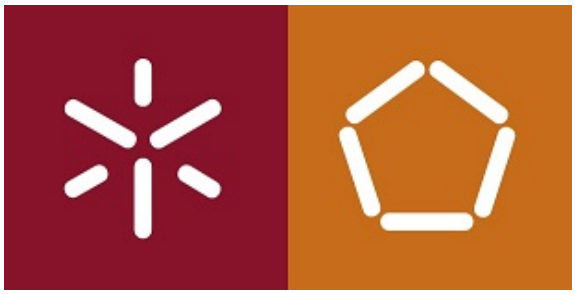

Universidade do Minho
Departamento de Informática

Aurras - Aplicação de filtros

Sistemas Operativos

Grupo 70

17 de Junho, 2021


Henrique Costa
(a93325)


Matilde Bravo
(a93246)


Mariana Rodrigues
(a93229)

Índice

1 Introdução **4**

2 Cliente **5**
2.1 Introdução 5
2.2 Request 6
2.3 Handshake 7
2.4 Transform 8

3 Servidor **9**
3.1 Introdução 9
3.2 Divisão do servidor 10
3.2.1 Processo da Queue 10
3.2.2 Processo das Execuções 10

4 Extensibilidade da aplicação **11**

5 Conclusão **12**

A Diagrama de Classes **13**

Lista de Figuras

2.1 Diagrama Cliente-Servidor 5
2.2 Diagrama Clientes-Servidor 5
2.3 Diagrama Servidor-Clientes 5
2.4 Excerto da struct Request 6
2.5 Excerto de um Request do tipo HANDSHAKE 7
2.6 Excerto de um Request do tipo TRANSFORM 8
2.7 Excerto de uma Reply 8

3.1 Pípes anónimos utilizados 9
3.2 Excerto de código da struct catalogo_filtros e filtros 9
3.3 Excerto de código da struct queue 10

1. Introdução

Este trabalho foi realizado no âmbito da unidade curricular Sistemas Operativos ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho, e teve como objectivo a implementação de um programa Cliente-Servidor para aplicação de filtros a áudios. Para tal, utilizamos conceitos lecionados em aula, como **pipes anônimos**, **pipes com nome**, **sinais**, **fork()**, **exec()**, dentre outros.

2. Cliente

2.1 Introdução

O código **cliente** do programa baseia-se em 2 conceitos: (1) envio de pedidos (**Request**) para o servidor e, (2) recebimento de respostas do servidor (**Reply**).

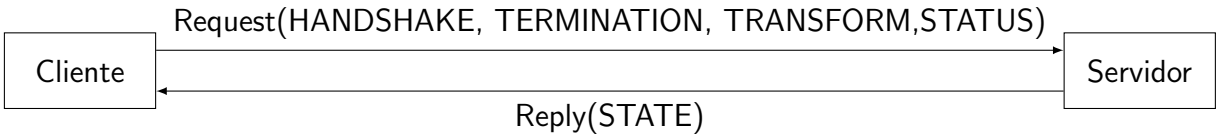

Figura 2.1: Diagrama Cliente-Servidor

O envio de pedidos para o servidor ocorre através de um **pipe com nome** chamado "client_to_server". Tal pipe é comum a todos os clientes que se conectam ao servidor. Por outro lado, o recebimento de algo vindo do servidor depende do cliente, isto é, é criado um pipe particular no sentido servidor-cliente. Para cada cliente que se conecta ao servidor, o pipe de recebimento de respotas vindo do servidor terá o nome: tubo_[pid_cliente]. Isto só é possível graças ao envio do **pid** do cliente através de um **Request**. Assim, quando o servidor tiver de informar algo à um cliente específico, basta escrever para o pipe que será initulado como a concatenação de "tubo_"+ Request.pid.

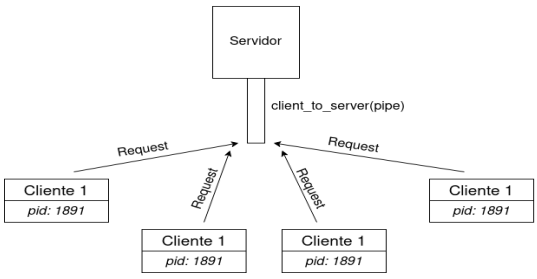

Figura 2.2: Diagrama Clientes-Servidor

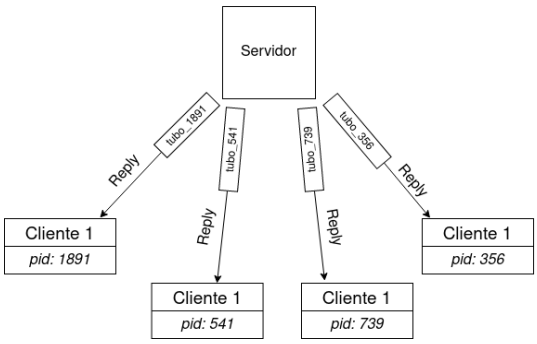

Figura 2.3: Diagrama Servidor-Clientes

2.2 Request

Nesta secção será apresentado a estrutura de um **Request**. Um Request é constituído por:

- PID do cliente
- Tipo do request
- Número de filtros
- Lista de filtros
- Nome do ficheiro de input
- Nome do ficheiro de output

A componente **Tipo de request** designa se o cliente inseriu um pedido do tipo **TRANSFORM**, **STATUS** ou **HANDSHAKE**. Na secção seguinte será apresentado com mais pormenor tais tipos de pedidos.

O **PID do cliente** será necessário para a comunicação no sentido servidor-cliente, com o propóstio do servidor ter as informações necessárias para se conectar individualmente com um certo cliente.

O restante das componentes compõem um pedido de aplicação de filtros:

- Lista de filtros corresponde a um array de 32 elementos (número máximo de filtros que se pode aplicar com um pedido), onde cada índice corresponde ao identificador do filtro(0,1,2,3,... ou 31, baseado na ordem do ficheiro de configuração).
- Número de filtros corresponde a quantidade de filtros pedidos pelo cliente.
- Nome do ficheiro de input é o ficheiro a ser aplicado o filtro.
- Nome do ficheiro de output é o ficheiro com o filtro aplicado.

```
typedef struct request {
  pid_t       client_pid;
  RequestType request_type;
  size_t      number_filters;
  int         requested_filters[MAX_FILTER_NUMBER];
  char        input_file[MAX_FILENAME_LEN];
  char        output_file[MAX_FILENAME_LEN];
} Request;
```

Figura 2.4: Excerto da struct Request

2.3 Handshake

HANDSHAKE é o tipo de pedido que realiza a primeira conexão do cliente com o servidor. A partir de tal pedido o cliente passa a conhecer quais são os filtros disponíveis. Os filtros disponíveis são enviados para o cliente através de uma **string** contendo os identificadores dos filtros separados pelo caráter delimitador **;** e terminada por
0. Isto é importante para a manutenção do código de modo a tornar fácil a adição de novos filtros ao progama. A validação dos filtros dado como argumentos do programa acontece do lado do cliente, justamente após um Request do tipo HANDSHAKE com o servidor. Segue um exerto da obtenção dos filtros disponíveis através de um HANDSHAKE.

```
Request handshake = {.client_pid = getpid(), .request_type = HANDSHAKE};

    /***** create the server_to_client pipe to get the Reply *****/
    char server_to_client_fifo_name[1024];
    sprintf(server_to_client_fifo_name, "tubo_%d", handshake.client_pid);
    mkfifo(server_to_client_fifo_name, 0644);

    /*************** send the request to server ****************/
    int client_to_server = open("client_to_server", O_WRONLY);
    write(client_to_server, &handshake, sizeof(struct request));

    /****** open the reply_pipe to read the server  reply *******/
    int server_to_client = open(server_to_client_fifo_name, O_RDONLY);

    char* filters = (char*) malloc(sizeof(char) * BUFSIZ);
    read(server_to_client, filters, BUFSIZ);
    /* finished handshake */
    return filters;
```

Figura 2.5: Excerto de um Request do tipo HANDSHAKE

É importante ressaltar o facto de primeiro criarmos o pipe particular de conexão do servidor com o cliente, enviarmos de seguida o Request e por fim abrir o extremo de leitura do pipe servidor-cliente. Isto porque caso abrirmos o extremo de leitura do pipe antes de enviarmos o Request, o processo ficará bloqueado porque não haverá nenhum outro processo que terá aberto o extremo de escrita, dado que o Request ainda não foi realizado. Pois isso, a ordem do código é vital para a sincronização de comunicação do cliente com o servidor.

2.4 Transform

Um Request do tipo **TRANSFORM** realiza a operação primordial do programa: aplicação de filtros. Dentro da contrução e execução deste Request, existe antecipadamente uma requisição do tipo HANDSHAKE (citada na secção anterior). Assim, obtemos os filtros disponíveis e de seguida passamos a sua validação e caso haja sucesso, começamos a preencher a Resquest. Estes passos acontecem nas funções `parser_filenames()` e `parser_filters()`.

Estabelecemos a conexão com o servidor de forma similar ao excerto mostrado na secção anterior. A diferença nesta requisição se encontra na *espera pelo feedback do servidor*. Ao ser feita uma requisição do tipo **TRANSFORM**, o cliente recebe constantes atualizações a respeito de seu pedido. Isto é, se o pedido esta **pending**, **processing** ou **finished**. Sabendo isto, aposó o envio do Request ao servidor, o cliente deve aguardar informações vinda do servidor, pelo *pipe* particular entre este cliente e o sevidor. Segue-se então um excerto de como isto é realizado:

```
while (read(server_to_client, &reply, sizeof(struct reply)) > 0) {
            show_state(reply.state, last_state);
            if (reply.state == FINISHED) break;
        }
        unlink(server_to_client_fifo_name);
// (...)
}
```

Figura 2.6: Excerto de um Request do tipo TRANSFORM

Reparemos aqui na introdução de uma nova estrutura chamada **reply**. De modo a sabermos o estado de um Request por parte do cliente, criamos a struct `reply`, que corresponde ao "oposto"de um Request, sendo o tal "feedback"mencionado, no sentido servidor cliente. Assim, a **struct reply** contém as 3 formas possíveis de estado de um Request (sendo passível de ser ampliada facilmente caso o programa receba atualizações);

```
typedef enum state { NOTHING = -1, PENDING, PROCESSING, FINISHED } State;

typedef struct reply {
    State state;
} Reply;
```

Figura 2.7: Excerto de uma Reply

OBSERVAÇÂO: O estado **NOTHING** simboliza algum erro por parte do servidor, como uma falha na execução de um filtro.

3. Servidor

3.1 Introdução

Será no **servido**, onde todos os **Request** serão tratados e organizados. Aqui, decidiu-se a divisão do código em 2 processos distintos, juntamente com o processo pai.

O processo pai encontra-se encarregue de receber todos os **request**(s) enviados pelo **cliente**. Aqui, terá que ser feita uma decisão e consoante isso, esse mesmo *request* será enviado para um dos processos criados a partir do processo pai.

Um desses dois processos ficará apenas encarregue pela execução dos **request**(s). Este processo (*Processo das execuções*) apenas receberá **request**(s) prontas a serem executados. Já o outro, (*Processo da Queue*) encontra-se encarrega pela organização e validação dos **request**(s) a serem executados.

Toda a comunicação necessária feita entre o processo pai e os processos criados por este, é feita através da utilização de pipes anónimos.

```
int pipe_onhold[2], pipe_updates[2], pipe_execucao[2];
```

Figura 3.1: Pípes anónimos utilizados

Aquando a iniciação do servidor este terá que ler e armazenar informação relativa a todos os filtros existentes. De modo a armazenar toda essa informação, criou-se as seguintes estruturas:

```
typedef struct filtro {
  char*  identificador;
  char*  ficheiro_executavel;
  size_t max_instancias;
  size_t em_processamento;
} Filtro;

typedef struct catalogo_filtros {
  Filtro* filtros[MAX_FILTER_NUMBER];
  size_t  used;
} CatalogoFiltros;
```

Figura 3.2: Excerto de código da struct catalogo_filtros e filtros

Com isto, conseguimos criar um array de filtros com comprimento fixo através de um caminho para uma configuração dada pelo utilizador no momento da inizialização do servidor.

3.2 Divisão do servidor

3.2.1 Processo da Queue

Este Processo, como outrora mencionada, tratará da organização dos pedidos feitos, recebendo tanto pedidos a serem executados, como pedidos já concluídos. Com isto conseguimos manter o nosso *Catálogo de filtros* sempre atualizado.

Para auxílio neste processo, decidiu-se que seria essencial a criação de uma estrutura de dados intitulada **Queue**.

```
typedef struct queue {
  Request*      request;
  struct queue* prox;
} Queue;
```

Figura 3.3: Excerto de código da struct queue

Aquando a receção de um **request** a ser executado, este irá verificar se o mesmo já pode ser executado. Se esse for o caso, enviará o mesmo para o *processo das execuções*, sendo assim, este nunca chega a entrar na queue. Caso contrário, o pedido é sempre adicionado à cauda da queue.

Aquando a receção de um *request* concluído, o *processo da Queue* primeiramente irá atualizar o *catálogo de filtros*, após isso verificará se existe algum *request* que se encontre na queue pronto a ser executado, começando essa pesquisa à cabeça da *queue*. No caso de ter encontrado algum, remove-o da queue e envia-o para o *Processo das Execuções*.

3.2.2 Processo das Execuções

É neste processo que os *request* irão ser executados. Primeiramente informamos o cliente que o seu pedido encontra-se a ser processado, após isso iremos executar o pedido em questão. Apresentando o resultado desse request e enviando esse mesmo resultado para o cliente.

4. Extensibilidade da aplicação

Graças ao mecanismo de **Handshake** criado, podemos garantir uma base sólida para constantes atualizações do programa. Isto porque o lado do **cliente** se baseia no resultado de um **Request** do tipo **Handshake** para proessguir com a validação dos comandos inseridos pelo utilizador. Assim, o programa cliente é capaz de analisar os filtros inseridos a partir do estado do ficheirod e configuração do lado do servidor. Mudando o ficheiro de configuração, mudamos a forma de validar os filtros.

5. Conclusão

Reconhecemos que a maior dificuldade encontrada neste trabalho foi lidar com os **Deadlocks**. Dada a magnitude de organização imposta no processamento de um pedido, era inevitável a utilização de uma quantidade de **forks()** e **pipes** que nunca haviamos trabalhado antes. Gerir a abertura e fechamento de **pipes anônimos** gerou nossos maiores obstáculos e imprevistos durante o trabalho. Porém, com este trabalho passamos a conhecer novos mecanismos deste nível de programação que a disciplina proporciona, como por exemplo, conhecer maneiras de "não bloquear"um processo, mesmo quando não há informações a serem lidas em pipes.

A. Diagrama de Classes