Stream Processing

Relatório do trabalho prático de Sistemas Operativos

Mestrado Integrado em Engenharia Informática

Grupo 85

Ana Paula Carvalho (A61855)
João Pires Barreira (A73831)
José Moreira (A43099)

Maio 2017

Universidade do Minho
Escola de Engenharia

Índice

1 Introdução

Foi-nos proposto, no âmbito da unidade curricular de Sistemas Operativos, o desenvolvimento de um sistema de *stream processing*. Este tipo de sistema utiliza uma rede de componentes para filtrar, modificar e processar um fluxo de eventos, permitindo tratar grandes quantidades de dados explorando a concorrência entre processos.

Ao longo deste relatório, iremos explicitar a nossa abordagem ao problema, justificar a estrutura do nosso sistema e demonstrar a utilização dos conhecimentos adquiridos nas aulas, nomeadamente no que toca a criação de processos, duplicação de descritores, criação de *pipes* com nome, execução de processos, sinais e várias *system calls*.

2 Estrutura da nossa implementação

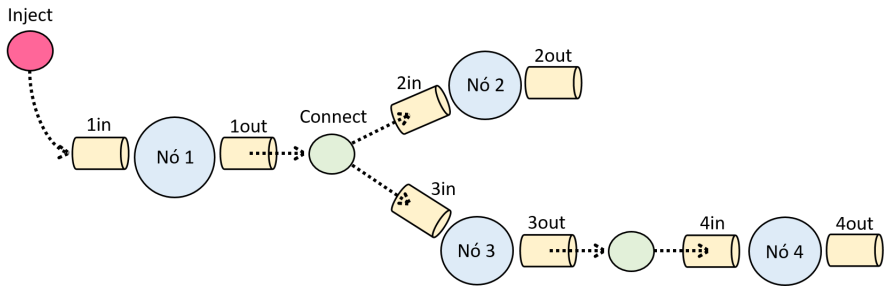

Figura 1 - Esquematização de uma rede criada pelo nosso programa

2.1 Variáveis globais

- **int nodes[MAX_SIZE]** - *Array* de inteiros que indica os nós existentes na rede. Todas as posições do *array* são inicializadas a zero. Caso, por exemplo, exista o nó 5 na rede, a posição índice 5 deste array terá o valor 1.

- **int nodespid[MAX_SIZE]** - *Array* com os *PIDs* dos processos dos nós existentes na rede.

- **struct fanout** - Estrutura de dados responsável por guardar as informações relativas a um *fanout* (utilizado para as conexões entre os nós da rede). Possui um campo correspondente ao *PID* do processo que executa o *fanout* (pid), um *array* de inteiros com os nós das suas saídas (outs) e um inteiro que apenas tem o tamanho do *array* anterior (numouts).

- **Fanout connections[MAX_SIZE]** - *Array* de estruturas *fanout* que corresponde ao conjunto de todas as conexões entre nós da rede. O índice deste array indica o ID do nó IN do fanout. Todas as posições deste array são inicializadas a *NULL*. Assim, se, por exemplo, existir uma conexão a ligar o nó 5 a outros quaisquer nós, a posição 5 deste *array* tem um apontador para uma estrutura *fanout* com os dados correspondentes ao mesmo.

- **int stopfan** - Serve para parar um processo a executar um *fanout* (conexão entre os nós) sem ser necessário fazê-lo abruptamente (i.e. com *SIGKILL*). Pára o ciclo de leitura daquele processo na próxima iteração. É inicializado a zero e colocado a um pelo *handler* do sinal *SIGUSR1* recebido pelo processo que executa o *fanout*.

- **int busy** - Flag que indica se se está a processar um comando. É utilizado pela *main* e pelo *interpretador*. Faz com que os comandos lidos apenas são passados ao interpretador quando nenhum outro está a ser processado. Isto faz com que não hajam concorrências na **criação** dos

componentes da rede, evitando erros, por exemplo, ao aceder a FIFOs que ainda não tenham sido criados. A execução da rede continua a ser feita **concorrentemente**.

2.2 Comando *node*

É criado um processo responsável por executar o comando recebido. Além disso, são criados dois pipes com nome (*FIFOs*) correspondentes ao *FIFO in* (entrada do nó) e ao *FIFO out* (saída do nó). Para o nó cujo ID é 1, estes fifos têm o nome 1in e 1out (e são criados na pasta *tmp*).

Primeiro são redirecionadas a entrada e a saída do processo do nó (que executa o comando) para, respetivamente, a saída do *FIFO in* e a entrada do *FIFO out*. Caso o comando recebido não seja um filtro/componente, a saída é descartada, sendo chamada a função responsável pela criação do nó na rede com uma flag específica ativada.

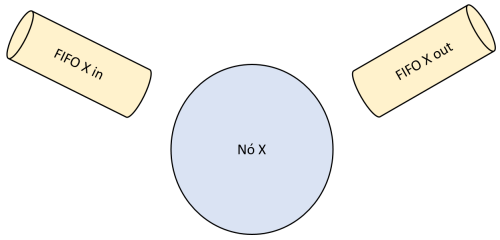

Figura 2 - Esquematização do nó

2.3 Comando *connect*

Em relação ao comando *connect*, é criado um processo que executa uma função *fanout*. Um *fanout* é uma função que recebe um descritor de ficheiro de onde lê, e replica o output nos nós de saída recebidos como parâmetro.

Assim sendo, apenas é criado uma nova conexão (*struct fanout*) com os dados recebidos e com o *PID* do processo que executa esta função. Caso já exista uma conexão a partir do nó recebido, essa conexão é terminada (com *SIGURS1* - ver descrição da variável global *stopfan*), sendo guardados os nós da saída dessa conexão pré-existente. De seguida, é criada uma nova conexão que além dos nós da saída recebidos, acrescenta os nós da conexão que existia anteriormente.

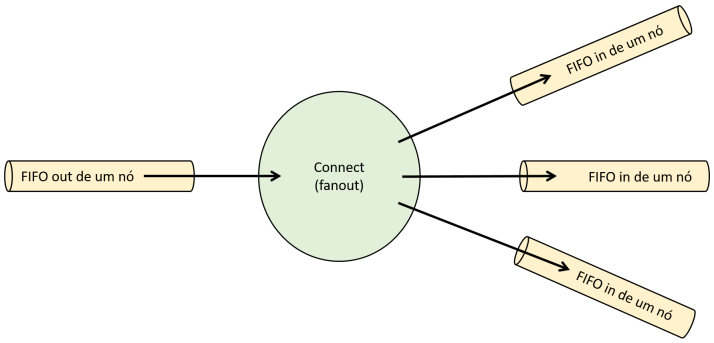

Figura 3 - Esquematização de uma conexão entre nós

2.4 Comando *disconnect*

Neste caso, o controlador vai ver se existe, de facto, uma conexão entre os dois nós recebido e, em caso afirmativo, termina o processo relativo à conexão pré-existente, criando um processo que executará a nova conexão com os nós de saída da conexão anterior (excetuando o nó que foi desconectado).

2.5 Comando *inject*

Para o caso do *inject*, apenas é criado um processo que executa o comando recebido para o *FIFO in* do nó da rede indicado. Para isso, esse *FIFO* tem de ser aberto e feito o redirecionamento apropriado no contexto do processo que executa o comando do *inject*.

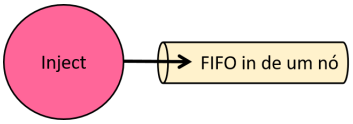

Figura 4 - Esquematização da injeção de dados num nó

2.6 Comando *remove* (funcionalidade avançada)

A função relativa à terminação de um nó da rede remove todas as ligações do nó a remover existentes na rede, matando, posteriormente, o processo que se encontrava a executar o processo relativo ao nó removido.

2.7 Comando *change* (funcionalidade avançada)

A função relativa à mudança do comando de um nó da rede remove o nó pré-existente (com o mesmo ID) da rede e cria um novo nó (também com o mesmo ID) que executará o novo comando, mantendo todas as conexões previamente existentes.

3 Funcionalidades

3.1 Filtros/Componentes

O nosso programa utiliza quatro filtros básicos que recebem input pelo *standard input* e produzem output para o *standard output*.

- ***Const*** - Este filtro recebe na sua execução um valor que anexará a cada linha que recebe.
- ***Filter*** - Este filtro reproduz as linhas que satisfazem uma condição indicada nos seus argumentos. Por exemplo, executando "filter 2 ¡ 4"reproduz as linhas em que o valor da segunda coluna e menor que o da quarta.
- ***Window*** - Este filtro reproduz todas as linhas acrescentando-lhe uma nova coluna com o resultado de uma operacão calculada sobre os valores da coluna indicada nas linhas anteriores
- ***Spawn*** - Este filtro reproduz todas as linhas, executando o comando indicado uma vez para cada uma delas, e acrescentando uma nova coluna com o respetivo exit status

3.2 Funcionalidades básicas

3.2.1 Permanência de uma rede

Todos os nós criados são processos independentes que ficam em funcionamento até serem removidos. Além disso, não terminam com *EOF*, ficando bloqueados até que recebam um novo input.

3.2.2 Concorrência nas escritas no mesmo nó

Cada nó da rede é connectado com um processo *fanout* que gere a entrada e respetivas saídas dos nos. A operação de escrita é utilizada usando um *buffer* com tamanho PIPE_BUF, sendo assim atómica, garantida pelo Sistema Operativo.

Caso seja preciso remover um nó, ou alterar algum componente, o processo de escrita não é interrompido, e terminará qualquer leitura e ou escrita.

3.2.3 Nós sem saída definida

Quando um nó não pertence a nenhum dos filtros pré-definidos a sua saida é descartada.

3.2.4 Especificação incremental da rede

A rede pode ser aumentada/alterada a qualquer altura. A nossa função fannout garante que não se fica com escritas/leituras parciais.

3.3 Funcionalidades avançadas

3.3.1 Casos de teste

No ficheiro *zip* relativo à entrega deste trabalho, foram também incluídos vários exemplos de redes que podem ser testadas utilizando o nosso programa (na pasta testes).

3.3.2 Alteração de componentes

Foi também implementada a a opção de mudar o comando executado por um nó da rede (comando *change*). Mais uma vez, existiu a preocupação em não corromper os eventos a serem processados (garantindo que nenhum evento é perdido). A rede fica a funcionar como estava anteriormente, mudando apenas o comando executado pelo nó recebido.

3.3.3 Remoção de nós

A opção de remoção de um nó da rede verifica se não escreve para outros nós e garante que não há problemas na criação de um novo nó naquela posição. Ou seja, garante também que cada evento (linha de texto) chega corretamente ao destino, não sendo corrompida.

3.4 Outras funcionalidades

O controlador recebe os comandos sequencialmente pelo *standard input*. No entanto, pode também receber um ficheiro de configuração como parâmetro (i.e. ./controlador ficheiro_de_configuração.txt).

No caso da leitura de um ficheiro de configuração, os comandos lidos apenas são passados ao interpretador quando nenhum outro está a ser processado. Isto faz com que não hajam concorrências na **criação** dos componentes da rede, evitando erros, por exemplo, ao aceder a FIFOs que ainda não tenham sido criados. A execução da rede continua a ser feita **concorrentemente** (que é fundamental no contexto do *stream processing*).

4 Conclusão

Este trabalho prático foi bastante interessante visto que pudemos aplicar os conceitos adquiridos nas aulas de Sistemas Operativos. A resolução sequencial dos guiões práticas, revelou-se uma peça fundamental para tornar possível a finalização deste projeto, em que fomos bem sucedidos na implementação tanto das funcionalidades básicas como das funcionalidades avançadas. A maior dificuldade sentida foi não sermos capazes, inicialmente, de visualizar o produto final da nossa abordagem ao problema, obstáculo que foi ultrapassado aos poucos e estabelecendo um objetivo de cada vez.