

**Programação de Sistemas**

**Relatório**

**Projecto Final**

Docente responsável

João Nuno de Oliveira e Silva

Grupo 6:

Diogo Salgueiro nº72777

Ricardo Ferro nº 72870

Data de entrega: 25/05/2015

Instituto Superior Técnico

Departamento de Engenharia Electrotécnica e de Computadores

2º Semestre 2014/2015

# Arquitectura do Sistema

Servidor

Cliente

*admin*

*Relauncher*

Figura 1 – Arquitectura geral do sistema

## Implementação dos componentes

Na figura 1 encontram-se os quatro componentes que constituem o sistema de *chat* proposto no enunciado. Os componentes centrais são o servidor e cliente, sendo o *relauncher* e o *admin* componentes que auxiliam a correcta execução do servidor. Cada um destes componentes está implementado num ficheiro “.c” à parte.

As estruturas de dados utilizadas pelo servidor (lista de utilizadores, lista de mensagens e *log* de eventos) tem as respectivas funções em ficheiros “.c” à parte.

Os quatro componentes são executados simultaneamente e diversos mecanismos serão implementados para que a sua interacção ocorra sem falhas. Naturalmente, poderão existir diversos clientes ligados a servidor, mas apenas um *relauncher* e um terminal de administração (*admin*).

Relativamente à utilização de threads, apenas o servidor e o *relauncher* utilizaram este tipo de processamento em paralelo (explicado com mais detalhe nas secções 2 e 4). O cliente e o terminal *admin* correm sempre com um único *thread*.

## 1.2. Comunicação entre componentes

A comunicação entre o servidor e os clientes é realizada através de *sockets*, trocando mensagens sob a forma de *protocol* buffers de estrutura bem definida. Esta comunicação será explicada com maior detalhe na secção 5.

Os *protocol buffers* são utilizados também em FIFOs que comunicam entre *threads* do servidor ou entre o *servidor* e o *relauncher*. Novamente, o uso de um *protocol buffer* genérico permite uma uniformização dos protocolos de comunicação.

Foi criado um programa auxiliar (admin) que comunica com o servidor através de um *socket* bem definido, utilizando novamente mensagens com *protocol buffers*. Este programa realiza a ponte entre o utilizador e o servidor, dado que permite que este último peça ao servidor para realizar operações sem que tenha a linha de comandos a comunicar directamente. Para além disso, recebe as respostas do servidor aos pedidos enviados e imprime-as na linha de comandos.

# Implementações de paralelismo

Com intuito de realizar diversas operações no servidor, foram implementadas diversas *threads* executadas concorrencialmente.

Em primeiro lugar, é criada uma *thread* destinada a comunicar com cada cliente que se ligue ao servidor. Desta forma, é possível responder a pedidos diferentes enviados por clientes diferentes ao mesmo tempo, tendo especial atenção a existência de regiões críticas que só podem ser acedidas por uma *thread* de cada vez.

Para além das *threads* dedicadas à comunicação com os clientes, existe uma *thread* dedicada exclusivamente a efectuar *broadcast* de uma mensagem para todos os clientes.

Em relação ao controlo de falhas do servidor são criadas diversas threads que comunicam com o programa auxiliar *relauncher*. O número de threads dependerá do modo de execução do servidor e se este é ou não responsável em manter o *relauncher* em execução (ou o contrário). Este mecanismo será explicado mais detalhadamente na secção 4.

# Estruturas de dados do servidor

A lista de clientes encontra-se implementada no servidor sob a forma de uma lista ligada, contendo dentro de cada elemento o seu nome, o ID da thread responsável para atender a pedidos vindos desse utilizador e o *socket* que trata da ligação entre o servidor e o cliente. Cada elemento é acrescentado a cada nova ligação estabelecida e a lista é inicializada no arranque do servidor (sem elementos).

O armazenamento das mensagens de *chat* é realizado de uma forma semelhante (lista ligada), tendo no interior de cada elemento uma *string* com a mensagem e o ID respectivo. Este ID é criado com base num ID global para toda a lista, sendo posteriormente utilizado para identificar o intervalo de mensagens a apresentar na resposta a uma *query*.

O log de todas as operações do servidor é implementado sob a forma de um ficheiro de texto que é acedido em modo de *append*, ou seja, é aberto e qualquer evento novo é escrito no final do ficheiro. Foi implementado um mecanismo de *timestamp* que permite associar uma data e hora a cada evento. É feita também uma distinção entre a forma de inicialização do servidor (normal e através do *relauncher*).

## 3.1 Concorrência

Foram implementadas técnicas de gestão de concorrência de dados em módulos onde várias threads possam aceder e alterar os mesmos dados (log.c, login\_list.c e chat\_storage.c, ou seja, todas as estruturas de dados do servidor). Para tal, foram implementados mutexes em diversas funções destes módulos, tendo o cuidado de bloquear os dados o menor tempo possível.

Por exemplo, no caso do *broadcast*, é utilizada uma FIFO para enviar dados para a *thread*, utilizando um mutex numa região de código comum a todas as *threads* do servidor para cada cliente, impedindo que vários clientes escrevam para esta FIFO ao mesmo tempo. Desta forma, garante-se que as mensagens na FIFO são lidas sequencialmente pela *thread* de *broadcast*, libertando a thread do cliente mesmo no caso de o *broadcast* demorar mais tempo do que a escrita na FIFO.

# Tolerância a falhas

Um dos mecanismos mais importantes para garantir que o servidor tem uma execução correcta, mesmo quando a própria aplicação falha, é a implementação de um programa auxiliar encarregue de relançar o servidor caso este falhe (*relauncher*).

Dado que este *relauncher* pode também sofrer uma falha ao longo da sua execução, é necessário garantir que a sua execução também é moitorizada pelo servidor. No entanto, este não é o único factor a ter em conta, uma vez que é necessário que o servidor chama correctamente o *relauncher* (e vice-versa) consoante as falhas que possam ter acontecido.

Assim, começou por implementar-se o servidor a chamar o *relauncher* através de um fork(), tornando o *relauncher* um “filho” do servidor. Uma vez que é possível que um “pai” saiba que um filho termine a sua execução, é simples relançar o *relauncher* cada vez que este termine inesperadamente.

No entanto, o próprio servidor pode terminar repentinamente, sendo necessário um mecanismo que permita ao *relauncher* (filho) saber o estado de execução do pai. Foi implementada uma FIFO, que é escrita periodicamente pelo servidor e acedida (periodicamente) pelo *relauncher*. Se o *relauncher* deixar de receber dados do servidor, poderá indicar que este falhou. Desta forma, o *relauncher* terá que criar uma nova instância do servidor, passando o servidor a ser “filho” do *relauncher*. A partir deste ponto, as tarefas invertem-se, passando o *relauncher* a controlar a execução do filho e a informá-lo que está em execução através da mesma FIFO.

Se o *relauncher* falhar neste modo de execução, as tarefas invertem-se novamente, passando o servidor a ter o *relauncher* como “filho”. Assim, ambos os processos mantêm-se a par um do outro, garantindo que não deixam de executar devido a qualquer falha.

Ao nível das threads necessárias para este controlo, basta explicitar que quando um dos processos é pai, tem duas threads em execução: uma thread para controlar a execução do filho e outra para a escrita na FIFO. Por sua vez, o filho só tem uma thread que lê a FIFO para saber o estado de execução do pai. Se o pai falhar, o “filho” assume o papel de “pai” e o número e função das threads inverte-se.

É necessário ter em conta a distinção entre uma falha de execução do servidor e a sua saída ordeira através do comando QUIT fornecido pelo utilizador. Através da FIFO e dos códigos de retorno de fim de execução foi possível fazer esta distinção.

# Comunicação

A comunicação entre o servidor e os vários clientes é efectuada através de um *socket* TCP, dentro do qual são trocados *protocol buffers* com uma estrutura bem definida. O *protocol buffer* tem a estrutura descrita em seguida:

message Message{

required int32 type = 1;

optional string username = 2;

optional string chat = 3;

optional int32 query\_id\_min = 4;

optional int32 query\_id\_max = 5;

optional string query\_resp = 6;

optional string log\_resp = 7;

}

O primeiro campo é sempre incluído em qualquer tipo de mensagem trocada e tem como principal função identificar o tipo de mensagem (tal como o nome sugere). No ficheiro message.h estão descritos todos os valores possíveis para este campo, dado que este ficheiro é utilizado por quase todos os componentes do sistema.

Os restantes campos são utilizados consoante o tipo de mensagem a ser trocada. Desta forma, ao utilizar apenas um tipo de *protocol buffer* é possível garantir uma melhor unificação entre todos os componentes do sistema.

No arranque do servidor, é reservado o porto 3000 para receber comunicações enviadas por clientes que se queiram registar. Ao receber uma nova ligação, o servidor cria uma nova *thread* responsável pela comunicação com cada cliente, atribuindo um novo *socket*. Dentro desta nova *thread*, o servidor recebe os comandos de LOGIN, CHAT, QUERY e DISC enviadas pelo cliente. Por uma questão de coerência e controlo de execução, o servidor responde a todas as mensagens enviadas pelos clientes.

# Testes unitários

Com vista a testar um dos componentes que constituem este sistema, sem recorrer à implementação final do mesmo, recorreu-se à ferramenta *Ceedling*. Esta ferramenta contém, entre outros componentes, uma componente que permite realizar testes unitários (*unit tests*) a funções presentes em ficheiros exteriores aos ficheiros principais do sistema.

Desta forma, é possível testar isoladamente o comportamento de certos componentes do sistema. Após a instalação das dependências necessárias, foi possível criar um projecto no *Ceedling* dedicado apenas ao teste das funções presentes no módulo “chat\_storage.c”. Esta componente apresenta a definição da estrutura de armazenamento das mensagens de chat (lista ligada) e funções que manipulam a mesma. É necessário salientar que no sistema final, este módulo tem *mutexes* e uma sintaxe ligeiramente diferente (para melhor integração com os componentes finais).

Dado que estão criadas 4 funções (*cs\_init*, *create\_message*, *add\_message* e *query*) foi criado um ficheiro de teste para cada uma delas, como se explicita em seguida:

* *test\_cs\_init.c*: teste dedicado a verificar se a função que cria uma lista vazia é excutada correctamente.
* *test\_create\_message.c*: teste dedicado a avaliar a criação de cada elemento da lista, verificando a alocação de cada um dos seus campos. A função *create\_message* é invocada pela função *add\_message*.
* *test\_add\_message.c*: teste com intuito de verificar a correcta adição de mensagem à lista, verificando também se esta não excede o tamanho máximo.
* *test\_query.c*: este último teste testa implicitamente todos os testes anteriores, focando a sua atenção na obtenção de resultados de *queries* com diferentes parâmetros. Este teste assume que os valores são verificados pelo programa de cliente, nunca chamando a função com índices de pesquisa trocados ou com o primeiro índice igual a zero. Desta forma, verifica-se se o primeiro índice não é superior ao número de mensagens existentes na lista e diferentes opções que poderão ser realizadas pela função e respectiva impressão das mensagens.

Todo o código necessário à realização de cada um dos testes encontra-se na pasta criada especialmente para o efeito, bastando executar o comando “*rake test:all*” a partir da raiz da directoria.

# Conclusões e comentários