Ciências ULisboa | Informática

2019/20

Sistemas Distribuídos

Projeto 1

1. Descrição Geral

A componente teórico-prática da disciplina de sistemas distribuídos consiste no desenvolvimento de quatro projetos, utilizando a linguagem de programação C [4], 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 primeiro projeto será concretizar um serviço de armazenamento de pares chave-valor (nos moldes da interface *java.util.Map* da API Java) similar ao utilizado pela *Amazon* para dar suporte aos seus serviços Web [1]. Neste sentido, as estruturas de dados utilizadas para armazenar esta informação são uma **lista encadeada simples** [2] e uma **tabela *hash*** [3], dada a sua elevada eficiência ao nível da pesquisa.

Sendo este tipo de serviço utilizado em sistemas distribuídos torna-se imperativa a troca de informação via rede. Esta informação deve ser enviada/recebida corretamente e pode representar qualquer estrutura de dados. Neste sentido torna-se necessário desenvolver mecanismos de serialização/deserialização a utilizar no envio/receção de informação. Neste sentido, no projeto 1, pretende-se que os alunos implementem as funções necessárias para **serializar** (codificar) e **de-serializar** (descodificar) estruturas de dados em arrays de bytes (para serem transmitidos na rede).

2. Descrição específica

O projeto 1 consiste na concretização dos seguintes módulos fundamentais:

- Definição do tipo de dados a armazenar;
- Definição de uma lista encadeada simples;
- Definição de uma estrutura de dados (tabela *hash*) onde serão armazenados pares (chave, valor) nos servidores, recorrendo a listas encadeadas simples;
- Criação da tabela e das operações sobre a mesma;
- Criação de um módulo que define funções para serializar (codificar) e de-serializar (descodificar) uma estrutura de dados complexa (mensagem) numa sequência de bytes, que são necessárias para a comunicação entre máquinas.

Para cada um destes módulos, é fornecido um ficheiro *header* (ficheiro com extensão *.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, cujas implementações serão também incluídas no ficheiro *X.c*. Os ficheiros *.h* apresentados neste documento, bem como alguns testes para as concretizações realizadas, serão disponibilizados na página da disciplina.

Juntamente com o enunciado do projeto é disponibilizado, aos alunos, um ficheiro zip contendo todos os *headers* definidos neste enunciado e um conjunto de ficheiros "-private.h" sugerindo funções que podem ser úteis implementar.

2.1. Definição do elemento de dados

A primeira tarefa consiste em definir o formato dos dados (associados a uma chave) que serão armazenados na tabela *hash*, no servidor. Para isso, é dado o ficheiro *data.h* que define a estrutura (*struct*) que contém os dados e respetiva dimensão, bem como funções para a sua criação, destruição e duplicação.

```
#ifndef _DATA_H
#define _DATA_H /* Módulo data */

/* Estrutura que define os dados.
 */
struct data_t {
      int datasize; /* Tamanho do bloco de dados */
      void *data;   /* Conteúdo arbitrário */
};

/* Função que cria um novo elemento de dados data_t, reservando a memória
 * necessária para armazenar os dados, especificada pelo parâmetro size
 */
struct data_t *data_create(int size);

/* Função que cria um novo elemento de dados data_t, inicializando o campo
 * data com o valor passado no parâmetro data, sem necessidade de reservar
 * memória para os dados.
 */
struct data_t *data_create2(int size, void *data);

/* Função que elimina um bloco de dados, apontado pelo parâmetro data,
 * libertando toda a memória por ele ocupada.
 */
void data_destroy(struct data_t *data);

/* Função que duplica uma estrutura data_t, reservando toda a memória
 * necessária para a nova estrutura, inclusivamente dados.
 */
struct data_t *data_dup(struct data_t *data);

#endif
```

Obs: *void** é um tipo que representa um apontador para um bloco genérico de dados. Na prática é equivalente a *char**, mas é aqui utilizado para evitar confusão com strings.

2.2. Definição de uma entrada (par chave-valor)

Definidos que estão os dados (o *valor*), é agora necessário criar uma entrada para a tabela *hash*, definida como um par {chave, valor}.

Para este efeito, é dado o ficheiro *entry.h* que define a estrutura de uma entrada na tabela, bem como algumas operações necessárias. Estas funções devem, naturalmente, utilizar as que estão implementadas no módulo `data`, se necessário.

```
#ifndef _ENTRY_H
#define _ENTRY_H /* Módulo entry */

#include "data.h"

/* Esta estrutura define o par {chave, valor} para a tabela
 */
struct entry_t {
```

```
	char *key;  /* string, cadeia de caracteres terminada por '\0' */
	struct data_t *value; /* Bloco de dados */
};

/* Função que cria uma entry, reservando a memória necessária para a
 * estrutura e inicializando os campos key e value, respetivamente, com a
 * string e o bloco de dados passados como parâmetros, sem reservar
 * memória para estes campos.

 */
struct entry_t *entry_create(char *key, struct data_t *data);

/* Função que elimina uma entry, libertando a memória por ela ocupada
 */
void entry_destroy(struct entry_t *entry);

/* Função que duplica uma entry, reservando a memória necessária para a
 * nova estrutura.
 */
struct entry_t *entry_dup(struct entry_t *entry);

#endif
```

2.3. Definição da estrutura de dados lista encadeada simples

O ficheiro *list.h* define as estruturas e as funções que concretizam uma lista ligada simples [2] que armazena estruturas do tipo *entry_t* contendo chaves e valores.

```
#ifndef _LIST_H
#define _LIST_H /* Módulo list */

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

struct list_t; /* a definir pelo grupo em list-private.h */

/* Função que cria uma nova lista (estrutura list_t a ser definida pelo
 * grupo no ficheiro list-private.h).
 * Em caso de erro, retorna NULL.
 */
struct list_t *list_create();

/* Função que elimina uma lista, libertando *toda* a memoria utilizada
 * pela lista.
 */
void list_destroy(struct list_t *list);

/* Função que adiciona no final da lista (tail) a entry passada como
 * argumento caso não exista na lista uma entry com key igual àquela
 * que queremos inserir.
 * Caso exista, os dados da entry (value) já existente na lista serão
 * substituídos pelos os da nova entry.
 * Retorna 0 (OK) ou -1 (erro).
 */
int list_add(struct list_t *list, struct entry_t *entry);

/* Função que elimina da lista a entry com a chave key.
 * Retorna 0 (OK) ou -1 (erro).
 */
int list_remove(struct list_t *list, char *key);

/* Função que obtém da lista a entry com a chave key.
```

```
 * Retorna a referência da entry na lista ou NULL em caso de erro.
 * Obs: as funções list_remove e list_destroy vão libertar a memória
 * ocupada pela entry ou lista, significando que é retornado NULL
 * quando é pretendido o acesso a uma entry inexistente.
*/
struct entry_t *list_get(struct list_t *list, char *key);

/* Função que retorna o tamanho (número de elementos (entries)) da lista,
 * ou -1 (erro).
 */
int list_size(struct list_t *list);

/* Função que devolve um array de char* com a cópia de todas as keys da
 * lista, colocando o último elemento do array com o valor NULL e
 * reservando toda a memória necessária.
 */
char **list_get_keys(struct list_t *list);

/* Função que liberta a memória ocupada pelo array das keys da lista,
 * obtido pela função list_get_keys.
 */
void list_free_keys(char **keys);

#endif
```

2.4. Tabela *hash*

A tabela *hash* deve armazenar os dados e oferecer operações do tipo **put**, **get**, **remove, list_keys** e **size**. A estrutura de dados ideal para armazenar o tipo de informação que desejamos e que oferece métodos de pesquisa eficientes é a tabela *hash* [3]. Uma função *hash* é usada para transformar cada chave num índice de um array (*list*) onde ficará armazenado o par chave-valor.

O ficheiro table.h define as estruturas e as funções a serem concretizadas neste módulo.

Recomenda-se a utilização da lista (módulo `list`) na concretização da tabela *hash*: a ideia é ter um *array* de tamanho fixo (definido pelo parâmetro da função *table_create*) contendo listas, ou seja, cada elemento do array será uma lista encadeada. Quando uma inserção ou procura for realizada, primeiro aplica-se uma função de *hash* para descobrir em que lista a entrada (*entry*) será inserida/procurada/removida, podendo depois fazer-se a inserção, procura, ou remoção nessa lista.

A função de *hash* a ser usada para mapear as chaves (strings) em índices de listas (inteiros entre um *0* e *n-1*) pode ser qualquer uma, a decidir pelo grupo. Por exemplo, uma possível função de *hash* consiste em somar o valor ASCII de todos os caracteres da chave e depois calcular o resto da divisão dessa soma por *n* (i.e, *soma(key) % n*).

O ficheiro *table.h* que define as estruturas e as funções a serem concretizadas neste módulo é o seguinte:

```
#ifndef _TABLE_H
#define _TABLE_H /* Módulo table */

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

struct table_t; /* A definir pelo grupo em table-private.h */

/* Função para criar/inicializar uma nova tabela hash, com n
```

```
 * linhas (n = módulo da função hash)
 * Em caso de erro retorna NULL.
 */
struct table_t *table_create(int n);

/* Função para libertar toda a memória ocupada por uma tabela.
 */
void table_destroy(struct table_t *table);

/* Função para adicionar um par chave-valor à tabela.
 * Os dados de entrada desta função deverão ser copiados, ou seja, a
 * função vai *COPIAR* a key (string) e os dados para um novo espaço de
 * memória que tem de ser reservado. Se a key já existir na tabela,
 * a função tem de substituir a entrada existente pela nova, fazendo
 * a necessária gestão da memória para armazenar os novos dados.
 * Retorna 0 (ok), em caso de adição ou substituição, ou -1 (erro), em
 * caso de erro.
 */
int table_put(struct table_t *table, char *key, struct data_t *value);

/* Função para obter da tabela o valor associado à chave key.
 * A função deve devolver uma cópia dos dados que terão de ser
 * libertados no contexto da função que chamou table_get, ou seja, a
 * função aloca memória para armazenar uma *CÓPIA* dos dados da tabela,
 * retorna o endereço desta memória com a cópia dos dados, assumindo-se
 * que esta memória será depois libertada pelo programa que chamou
 * a função.
 * Devolve NULL em caso de erro.
 */
struct data_t *table_get(struct table_t *table, char *key);

/* Função para remover um elemento da tabela, indicado pela chave key,
 * libertando toda a memória alocada na respetiva operação table_put.
 * Retorna 0 (ok) ou -1 (key not found or error).
 */
int table_del(struct table_t *table, char *key);

/* Função que devolve o número de elementos contidos na tabela.
 */
int table_size(struct table_t *table);

/* Função que devolve um array de char* com a cópia de todas as keys da
 * tabela, colocando o último elemento do array com o valor NULL e
 * reservando toda a memória necessária.
 */
char **table_get_keys(struct table_t *table);

/* Função que liberta toda a memória alocada por table_get_keys().
 */
void table_free_keys(char **keys);

#endif
```

2.5. Marshaling e unmarshaling de estruturas de dados

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

```
#ifndef _SERIALIZATION_H
#define _SERIALIZATION_H

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

/* Serializa uma estrutura data num buffer que será alocado
 * dentro da função. Além disso, retorna o tamanho do buffer
 * alocado ou -1 em caso de erro.
 */
int data_to_buffer(struct data_t *data, char **data_buf);

/* De-serializa a mensagem contida em data_buf, com tamanho
 * data_buf_size, colocando-a e retornando-a numa struct
 * data_t, cujo espaco em memoria deve ser reservado.
 * Devolve NULL em caso de erro.
 */
struct data_t *buffer_to_data(char *data_buf, int data_buf_size);

/* Serializa uma estrutura entry num buffer que sera alocado
 * dentro da função. Além disso, retorna o tamanho deste
 * buffer alocado ou -1 em caso de erro.
 */
int entry_to_buffer(struct entry_t *data, char **entry_buf);

/* De-serializa a mensagem contida em entry_buf, com tamanho
 * entry_buf_size, colocando-a e retornando-a numa struct
 * entry_t, cujo espaco em memoria deve ser reservado.
 * Devolve NULL em caso de erro.
 */
struct entry_t *buffer_to_entry(char *entry_buf, int entry_buf_size);

#endif
```

3. Entrega

A entrega do projeto 1 tem de ser feita de acordo com as seguintes regras:

1. 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-projeto1.zip`** (XX é o número do grupo).
2. Submeter o ficheiro **`grupoXX-projeto1.zip`** na página da disciplina no moodle da FCUL, utilizando a atividade disponibilizada para tal. Apenas um dos elementos do grupo deve submeter, considerando-se apenas a submissão mais recente no caso de existirem várias.

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

- o ficheiro README, onde os alunos podem incluir informações que julguem necessárias (e.g., limitações na implementação);
- diretorias adicionais, nomeadamente:
  - 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 incluídos no ficheiro ZIP os ficheiros objeto (.o) ou

executáveis. Os ficheiros de teste também não deverão ser incluídos no ficheiro ZIP. No momento da avaliação eles serão colocados pelos docentes dentro da diretoria **`grupoXX/source`**. Espera-se que o `Makefile` compile os módulos bem como os programas de teste. Os ficheiros executáveis resultantes devem ficar na diretoria **`grupoXX/binary`** com os nomes **`test_data`**, **`test_entry`**, **`test_list`**, **`test_table`** e **`test_serialization`**.

Na entrega do trabalho, é ainda necessário ter em conta que:

- **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 disciplina, no Moodle, 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 um cabeçalho com três ou quatro linhas de comentários a dizer o número do grupo e o nome e número dos 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 dia 13/10/2019 até às 23:55hs.

Após esta data, a submissão do trabalho através do Moodle deixará de ser permitida.

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 . *Linked List*. https://en.wikipedia.org/wiki/Linked_list.

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

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