INSTITUTO SUPERIOR TÉCNICO

PROGRAMAÇÃO DE SISTEMAS

Project

Key-Value Store

Grupo 1

75847 – José Dias

76090 – Diogo Ferreira

29 de Maio de 2016

Índice

1. Arquitetura do sistema

No sistema *key-value* é possível identificar uma arquitetura com diversos componentes. Dessas componentes fazem parte: o servidor que vai ser subdividido em dois componentes, *Front server* e *Data server*; o cliente que vai utilizar uma biblioteca fornecida; o utilizador que poderá realizar comandos, por exemplo "quit", a partir da interface (terminal) que possibilita a ligação ao servidor *Front server*; dois ficheiros (backup e log) que têm como fim uma possível recuperação de dados.

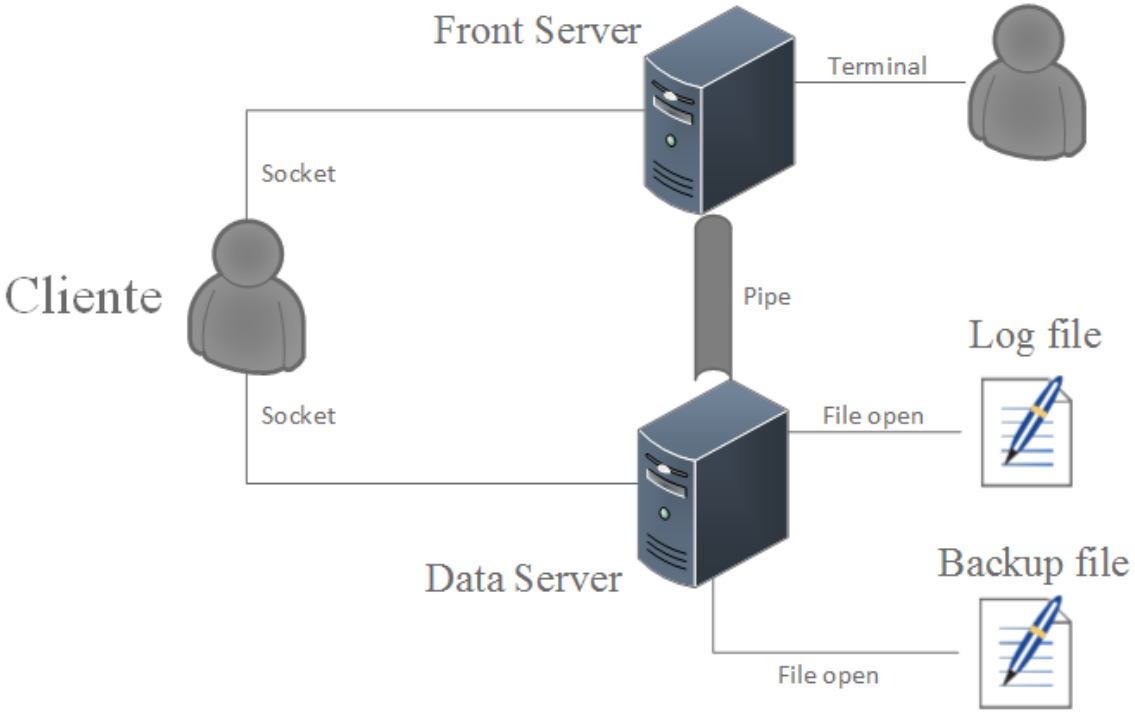

Figura 1 - Arquitetura do Sistema.

O programa do cliente deve seguir o protótipo das funções que vão ser incluídas na biblioteca, para que caso a biblioteca seja substituída, não seja necessário realizar alterações no cliente, permitindo a reutilização e extensibilidade do código.

O *Front server,* apesar de fazer parte do servidor, não deve saber dos detalhes internos das outras componentes do servidor, como por exemplo os dados guardados no *Data server*.

Apesar de pequenas dependências, os componentes são totalmente independentes.

2. Data Server

2.1. Estrutura de dados

De modo a ser possível guardar os valores dos pares (key, value) de forma eficiente implementou-se uma lista ligada ordenada numericamente e em ordem crescente dos valores da key.

Para os nós da lista é definida uma estrutura de dados à qual chamou-se de *node,* que contém:

- Key (uint32_t key);
- Value (char * value);
- Ponteiro para o próximo nó (struct node * next).

2.2. Mecanismos de sincronização

Como o *Data server* responde a pedidos de vários clientes em simultâneo necessita de de uma forma de sincronização no acesso aos dados. Para tal consideram-se dois read/write locks e um mutex.

O primeiro lock é bloqueado imediatamente antes da lista ser percorrida e desbloqueado quando se deixa de percorrer a lista. Quando a operação a executar é eliminação utiliza-se um write lock. Quando a operação é de leitura ou inserção de valores utiliza-se um read lock.

Quando um determinado elemento é encontrado para modificação, eliminação ou leitura, bloqueia-se o segundo lock antes da primeira instrução que envolve modificação da lista e desbloqueia-se imediatamente após a última. Nesta fase quando a operação for de leitura utiliza-se um read lock, caso contrário utiliza-se um write lock.

O mutex é utilizado para ser adicionado um valor à lista de cada vez, este é apenas utilizado nas inserções da lista.

3. Front server

De maneira a conseguir realizar *handle* dos inputs ou do sinal, ou seja, terminar o servidor de forma ordeira realizando um novo *backup*, são inicializados dois *pipes* (comunicação bidirecional) para a comunicação entre o *Front Server* e o *Data Server.* Um dos *pipes* serve para o *Front Server* pedir ao *Data Server* para realizar o *backup* e o outro para o *Data server*

responder ao *Front server* que o *backup* já se encontra realizado. Para além dos *pipes* existem duas variáveis globais com os *pids* referentes ao *Front server* e ao *Data server* de modo a que seja possível o *Front server* terminar os dois.

3.1. Inputs do utilizador

O *Front server* deve ser capaz de receber inputs do utilizador a partir do terminal quando é inicializado pela primeira vez. Quando recebe o comando "quit", envia o pedido para o *Data server* realizar o backup através do *pipe* referido em na secção 3.

Caso o *Front Server* tenha sido chamado através do *Data Server*, ou seja, o *Front server* já foi iniciado uma vez no passado e agora o *Data server* tinha deixado de receber a comunicação deste reinicializando-o, o *Front server* já não vai conseguir aceder ao terminal e receber os inputs do utilizador.

3.2. Sinais

O *Front server* consegue receber o sinal SIGINT que é enviado carregando no Ctrl+C no terminal onde foi executado no programa, ao receber este sinal o *Front server* faz com que o sistema servidor (*Data*+*Front*) termine ordeiramente realizando sempre o *backup* (o *backup* e a terminação é executada da maneira enunciada previamente).

4. Biblioteca do Cliente

Funções implementadas do lado da biblioteca do cliente:

- **int kv_connect (char * kv_server_ip , int kv_server_port);**

Nesta função é realizado o *connect* primeiramente ao *Front Server* que devolve o porto do *Data Server* e depois é realizado o *connect* ao *Data Server* e é retornado o *file descriptor* da conexão com o *Data Server*.

- **void kv_close(int kv_descriptor);**

Esta função simplesmente realiza o *close()*. Apesar de esta função apenas ter um *close* não significa que deixe de ser utilizada, pois se mudarmos de bibliotecas ou quisermos que o *close* realize algo diferente não será necessário mudar todos os clientes, mas sim a biblioteca.

- **int kv_write(int kv_descriptor uint32_t, char * value , uint32_t value_length, int kv_overwrite);**

Esta função cria a mensagem para a comunicação (ver Protocolos de comunicação) que vai enviar no *socket* e envia-a para o *file descriptor* que lhe é passado. O valor da operação vai ser definido entre 1 ou 2 dependendo se a opção *overwrite* for ou não ativada (0 ou 1).

É realizado um *read* do *socket* de maneira a saber se o *write* na *key-value store* foi bem ou mal realizado e deverá ser retornado de modo ao cliente ter conhecimento da realização ou não do seu pedido.

- **int kv_read(int kv_descriptor, uint32_t key, char*value, uint32_t value_length);**

Esta função cria a mensagem para a comunicação (ver Protocolos de comunicação) que vai enviar no *socket* e envia-a para o *file descriptor* que lhe é passado. O valor da operação é igual a 3 e o *value length* igual ao comprimento que pretende ler. A mensagem é enviada pelo *socket* e deverá ser recebido no *socket* o valor do tamanho que se encontrava no *key-value store.* Posteriormente é recebido o *value* e caso não exista no valor do comprimento deverá ser retornado um valor de erro.

- **int kv_delete(int kv_descriptor,uint32_t key);**

Esta função cria a mensagem com a operação igual a 4 e o valor do comprimento igual a 0.

A mensagem é enviada a partir do *socket* identificado pelo *file descriptor* (kv_descriptor) e é realizado um *read* para saber se esta função foi bem ou mal concretizada.

5. Protocolos de comunicação

De modo a definir um protocolo de comunicação entre o cliente e o servidor definiu-se uma estrutura de dados para a mensagem que estes enviam entre eles.

```
typedef struct msg{

    uint8_t operation;

    uint32_t key;

    uint32_t value_length;
```

```
}message;
```

O valor da operação varia entre 1 e 4 consoante se pretende realizar um *write* na *key value store* com *overwrite* (2) ou não (1), se se pretende realizar um *read* (3) ou um *delete* (4).

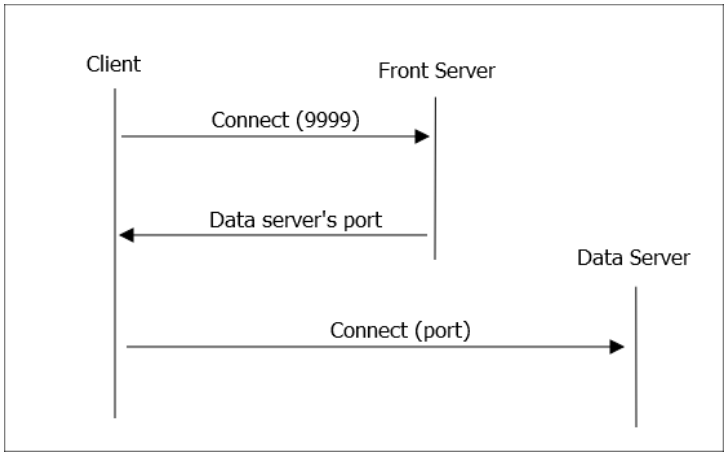

Figura 2 - Protocolo para conexão com o Data server.

Em resposta e estas mensagens é enviado um valor do tipo inteiro que pode ter um valor negativo (NOK) caso a mensagem não seja a esperada ou pode ter um valor positivo (OK) caso a mensagem se encontre correcta.

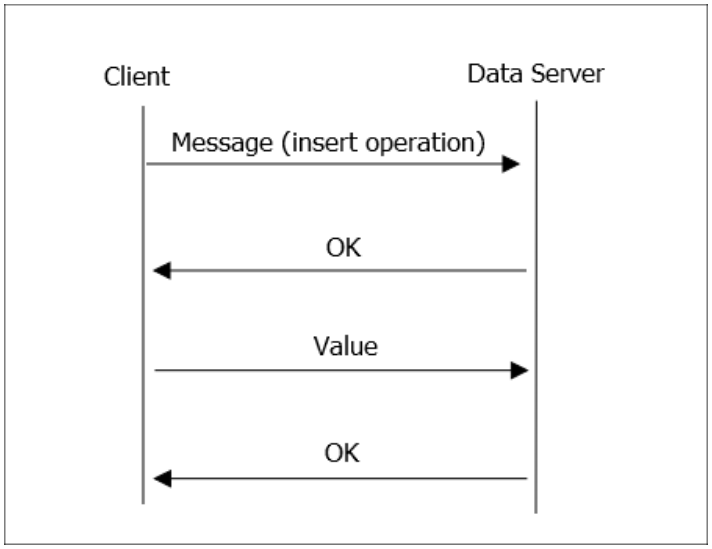

Figura 3 - Protocolo para inserção de um valor na key-value store.

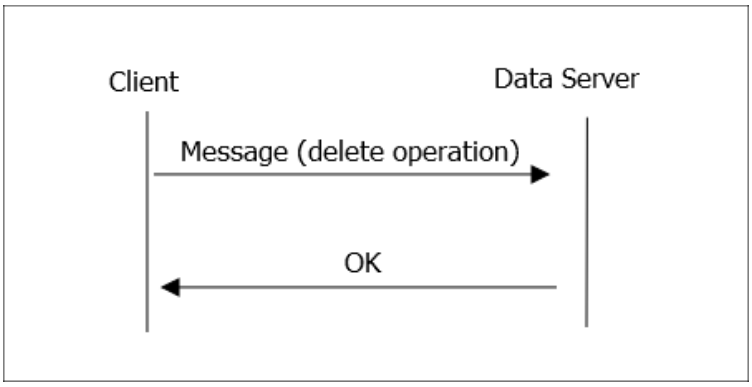

Figura 4 - Protocolo para apagar um valor da key-value store.

6. Gestão de Threads

6.1. Threads Data Server

Quando o servidor é executado existe apenas uma *thread* a que chamamos de *main*, mas tendo em vista a realização de um servidor capaz de receber vários pedidos recorremos a uma política de *threads on demand*, sendo que após cada *accept* de um novo cliente no *socket* é criada uma *thread* enviando como argumento o *file descriptor* do *socket* que foi estabelecido com o cliente.

Para além desta criação de *threads on demand* existe também a criação de *threads* quando é inicializado o *Data server*, que vão ter como responsabilidade a verificação do estado do *Front server* e a escrita no ficheiro de backup.

6.2. Threads Front Server

Tal como no *Data server,* o *Front server* utiliza uma política de criação de *threads on demand*, a criação de *threads* é efetuada após o *accept* do *socket*.

Para além da criação de *threads on demand* existe também outras *threads* que vão verificar o estado do servidor de Data e receber os inputs do terminal realizando a comunicação com o utilizador.

7. Tolerância à falta

No nosso programa principal é realizado uma chamada de sistema *fork* e para os dois processos diferentes são chamadas funções diferentes às quais chamámos *maindataserver*() e *mainfrontserver*(). Estas enviam de 1 em 1 segundo o seu *pid* através um *pipe* para o outro

componente do servidor, por exemplo, após o *fork,* se o *Front server* tiver como *pid* 445 então este vai enviar de 1 em 1 segundo 445 para o *Data server* através do *pipe*. Se após cinco segundos um dos componentes do servidor (*data* ou *front*) não obtiver resposta relativamente ao outro componente este deverá enviar um sinal de *kill* (SIGKILL) para o último valor de *pid* que chegou e deverá realizar a chamada de sistema *fork* e voltar a correr a função *main* correspondente ao componente em falta.

Para conseguirmos a comunicação bidirecional utilizaram-se dois *pipes*.

8. Backup/Recuperação de dados

Para a recuperação de dados utilizou-se um sistema híbrido com recurso a um *log* e a um *backup*. Sempre que o *Data server* for iniciado é realizada uma leitura de valores do *backup* seguido do *log*.

8.1. Log

O ficheiro log.txt guarda todas as instruções que podem causar alterações na lista do *Data server*, *delete* ou *insert*. No log é guardado numa linha o valor da operação, a *key* e o comprimento do *value*, caso a operação seja de *delete* então este valor é 0, na linha seguinte é apresentado o *value* caso a operação não seja de *delete*. Como a utilização de uma ou duas linhas no *log* é dependente da operação, implementámos um *mutex* de maneira a que se for necessário escrever duas linhas, nada possa escrever entre elas, caso fosse utilizado apenas uma linha, este *mutex* não seria necessário pois a chamada de sistema *write* funciona atomicamente e não deixaria escrever uma a meio da outra.

O ficheiro log é bastante importante pois caso a lista apresente um tamanho considerável a realização de um backup pode ser muito demorada quando comparada com a colocação de apenas uma instrução num ficheiro.

8.2. Backup

No ficheiro backup.txt é guardado numa linha o valor da *key* seguido do tamanho do *value* que é apresentado na linha seguinte. O *backup* é criado/reescrito sempre que o servidor termina ordeiramente, sendo que após a escrita o log é reinicializado, caso aconteça um erro e o log não seja apagado não vai realizar nenhuma diferença na lista quando o *Data server*

for inicializado novamente pois as instruções enviadas só escrevem por cima do que já se encontra na lista.

9. Tratamento de erros

Relativamente ao tratamento de erros se os *pipes*, *mutex*s ou *threads* forem mal inicializados vão surgir mensagens de erro.

No caso de os *sockets* não serem criados, *Data server* tenta realizar o *bind* no porto numericamente superior até conseguir. O *Front server* sabe qual o porto em que foi realizado o *bind*, pois este porto é ser uma variável global.

No caso de não conseguirem ser abertos os ficheiros log.txt e backup.txt vão surgir mensagens no terminal de erro.