Ciências ULisboa | Informática

2017/18

Sistemas Distribuídos

Projeto 2

1. Descrição geral

A componente teórico-prática da disciplina de sistemas distribuídos está divida em quatro projetos, sendo que a realização de cada um deles é necessária para a realização do projeto seguinte. Por essa razão, **é muito importante que consigam ir cumprindo os objetivos de cada projeto, de forma a não hipotecar os projetos seguintes**.

O objetivo geral do projeto será concretizar um serviço de armazenamento de pares chave-valor similar ao utilizado pela *Amazon* para dar suporte aos seus serviços Web [1], utilizando para tal uma **tabela *hash*** [2]. No projeto 1 foram definidas estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na tabela, bem como para implementar a tabela recorrendo à técnica de *coalesced chaining* para resolver o problema das colisões de chaves no mesmo índice da tabela.

No projeto 2 os objetivos são a implementação das funções necessárias para serializar e de-serializar estruturas complexas em mensagens (1 semana), a concretização de um conjunto de tabelas *hash* num servidor, e a implementação de um cliente com uma interface similar à das funções que interagem com as tabelas na memória (2 semanas). Isto implica que:

1. através do cliente, o utilizador irá invocar operações, que serão transformadas em mensagens e enviadas pela rede até ao servidor;
2. este, por sua vez interpretará essas mensagens, e;
    a. realizará as operações correspondentes nas tabelas locais;
    b. enviará posteriormente a resposta transformada em mensagem, ao cliente;
3. por sua vez, o cliente interpretará a mensagem de resposta, e;
4. procederá de acordo com o resultado, ficando de seguida pronto para a próxima operação.

A concretização do cliente e do servidor será efetuada combinando o código desenvolvido nos vários módulos com as técnicas para comunicação por *sockets* TCP (aulas TP da segunda semana do projeto 2). Espera-se uma grande fiabilidade por parte do servidor, portanto não pode haver condições de erro não verificadas ou gestão de memória ineficiente a fim de evitar que este sofra um *crash*, o que deixaria todos os clientes sem as tabelas partilhadas (no caso da Amazon, se o serviço que mantém as cestas de compras dos clientes não funciona, a empresa não vende, perde milhares de dólares por hora… e os programadores são despedidos!).

2. Descrição específica

O projeto 2 consiste na concretização em C de:

1. um módulo que define uma estrutura de dados (mensagem) e funções para codificação e descodificação de estruturas complexas em mensagens, para possibilitar a comunicação entre cliente e servidor no ponto seguinte;
2. dois programas, a serem executados da seguinte forma:
    a. **table-server** <port> <table1_size> [<table2_size> …]
    <*port*> é o número do porto TCP ao qual o servidor se deve associar (fazer *bind*).
    <*tableX_size*> é o tamanho da tabela número X a criar no servidor.
    b. **table-client** <*server*>:<*port*>
    <*server*> é o endereço IP ou nome do servidor da tabela.
    <*port*> é o número do porto TCP onde o servidor está à espera de ligações.

Para cada um dos módulos (em 1 e 2), é fornecido um ficheiro *.h* com os cabeçalhos das funções, que **não pode ser alterado**. As concretizações das funções definidas nos ficheiros *X.h* devem ser feitas num ficheiro *X.c*, utilizando os algoritmos e métodos que o grupo achar convenientes. Se o grupo entender necessário, ou se for pedido, também pode criar um ficheiro *X-private.h* para acrescentar outras definições, a incluir no ficheiro *X.c*. Os ficheiros *.h* apresentados neste documento bem como alguns testes para as concretizações realizadas, serão disponibilizados na página da disciplina.

Adicionalmente, serão fornecidos de antemão alguns dos ficheiros .c que os grupos têm de desenvolver. Nesses ficheiros existirão comentários nas definições de estruturas e funções a implementar, como sugestão da sequência de tarefas que devem ser executadas.

2.1. *Marshaling* e *unmarshaling* de mensagens

Este módulo do projeto consiste em transformar uma estrutura de dados complexa num formato que possa ser enviado pela rede (isto é, convertê-la num formato "unidimensional"), e vice-versa. O ficheiro *message.h* define as estruturas e as funções a serem concretizadas neste módulo.

Note-se que neste módulo vão ser usadas uniões [3, 4]. No caso da *struct message_t* abaixo, a ideia é que, dependendo do código da operação representada na mensagem (campo *opcode*), uma variável de tipo diferente possa ser utilizada.

```
#ifndef _MESSAGE_H
#define _MESSAGE_H

#include "data.h"
#include "entry.h"

#define _SHORT 2
#define _INT 4

/* Define os possíveis opcodes da mensagem */
#define OC_SIZE         10
#define OC_UPDATE       20
#define OC_GET          30
#define OC_PUT          40
#define OC_COLLS        50

/* Define códigos para os possíveis conteúdos da mensagem */
#define CT_RESULT       10
#define CT_VALUE        20
#define CT_KEY          30
#define CT_KEYS         40
#define CT_ENTRY        50

/* Estrutura que representa uma mensagem genérica a ser transmitida.
 * Esta mensagem pode ter vários tipos de conteúdos. */
struct message_t {
      short opcode;     /* código da operação na mensagem */
      short c_type;     /* tipo do conteúdo da mensagem */
      short table_num; /* número da tabela */
      union content_u {
            int result;
            struct data_t *data;
            char *key;
            char **keys;
            struct entry_t *entry;
      } content;     /* conteúdo da mensagem */
};
```

```
/* Converte o conteúdo de uma message_t num char *, retornando o tamanho do
 * buffer alocado para a mensagem serializada como um array de bytes, ou -1
 * em caso de erro.
 * A mensagem serializada numa sequência de bytes, deve ter o seguinte
 * formato:
 *
 *  OPCODE          C_TYPE           TABLE_NUM
 * [2 bytes]       [2 bytes]         [2 bytes]
 *
 * a partir daí, o formato difere para cada tipo de conteúdo (c_type):
 *
 * CT_ENTRY:      KEYSIZE(KS)          KEY      DATASIZE(DS)          DATA
 *                 [2 bytes]         [KS bytes]   [4 bytes]         [DS bytes]
 *
 * CT_KEY:        KEYSIZE(KS)          KEY
 *                 [2 bytes]         [KS bytes]
 *
 * CT_KEYS:         NKEYS            KEYSIZE(KS)    KEY      ...
 *                [4 bytes]          [2 bytes]  [KS bytes]  ...
 *
 * CT_VALUE:      DATASIZE(DS)         DATA
 *                 [4 bytes]         [DS bytes]
 *
 * CT_RESULT:      RESULT
 *                [4 bytes]
 *
 * Notar que o `\0´ no fim da string e o NULL no fim do array de
 * chaves não são enviados nas mensagens.
 */
int message_to_buffer(struct message_t *msg, char **msg_buf);

/* Transforma uma mensagem no array de bytes, msg_buf, para
 * uma struct message_t*
 */
struct message_t *buffer_to_message(char *msg_buf, int msg_size);

/* Liberta a memoria alocada na função buffer_to_message
 */
void free_message(struct message_t *msg);

#endif
```

2.2. Programas cliente e servidor

2.2.1 Cliente

O cliente usa as funcionalidades definidas nos módulos já desenvolvidos, juntamente com duas partes adicionais:

- Aplicação cliente interativa (table_client.c) com a função main()
- Biblioteca de comunicação (network_client.h/c)

table_client.c

A aplicação cliente a realizar é um programa interativo simples, que depois de executado aceita um comando (uma linha) do utilizador no *stdin*, invoca a função necessária da biblioteca de comunicação (subsecção seguinte), imprime a resposta no ecrã, e volta a aceitar o próximo comando.

Depois de receber o comando do utilizador, este módulo tem então de preencher uma estrutura *message_t* com o *opcode* (tipo da operação) e os campos usados de acordo com o conteúdo requerido pela mensagem. De seguida vai passar esta estrutura para o módulo de comunicação,

recebendo depois uma resposta deste módulo (ver próxima secção). Cada comando vai ser inserido pelo utilizador numa única linha, havendo as seguintes alternativas:

- put <número da tabela> <key> <data>
- get <número da tabela> <key>
- update <número da tabela> <key> <data>
- size <número da tabela>
- collisions <número da tabela>
- quit

Note que a aplicação cliente vai usar o campo 'data' da estrutura `data_t` do projeto 1 para guardar a *string* <data> introduzida pelo utilizador nos comandos put e update (que pode conter espaços, i.e., é a *string* completa após a chave <key>). No entanto, todos os restantes módulos (*network_client.c* e servidor) devem suportar dados arbitrários neste campo (qualquer vetor de bytes).

Dica: uma boa forma de ler e tratar os comandos inseridos é usando as funções *fgets* e *strtok*.

Biblioteca de comunicação com o cliente: network_client.c

O módulo *network-client.c* vai serializar a mensagem, utilizando a função *message_to_buffer* definida anteriormente, enviá-la ao servidor, e esperar pela resposta. A biblioteca de comunicação *network_client.c* tem a seguinte interface:

```
#ifndef _NETWORK_CLIENT_H
#define _NETWORK_CLIENT_H

#include "message.h"

struct server_t; //a definir pelo grupo em network_client-private.h

/* Esta função deve estabelecer a ligação com o servidor;
 *  - address_port é uma string no formato <hostname>:<port>
 *    (exemplo: 10.10.10.10:10000)
 *  - retorna toda a informacão necessária (e.g., descritor da
 *    socket) na estrutura server_t
 */
struct server_t *network_connect(const char *address_port);

/* Esta função deve
 * - Obter o descritor da ligação (socket) da estrutura server_t;
 * - enviar a mensagem msg ao servidor e receber dele uma resposta;
 * - retornar a mensagem obtida como resposta ou NULL em caso
 *   de erro.
 */
struct message_t *network_send_receive(struct server_t *server,
                                       struct message_t *msg);

/* A função network_close() fecha a ligação estabelecida por
* network_connect(). Se esta alocou memória, a função deve liberta-la
*/
int network_close(struct server_t *server);

#endif
```

As mensagens na rede, tanto os pedidos do cliente como as respostas do servidor, devem usar o formato definido no módulo message.h. Para facilitar a receção de mensagens, devem ser executados os três passos seguintes sempre que se envia uma mensagem:

a) usar *message_to_buffer* para transformar a estrutura com a mensagem num *buffer* e determinar o número de bytes da mensagem serializada;
b) enviar primeiro um inteiro (4 bytes) em formato de rede indicando ao interlocutor o tamanho da mensagem serializada que será enviada de seguida, e;
c) enviar o *buffer* (que será interpretado pelo interlocutorr através de *buffer_to_message*).

2.2.2 Servidor

O servidor a implementar usa todas as funcionalidades desenvolvidas anteriormente, além da aplicação servidor (*table_server.c*) com a função *main()*.

table_server.c

O servidor concretiza um conjunto de tabelas que podem ser acedidas no porto definido na linha de comando, e deve aceitar ligações neste porto e em qualquer interface. Os servidores deverão suportar apenas um cliente de cada vez. Como o servidor apenas necessita de tratar um pedido de cada vez, não será necessário nesta fase recorrer à subdivisão do programa em *threads* ou processos filho (e.g., através da chamada de sistema *fork()*).

Observações

Algumas observações e dicas úteis:

- Recomenda-se a criação de funções *read_all* e *write_all*, que vão receber e enviar *arrays* de bytes completos pela rede (lembrar que as funções *read*/*write* nem sempre lêem/escrevem tudo o que pedimos).
  Um bom sítio para concretizar essas funções é num módulo separado, a ser incluído pelo cliente e servidor, ou no *message-private.h* (concretizando-as no *message.c*).
- Usar a função *signal()* para ignorar sinais do tipo SIGPIPE, lançados quando uma das pontas comunicantes fecha o *socket* de maneira inesperada. Isto deve ser feito tanto no cliente como no servidor, evitando que um programa termine quando a outra parte é desligada.

.
Para cada mensagem enviada pelo cliente, deverá haver uma mensagem de resposta correspondente. A tabela seguinte define como deve ser configurada a estrutura `message_t` (omitindo o membro `table_num`), para cada uma das invocações feitas pelo cliente e das respostas dadas pelo servidor.

COMANDO CLIENTE	OPCODE PEDIDO	OPCODE RESPOSTA	C_TYPE PEDIDO	CT_TYPE RESPOSTA
PUT	OC_PUT	OC_PUT+1	CT_ENTRY	CT_RESULT
GET	OC_GET	OC_GET+1	CT_KEY	CT_VALUE (chave existe ou não) CT_KEYS (chave = "*")
UPDATE	OC_UPDATE	OC_UPDATE+1	CT_ENTRY	CT_RESULT
SIZE	OC_SIZE	OC_SIZE+1	-	CT_RESULT
COLLISIONS	OC_COLLS	OC_COLLS+1	-	CT_RESULT

Caso algum dos pedidos não possa ser atendido devido a um erro, o servidor vai retornar {OC_RT_ERROR, CT_RESULT, *errcode*}, onde *errcode* é o código do erro retornado ao executar a operação na tabela do servidor (em geral, -1). Para tal, é necessário adicionar esse novo *opcode*, por exemplo com valor 99, no vosso *message-private.h*.

Note que o caso onde uma chave não é encontrada no *get* não se caracteriza como erro. Quando isso ocorre, o servidor deve retornar de acordo com a tabela acima, mas definindo um `data_t` com `size=0` e `data=NULL`.

3. Entrega

A entrega do projeto 2 consiste em colocar todos os ficheiros do projeto, bem como o ficheiro README mencionado abaixo, num ficheiro com compressão no formato ZIP. O nome do ficheiro será **`grupoXX-projeto2.zip`** (XX é o número do grupo). Este ficheiro será depois entregue na página da disciplina, no moodle da FCUL.

O ficheiro ZIP deverá conter uma diretoria cujo nome é **`grupoXX`**, onde **`XX`** é o número do grupo. Nesta diretoria serão colocados:

- o ficheiro de texto README, onde os alunos podem incluir informações que julguem necessárias (e.g., limitações na implementação);
- diretorias adicionais:
  - include: para armazenar os ficheiros .h;
  - source: para armazenar os ficheiros *.c;*
  - object: para armazenar os ficheiros objeto;
  - binary: para armazenar os ficheiros executáveis.
- um ficheiro `Makefile` que permita a correta compilação de todos os ficheiros entregues. Não devem ser entregues ficheiros objeto (.o) ou executáveis. Os ficheiros de teste não deverão ser incluídos no ficheiro ZIP. No momento da avaliação eles serão colocados pelos docentes dentro da diretoria **`grupoXX/source`**. Os executáveis a gerar deverão ter o mesmo nome do ficheiro .c correspondente (sem a extensão .c). Espera-se que o Makefile compile os módulos bem como os programas de teste, o cliente e o servidor. Os ficheiros executáveis resultantes devem ficar na diretoria **`grupoXX/binary`**.
  Se não for incluído um `Makefile`, se o mesmo não compilar os ficheiros fonte, ou se houver erros de compilação (isto é, se não forem criados os ficheiros objeto e executáveis), o trabalho é considerado nulo.

  Na página da cadeira podem encontrar vídeos e documentos do utilitário `make` e dos ficheiros `Makefile` (cortesia da disciplina de Sistemas Operativos).

Todos os ficheiros entregues devem começar com três linhas de comentários a dizer o número do grupo e o nome e número de seus elementos.

Os programas são testados no ambiente dos laboratórios de aulas, pelo que se recomenda que os alunos testem os seus programas nesse ambiente.

O prazo de entrega é domingo, dia 29/10/2017, até às 22:00hs.

4. Bibliografia

[1] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. Proc. of the 21st Symposium on Operating System Principles – SOSP'07. pp. 205-220. Out. de 2007.

[2] Wikipedia. *Hash Table*. http://en.wikipedia.org/wiki/Hash_table.

[3] B. W. Kernighan, D. M. Ritchie, C Programming Language, 2nd Ed, Prentice-Hall, 1988.

[4] http://markburgess.org/CTutorial/CTutorial.html#Unions