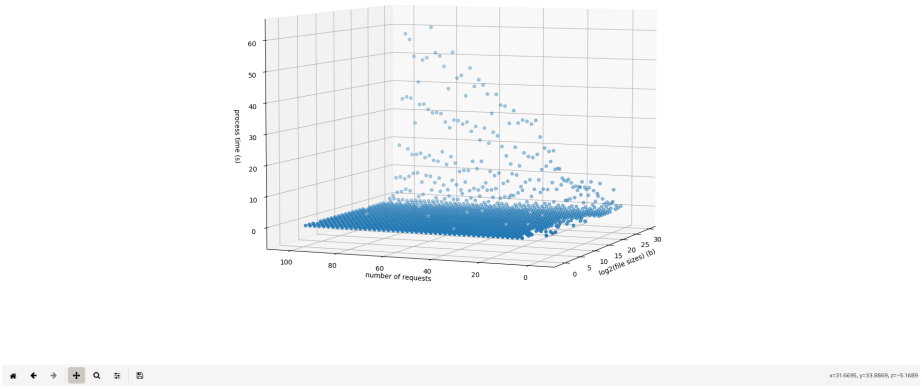

Figura 3: tempo de pedidos em paralelo para diferentes tamanhos de ficheiros

4.3 Análise dos Resultados

Através da figura 2 podemos observar que existe um pequeno tempo extra que o *SDStore* demora em relação à *bash* o qual é constante e corresponde a cerca de 0.00211 segundos, no entanto, para ficheiros de tamanho bastante elevado este tempo é desprezável visto que representa menos que 0.2% para operações que demorem mais que um segundo.

Através da figura 3 podemos observar que o tempo de processamento aumenta linearmente com o número de pedidos demonstrando que o que está a aumentar é o tempo de espera na fila e que não existe nenhum aumento de tempo devido ao tamanho do ficheiro de entrada no *SDStore*.

5 Conclusão

Deste modo, e para concluir, o grupo considera ter cumprido todos os requisitos impostos, implementando uma aplicação cliente/servidor através de comunicação entre processos não só fiável como também com um bom desempenho, permitindo aplicar transformações a ficheiros de forma concorrente.

Como em qualquer projeto, existem aspetos que podiam ter sido melhorados. Um desses aspetos seria a implementação dos pedidos de estado, que roda em tempo linear no número de pedidos; pelo que deverá ser possível implementar esses pedidos de forma mais eficientes. No entanto, a solução atual não apresenta problemas de desempenho. Outra alteração seria a utilização de expressões regulares para processamento dos pedidos no lado do cliente.

Assim sendo, o grupo considera o seu projeto como tendo sido bem sucedido.