Programação de Sistemas

Relatório do projecto

Bernardo Gomes, 75573 Mahomed Gani, 75796

20 de Maio de 2015

Conteúdo

1 Descrição do problema

O objectivo do projecto é a implementação de um sistema servidor-cliente que forneça um serviço de *chat* simples.

O servidor e o cliente utilizam TCP IP *sockets* como forma de comunicação, sendo disponibilizados ao cliente as seguintes formas de comunicação:

- envio da sua identificação através de um comando de *login*;
- envio de mensagens de *chat*;
- pedido de mensagens antigas que o servidor terá armazenado no passado.

Adicionalmente, o cliente terá a hipótese de realizar o pedido de desconexão, que no nosso projecto foi implementado como tipo de mensagem, mas que poderia ter sido tratado de outra forma.

Relativamente ao servidor, este deverá disponibilizar mecanismos tal que seja possível a ligação de múltiplos clientes, e que as suas mensagens sejam interpretadas de forma eficaz e sejam produzidas as respectivas respostas ou acções.

2 Resumo da abordagem ao problema

Numa fase inicial, implementou-se as funções básicas de leitura do teclado quer do lado do servidor, quer do lado do cliente, na medida em que sem o programa conhecer os comandos de comunicação que o utilizador pretende utilizar, não terá o funcionamento espectável.

Tendo o cliente apenas as funcionalidades descritas na secção anterior, deverão existir duas secções de código a serem executadas paralelamente: uma secção que deverá esperar por um comando introduzido pelo utilizador e que no caso de ser verificado como comando válido este seja enviado para o servidor, e um que deverá apenas esperar por uma mensagem de resposta do servidor (que pode corresponder a um *broadcast* de mensagens ou a respostas de comandos anteriores), e imprimi-la na interface.

Relativamente ao servidor, foi necessário abordar o problema de forma diferente por este ter requisitos diferentes. O facto de lhe ser exigido o processamento das mensagens vindas do lado de clientes, de ser necessário ligar-se a mais do que um cliente à medida que as circunstâncias assim o exijam, a necessidade de fazer *broadcast* de todos os *chats* recebidos e a possibilidade de o administrador do sistema poder pedir o registo de toda a actividade inerente ao funcionamento do programa e forçar o seu encerramento, tudo isto de forma simultânea, exige uma simultaneidade de pelo menos quatro troços de código. Para solucionar este problema, foram utilizadas *threads*, sendo o meio de comunicação entre elas uma estrutura conhecida por todas. No caso do *broadcast*, de forma a garantir uma certa ordem de envio face à ordem de chegada das mensagens ao servidor, utilizou-se um *FIFO* entre as *threads* que lidam com a chegada de mensagens de um certo cliente (uma *thread* para cada cliente) e a que lida com a divulgação das mensagens para todos.

3 Arquitectura do sistema

3.1 Módulos

Além dos ficheiros que originam os executáveis do projecto, foram criados os seguintes módulos:

- "structures.h"que contém todas as estruturas de dados necessárias para o funcionamento do servidor (explicação mais detalhada na secção seguinte do relatório;
- "login_data.h"onde estão incluídas as funções de inserção de um novo cliente com *login* válido, remoção de um cliente que pretenda desconectar-se e uma função que encontra o *username* do cliente que mandou uma mensagem do tipo *chat*;
- "server.h"que inclui as *threads* do funcionamento do servidor e a função do *server* propriamente dito que lança estas mesmas *threads*.
- "storage.h"cujas funções nele incluídas referem-se aos mecanismos de armazenamento de mensagens do tipo *chat*, aos comandos a ele associados e à validação de novos *usernames*.
- "log.h"cujos *headers* nele contidos são relativos às funções de leitura e escrita no ficheiro onde toda a actividade do servidor é registada. Quando se faz *kill* do processo responsável pelo servidor, a actividade do mesmo foi terminada de forma irregular, pelo que o processo responsável por o "ressuscitar"chama uma função, também contida neste módulo, cuja função será de encontrar o último identificador escrito no ficheiro.

3.2 Estruturas de dados

Tendo em consideração a pouca complexidade do código do cliente face ao escrito para o servidor, as estruturas de dados utilizadas para este último foram colocadas num ficheiro de *header* isolado enquanto que a única estrutura utilizada para o cliente está incluída no único ficheiro de *source* do mesmo.

3.2.1 Cliente

Para este caso, tendo em conta a utilização de *threads* para solucionar o paralelismo das funções desempenhadas pelo programa, optou-se pela utilização de uma estrutura "variables_client"que funciona como variável global que contém a informação do *socket* (*file descriptor* correspondente e os endereços necessários à comunicação) através do qual são feitas as comunicações.

3.2.2 Servidor

Tendo em vista a necessidade de fazer as *threads* conhecerem *flags* relativas a comandos dados no teclado, a necessidade de armazenar as mensagens que os utilizadores vão enviando e a funcionalidade de *broadcast*, fez-se uma abordagem ao problema com uma estrutura genérica do servidor ("variables"), que alberga todos estes dados.

Esta estrutura, irá conter os seguintes campos:

- uma *flag* ("flag_quit") a indicar que o administrador do *chat* pretende encerrá-lo;
- uma estrutura "client_database", que irá incluir o número de clientes que estão conectados com *login* feito, com sucesso, e uma base de dados (estrutura "users") de todos os *usernames* e *file descriptors* dos utilizadores mencionados anteriormente;
- um número inteiro "n_stored_messages"que guarda o número de mensagens recebidas;
- um número inteiro "n_aux_message_storage"que guarda o número de conjuntos de vinte mensagens que estão armazenados (arredondado por excesso). Na prática irá contar o número de elementos que estão na lista iniciada pelo ponteiro que será falado no ponto seguinte;
- um ponteiro para uma estrutura do tipo "server_database", cujo apontado irá conter um *array* de vinte mensagens. Este *item* da estrutura constitui a base de dados das mensagens recebidas;
- um contador para saber qual o número de entradas no ficheiro de *log*.

Além da estruturas de dados referidas anteriormente, é ainda utilizada a "buffer_aux", cujos parâmetros são a mensagem que o utilizador pretende transmitir e o *username*. "buffer_aux"será utilizada na comunicação entre *threads* (dos clientes com a de *broadcast*).

3.3 Mensagens do tipo *Protocol Buffer*

As mensagens de *Protocol Buffer* foram utilizadas no contexto de transmissão de mensagens servidor-cliente.

Tendo em vista os comandos de *input* consoante o tipo de mensagem a enviar, considerou-se uma mensagem do tipo "CLIENT_MESSAGE"cujo único elemento *required* será um inteiro que o programa irá interpretar para saber qual o tipo de mensagem que foi enviado. Assim para a variável do campo *type*, o programa irá interpretar a informação da seguinte forma:

1. tipo LOGIN
2. tipo DISC
3. tipo CHAT
4. tipo QUERY

Consoante o tipo lido, assim o servidor irá ler a mensagem propriamente dita ao(s) campo(s) adicional(ais) correspondente(s). No caso de ser do tipo 2 não é necessário mais nenhum campo.

Para a comunicação em sentido contrário, utilizou-se uma mensagem "RESPONSE"cujo campo obrigatório será o mesmo da mensagem explicada anteriormente, sendo o valor interpretado como sendo uma resposta ao tipo correspondente à enumeração anterior.

4 Troca de mensagens

4.1 Cliente

No binário do cliente, lidasse com as mensagens de forma a que o utilizador possa enviar comandos (ou mensagens) e simultaneamente receber o *broadcast* de outros utilizadores. Para que isto seja possível, a utilização de dois processos ou de duas *threads* torna-se necessária.

Tendo em conta que num serviço de *chat*, o modo de cliente deve funcionar como um todo, isto é, o utilizador deve poder enviar e receber mensagens, não será necessário assegurar que quando uns dos serviços termine, o outro se mantenha. Tendo também em consideração que será mais simples a implementação do programa através de variáveis globais que determinem se o cliente pretende fechar a sua sessão e que guardem informações relativas à ligação do *socket*, optou-se pela utilização de duas *threads*. Assim, a primeira lidará com a recepção de mensagens de outros clientes e a segunda será encarregue de enviar para o servidor o comando pedido pelo utilizador.

Sendo o binário executado no terminal, e tendo em vista uma utilização mais realista em oposição a uma abordagem meramente académica, colocou-se nesta secção do programa uma lista dos comandos possíveis e o seu significado.

4.1.1 Envio de mensagens

Do lado do utilizador, são apenas válidos quatro tipos de *requests*, já descritos anteriormente na secção da descrição do problema. Como tal, antes do envio de mensagens, é feita internamente uma validação do comando inserido no terminal. Apenas quando esta validação é realizada com sucesso, a *thread* responsável pelo envio da *request* procede à preparação da mensagem que o utilizador deu instrucção para envio. Neste caso, é preenchido o campo que indica o tipo da mensagem e o campo opcional com a mensagem em si.

No caso de a mensagem enviada ser do tipo "DISC", além de ser enviada para o servidor como indicação que o cliente pretende desconectar-se, fecha-se o *socket* e seguidamente o programa em si.

4.1.2 Recepção de mensagens

O tratamento das mensagens recebidas é feito de forma dual ao envio. Sendo a estrutura das mensagens de resposta constituída por um campo que indica o tipo de mensagem que se recebeu, após a recepção da mesma, é feita uma filtragem do tipo a que corresponde. Consoante o valor do número inteiro do campo *type* da resposta recebida, assim será feito o acesso ao campo opcional correspondente. Assim, no caso de serem recebidas mensagens do tipo:

- "LOGIN", será explicitado ao utilizador se este foi efectuado o não com sucesso;
- "CHAT"significa que a mensagem corresponde a um *broadcast* de uma mensagem que o servidor recebeu, pelo que imprimirá o nome do cliente que a enviou e a mensagem a transmitir;
- "QUERY", seram impressas as mensagens dentro da gama de id's pedidos, separadas por identificadores.

Acrescentamos ainda que não fará sentido especificar mensagens do tipo "DISC"na medida em que a partir do momento em que o cliente envia ao servidor a informação que se vai desligar, acaba mesmo por o fazer sem ser necessário esperar uma resposta válida. Cabe nesta altura ao servidor processar este pedido dentro da sua base de dados.

4.2 Servidor

Na função do servidor, deve ser possível lidar com as mensagens recebidas para que estas sejam processadas e possa ser gerada uma resposta.

Numa primeira fase, a *thread* que lida com o cliente em questão, fica bloqueada até receber uma mensagem do cliente, o que acabará por acontecer, a menos que o utilizador tenha apenas como único objectivo abrir o executável do cliente. Após a leitura, a mensagem será convertida e será lidada consoante o campo *required* recebido da seguinte forma:

1. :

   -verifica se já existiu um *login* válido. No caso de este ainda não se ter verificado, continua a sequência a seguir apresentada;

   -verifica se o *username* escolhido já foi escolhido;

   -no caso de ser novo, coloca a *flag* que regula o *login* válido a 1, insere o cliente na base de dados e coloca na variável de resposta uma indicação de que a operação escolhida foi realizada com sucesso;

   -no caso de o *username* escolhido já existir, é colocado na variável de resposta a indicação de que o *login* não foi realizado com sucesso dando a indicação de que o utilizador deverá tentar novamente;

   -é enviada a resposta.

2. recebe a indicação de que o utilizador pretende terminar sessão pelo que numa primeira fase verifica se foi feito *login*. No caso de ter sido feito, remove o utilizador da base de dados. A *thread* do cliente é fechada.

3. Apenas no caso de já ter sido feito *login*, é procurado na base de dados o nome do utilizador. Posteriormente, a informação é copiada para uma estrutura de forma a que a mensagem seja transmitida a todos os utilizadores em rede com *login* feito.

4. No caso de o *login* já ter sido efectuado, é procurado na base de dados as mensagens compreendidas entre os id's indicados, são concatenadas com identificadores, e posteriormente enviada a resposta.

Em qualquer um dos casos, é escrita uma mensagem no ficheiro de *log*.

O *broadcast* é feito pela leitura das mensagens contidas no FIFO e a mensagem é distribuída por todos os membros contidos na base de dados.

5 Armazenamento de mensagens do tipo chat

Tendo em vista uma ligeira optimização na procura das mensagens, decidiu-se armazená-las em lista, onde cada elemento desta contém um bloco de vinte

mensagens. Como tal, foi necessário proceder a funções mais complicadas para a identificar o local a colocá-las e o local a lê-las sempre que necessário.

As funções criadas foram as seguintes:

- "check_add_new_vector"que verifica se é necessário adicionar um novo vector de armazenamento de vinte mensagens à lista;
- "add_message_vector_to_database"que efectivamente adiciona este vector ao armazenamento;
- "get_write_position"que retorna o número de elementos da lista que é necessário percorrer e qual a posição do vector deste elemento ao qual é necessário aceder para armazenar nova mensagem à base de dados;
- "write_position"que copia a mensagem para o elemento retornado pela função anterior;
- "get_index"que tem um fim semelhante à função "get_write_position"no entanto serve para aceder a mensagens já anteriormente armazenadas.

Com auxílio destas ferramentas, é possível uma leitura e escrita na base de dados bastante mais optimizada relativamente a uma lista simples.

5.1 Módulos de teste

Com vista a testar as funções de acesso, criou-se um ficheiro .c para verificar as saídas das funções consoante as entradas de cada função.

Desta forma, para a função de inicialização, após se chamar a rotina, o número de mensagens armazenadas deve ser nulo bem como o número de elementos da lista.

Relativamente ao teste de adição de novos elementos à lista, testa-se o valor de retorna da função no caso de armazenar 39, 40, 41 e 2 mensagens. Sendo que apenas deve retornar que se deve adicionar um novo elemento quando o número de mensagens armazenadas é 40.

O teste relativo ao retorno dos locais onde as mensagens estão armazenadas baseia-se no mesmo princípio das anteriores.

6 *Performance/Known issues*

Durante a utilização do programa, a ligação de múltiplos clientes torna-se possível pela detecção dos mesmos na instrucção de *accept* e lançamento de uma nova *thread* cada vez que o programa desbloqueia a instrucção anterior. Esta solução, apesar de simplificar bastante a escrita de código e o paralelismo e comunicação dentro do servidor, limita o número de clientes, devido ao facto de o sistema operativo ter um limite de memória para cada processo. Tendo em consideração que o *chat* pretendido tem apenas fins académicos, decidimos simplificar a escrita do código e ter em conta que o número de utilizadores não irá exceder a memória destinada para o processo do servidor.

Devido à implementação realizada para o armazenamento de mensagens do tipo *chat* descrita anteriormente, o acesso a este armazenamento aquando da recepção de mensagens *query*, fica bastante mais optmizado durante o início do

programa pelo facto de termos passado o acesso de uma complexidade O(N) (lista simples) para um acesso que tem uma procura já direccionada e que consoante o id da mensagem faz a procura dentro de blocos de vinte mensagens. No entanto, quando a utilização do programa começa a receber um número já bastante elevado de mensagens, esta optimização acaba por ser irrelevante.

Assume-se que as mensagens enviadas pelos clientes não excedem os 100 caracteres. Desta forma, se o utilizador escrever uma mensagem com tamanho superior, apenas o início da mensagem é enviada.

Relativamente à recepção de mensagens por parte do cliente, sendo a encriptação feita por *protocol buffers*, e sendo mais simples ter apenas um tipo de mensagem deste tipo para todos os comandos/resposta aos comandos, utilizou-se um *buffer* com tamanho [10000], de forma a que a resposta ao comando de pedido "QUERY"seja realizável até noventa e nove mensagens (é necessário que haja memória para os outros campos da mensagem).

Acrescentamos ainda que a administração do sistema é feita de forma a que os clientes não tenham de esperar tempo praticamente nenhum pelas respostas e/ou *broadcasts*. No entanto, no caso de uma utilização mais global e aplicando a situações mais práticas, situações de elevada utilização com vários tipos de pedidos (QUERY e LOGIN) iriam certamente atrasar bastante o sistema devido à necessidade de aceder a posições de memória partilhadas. A actualizações destas posições não iram ter qualquer tipo de problemas.

No caso de se realizarem *kills* de processos, implementou-se o *relaunch* do processo do servidor. Neste caso, a ligação aos clientes é perdida, sendo necessário que estes corram novamente o programa e realizem novamente *login*. Se se fizer *kill* do processo que administra o funcionamento do servidor e o coloca em cima novamente caso morra, os clientes não notarão qualquer tipo de perturbação no sistema. No entanto, o administrador do *chat* irá perder qualquer tipo de acesso ao comandos de LOG e QUIT, pelo facto de o processo filho passar a correr em *background* na linha de comandos. Este facto irá também originar a impossibilidade de sair do programa de forma ordeira (com o comando QUIT) e consequentemente irá haver memória que não ficará limpa e será necessário fechar o FIFO de comunicação criado manualmente da directoria "tmp"do computador antes de se poder lançar novamente o *chat*.

6.1 *Bugs*

Durante a fase de aperfeiçoamento do código, deparámo-nos com a existência de dois *bugs*:

1. a introdução de o comando de QUIT sem a existência de clientes conectados no passado não é entendida pelo programa (em princípio durante uma execução normal de um serviço *chat* este problema nunca irá ocorrer);

2. a introdução do comando de LOG bloqueia futuras conecções de clientes.

Pela análise do código que foi escrito e tendo em conta o paralelismo e a divisão de tarefas dentro do servidor, não nos foi possível detectar a localização dos *bugs*, tendo a carga de trabalho das outras UC's também contribuído para o défice de tempo investido na correcção dos mesmos.

7 Soluções de *Fault tolerance*

Relativamente ao *Fault tolerance*, implementou-se o registo de actividade do servidor com recurso a um ficheiro "log.txt"que armazena todo o tipo de actividade realizada.

Não sendo necessário que a base de dados das mensagens e do registo dos clientes seja persistente após fecho do servidor, não se fez nenhum armazenamento que se mantenha após o *kill* do servidor.

8 Considerações finais

Antes da implementação do projecto, foi pensada qual a estrutura que o programa iria ter e a forma de implementar as trocas de mensagens que teriam de existir entre o servidor e os clientes e os mecanismos de comunicação internos ao servidor.

Foi observado numa primeira análise que a implementação do projecto com 4 tipos de mensagens diferentes e as suas respectivas respostas seria bastante complicada, pelo que foi necessário repensar essa parte das comunicações, tendo sido feita a implementação atrás descrita no relatório.

As funcionalidades de *relaunch* e de leitura das respostas por parte do cliente ficaram um pouco aquém daquilo que gostaríamos, no entanto devido à carga de trabalho de outras UC's não nos foi possível aperfeiçoar estas funcionalidades, pelo foram realizadas as soluções atrás descritas com mecanismos de protecção adequados de forma a que não ocorra *segmentation fault*.

Infelizmente, devido a falta de tempo, quando é dado o comando de *log* no lado do servidor não nos foi possível resolver a não conexão de futuros clientes e a saída normal do servidor caso não haja uma ligação de um cliente antes de ser dado o comando de QUIT.