INSTITUTO SUPERIOR TÉCNICO

PROGRAMAÇÃO DE SISTEMAS

PROJECTO

Relatório

78 247 – Rui Miguel Figueiredo
78 537 – Jorge Filipe Sacadura

Grupo 15
João Nuno de Oliveira e Silva

29 de Maio de 2016

1 Arquitetura do Servidor

Através da figura pode-se verificar que a arquitetura é constituída por:

- Front Server : Que vai corresponder servidor que recebe conexões dos clientes e os encaminha para o servidor de Dados;
- Data Server : Que representa o servidor onde os dados estão armazenados e com o qual se vai comunicar para realizar operações;
- Cliente : Representa cada cliente que se liga ao servidor;
- FIFO : Utilizada na comunicação Data Server - Front Server;
- Threads : Utilizadas para processar clientes em simultâneo;
- Ligações TCP : Para realizar a comunicação do servidor com os clientes utilizando o protocolo estipulado.

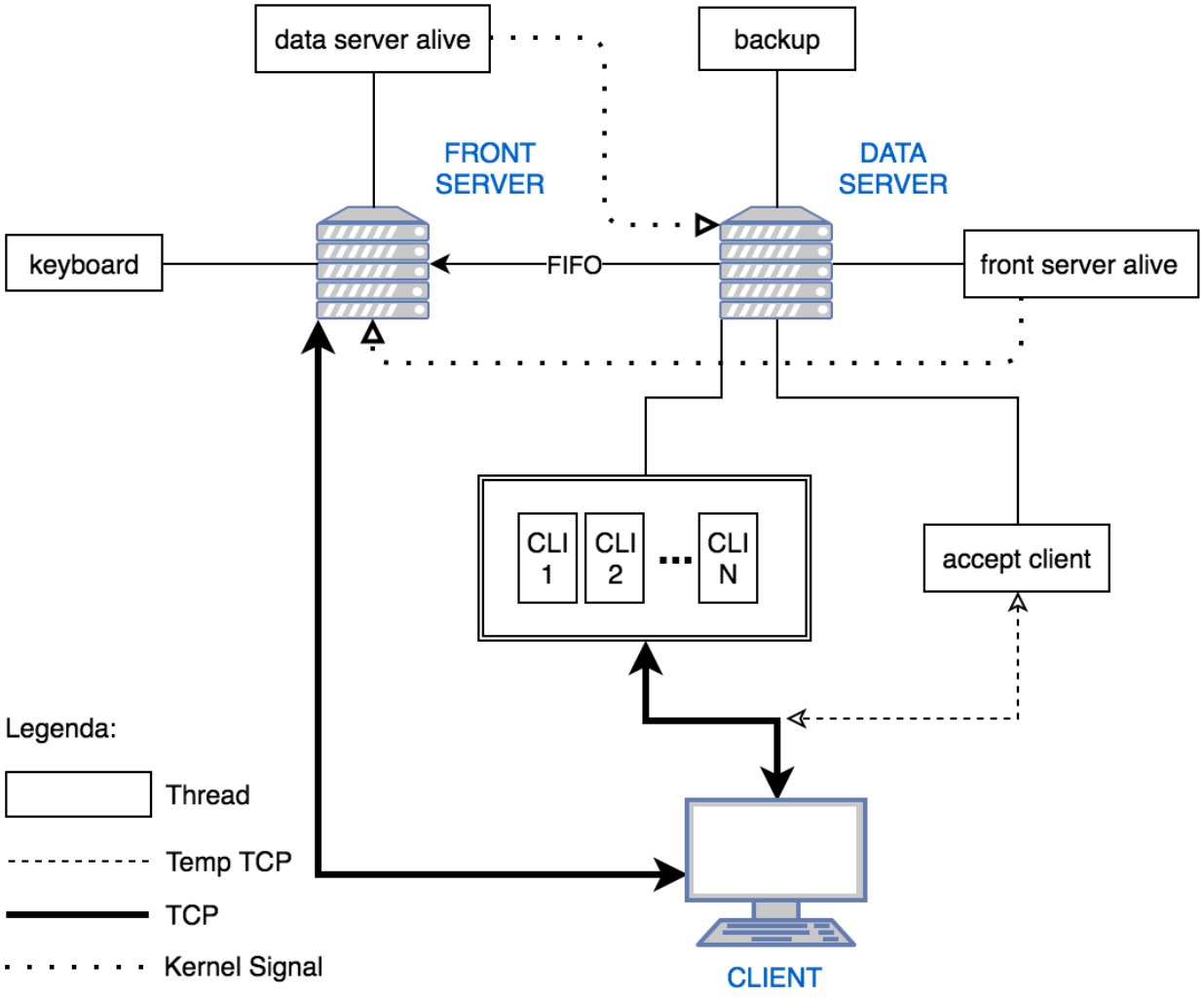

Figura 1: Diagrama de blocos representativo da arquitetura.

1.1 Implementação das Componentes

O servidor foi implementado de forma modular, sendo que é possível remover ou adicionar componentes com relativa facilidade. Seria possível desativar por exemplo o módulo de recuperação de falhas ou o módulo que trata da comunicação com o operador do servidor (comando de quit) sem que a operação das restantes componentes fosse afetada.

O servidor utiliza no máximo dois processos em cada momento (um para o Front Server e outro para o Data Server e um número de threads no Data Server que cresce linearmente com o número de clientes que estão ligados ao servidor para além de um número fixo com funções bem definidas.

1.2 Estrutura de Dados de armazenamento

Para armazenar os pares (key,value) em memória optou-se por utilizar uma hashtable. Uma hashtable é nada mais nada menos que um vetor de listas em que a função de dispersão é que define em que lista vai ficar guardado cada par (key, value). Assim cada elemento da lista vai ter armazenado uma key e o valor correspondente a essa key.

```
typedef struct hashtable_s {
    int line_nr;
    struct item_s ** table;
}hashtable_t;


typedef struct item_s{
    unsigned int key;
    char * value;
    struct item_s * next;
}item_t;
```

Como função de dispersão usou-se o resto da divisão inteira por 2011. Esta função de dispersão vai definir em muito o desempenho do servidor. De forma a ter poucas colisões e logo um melhor desempenho deviria-se efetuar um melhor estudo desta função dependendo das especificações finais do servidor e de quais key seriam previsivelmente mais usadas.

1.3 Inicialização dos Servidores

Numa inicialização normal do sistema o utilizador lança o Front Server que vai criar o seu socket e tentar dar bind na porta 9999 e se não conseguir nesta vai tentar as seguintes até conseguir em uma. Depois de dar bind vai lançar o Data Server. Abre um pipe e fica a espera que o Data Server lhe comunique a porta em que deu bind. Depois de ser informado da porta lança uma thread que vai verificar o estado do Data Server. Seguidamente entra num while onde fica à espera de conexões de clientes e vai fazendo accepts e informando os clientes da porta do Data Server. O Data Server depois de ser lançado vai fazer bind tentando primeiro a porta 10500 e se não conseguir vai tentando as seguintes até conseguir. Seguidamente abre um pipe e comunica ao front server a porta em que deu bind. Seguidamente irá realizar carregna secção do mesmo. Feito isto lança a thread que vai verificar se o Front Server sofreu um crash ou se

ainda continua a sua execução normal e segue para o while onde fica à espera por conexões de clientes.

2 Paralelismo

2.1 Threads

O Front Server possui três threads:

- `thread keyboard` : é utilizada para receber comandos do teclado, neste caso apenas o comando QUIT está implementado mas facilmente poderiam ser adicionados mais.
- `thread principal` : receber conexões dos clientes e a responder com a porta do data server. Como estas conexões duram muito pouco tempo uma vez que apenas é feito um send e seguidamente a conexão é terminada não se justificava estar a criar uma thread para tratar de cada cliente como veremos que é feito no data server.
- `thread Fault Tolerence` : É utilizada para garantir que se o Data Server 'crashar' o mesmo é relançado.

O Data Server possui três threads fixas:

- `thread backup` : é utilizada para fazer backup dos dados da hashtable em intervalos regulares como será abordado mais a frente.
- `thread principal` : receber conexões dos clientes e criar uma outra thread responsável por tratar das operações desse cliente.
- `thread Fault Tolerence` : Tem função semelhante à do Front Server.

Para tratar das conexões dos vários clientes foi criado um sistema de multithread on demand no data server. Cada vez que o servidor recebe uma conexão de um cliente cria uma thread que vai tratar de lidar com todas as operações que esse cliente pretender realizar. Quando o cliente sair a thread é destruída. Este esquema pareceu-nos mais eficiente que um sistema de Pull uma vez que por norma as conexões são longas e então não compensava ter várias threads a consumir recursos uma vez que o tempo que demora a criar cada thread é desprezável face ao tempo que demora cada conexão.

2.2 Sincronização no acesso à estrutura de dados

Existem alguns problemas quando varias threads acedem aceder à hash table em simultâneo, por exemplo pode haver problemas se uma thread fizer um delete e outra thread estiver a um read, quando esta vai ler a estrutura pode já não existir.

De forma a resolver este problema recorreu-se ao uso de mutex. Um mutex evita que duas treads tenham acesso simultaneamente a um recurso compartilhado que se localiza numa região cria. Assim cada vez que um cliente acede a uma das listas da hash table, faz um lock e todos os outros que tentem aceder à mesma lista bloqueiam até que este primeiro conclua a operação e realize unlock.

Além disso cada vez que o mecanismo periódico de backup é efetuado também todas as operações sobre a hashtable são bloqueadas até que o mesmo seja concluído.

3 Protocolo de comunicação Cliente <-> Servidor

A biblioteca de comunicação cliente <-> servidor poem ao dispor do cliente a relização de cinco operações.

1. Conectar-se ao servidor
2. Ler o valor de uma Key
3. Escrever o valor de uma Key
4. Apagar o valor de uma key
5. Fechar a conexão

Para a realização destas operações são trocadas mensagens entre o cliente o servidor, estas mensagens são compostas por três campos, ”key”, ”value_length”e ”info”.

O campo ”key”tem a função de identificar a chave sobre a qual se pretende realizar a operação.

O campo ”value_length”serve para definir o tamanho do valor a ler/escrever na ”hashtable”.

O campo ”info”serve como uma espécie de flag, que comunica a operação a realizar ou o retorno que a operação obteve no servidor.

Quando um cliente quer fazer uma leitura envia uma mensagem ao servidor a informar que quer fazer uma leitura referindo qual a key que quer ler e o número máximo de bytes que pode ler. O servidor responde com uma mensagem a informar quantos bytes tem disponíveis para o cliente ler ou -2 em caso de a key não existir no servidor. Seguidamente se a key existir o cliente lê o valor do servidor.

Quando o cliente quer fazer uma escrita envia uma mensagem a informar que quer fazer uma escrita, quantos bytes quer escrever e a key correspondente. O servidor recebe esta informação e seguidamente vai ler do socket o número de bytes que o cliente quer escrever. Por fim o servidor comunica ao cliente se a escrita foi ou não feita com sucesso. De forma a tornar este processo mais eficiente podia primeiro verificar-se do lado do servidor se a key já existe e no caso do cliente não querer fazer overwrite nem sequer era necessário comunicar o valor a escrever, isto para mensagens muito grandes iria apresentar algum ganho em tempo de processamento dos dados no entanto para mensagens pequenas poderíamos já não ter ganho nenhum uma vez que o fluxo de recv/send aumenta.

Na operação de delete o cliente envia a informação de delete e recebe uma mensagem com o resultado dessa operação, sucesso ou insucesso(no caso de se apagar um key inexistente).

Na operação de terminar conexão o cliente envia uma mensagem a dizer que pertende terminar a conexão e então a conexão é terminada e a thread que estava a tratar deste cliente no Data Server é destruida.

4 Backup de Dados

O servidor faz um backup da informação sempre que é desligado corretamente. O conteúdo da hashtable é copiado a um ficheiro chamado “backup.txt” usando a função “write” o que se assemelha a estar a escrever para uma socket.

De forma a garantir que mantemos toda a informação mesmo que exista um crash do sistema foi introduzido também um mecanismo de log. Ou seja sempre que há uma operação sobre

a hashtable, seja um write, um overwrite ou um delete essa operação é registada no ficheiro "backup.log".

De forma a não ter ficheiros de log muito extensos que possuem muita informação que é redundante foi ainda implementado um outro mecanismo que de intervalos de tempo regulares realiza um backup da hashtable para um ficheiro "backup.txt" garantindo desta forma ficheiros mais curtos numa inicialização do servidor, nesta altura o ficheiro de log é destruído e inicializado um novo.

Num servidor real esta operação podia ser implementada para ser realizada por exemplo no final de cada dia, numa hora em que houvesse pouca atividade no servidor, uma vez que a realização desta operação obriga tudo a parar.

Quando é iniciado , quer seja por ter ocorrido um crash ou por decisão do utilizador, a primeira coisa que o servidor faz é ler o ficheiro "backup.txt" e carregar o conteúdo todo para a hashtable e seguidamente realizar as operações existentes no ficheiro "backup.log" se o mesmo existir, só irá existir se o servidor não tiver saído ordeiramente.

5 Fault tolerance

Sendo este um projeto de "Programação de Sistemas"é necessário ter um mínimo de garantias para se ter um correto funcionamento perante situações anómalas como comportamento indesejado por parte do utilizador, falhas de comunicação, falhas do sistema, entre outras.

Deste modo, o grupo implementou um dos mecanismos lecionados nas aulas teóricas e laboratoriais para o caso de o servidor "crashar".

Esta solução gera um inconveniente, como um processo relança outro é perdida a "ligação"com o terminal associado ao programa inicial pois o processo que era "pai"(e tinha o terminal associado) pode-se tornar "filho"(que sendo um processo diferente deixa de estar associado ao terminal). Consequentemente, caso o processo original morra deixa-se de poder enviar o comando QUIT que controla o servidor.

O Front Server e o Data Server possuem uma thread que realiza operações para ver se o outro servidor está ou não vivo e relançar-lo no caso de o mesmo crashar. As duas threads seguem o mesmo processo. Primeiro, caso o processo seja filho fazem um waitpid e ficam bloqueadas até receber a notificação de morte do processo, no caso contrário prosseguem e tentam enviar sinais nulo ao processo que pretendem verificar e se não conseguirem entregar este sinal assumem que o processo de destino morto e voltam a inicializa-lo.

6 Conclusões e Comentários

Durante o desenvolvimento do projeto foi-se percebendo a importância dos mecanismos de sincronização, da utilização de protocolos de comunicação e a utilidade dos diferentes meios de comunicação entre processos, especialmente dos sockets que foi o que foi mais utilizado, juntamente com os mecanismos de paralelização, que é essencial na programação de qualquer sistema de complexidade moderada. Compreendeu-se também que é necessário recorrer a mutexes de forma a garantir consistência nos dados e o correto funcionamento do programa. De uma forma geral, o projeto contribui para consolidar a matéria lecionada nas aulas teóricas e laboratoriais.