Ciências ULisboa | Informática

2023/24

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/de-serialização a utilizar no envio/receção de informação.

Em suma, neste primeiro projeto pretende-se que os alunos implementem um conjunto de módulos de base na linguagem C, testando o seu funcionamento para garantir que as funcionalidades estão corretamente implementadas. Dado que estes módulos serão usados nos projetos seguintes, é importante assegurar que não têm *bugs*.

2. Descrição específica

O projeto 1 consiste na concretização dos seguintes módulos fundamentais:

- Criação de módulos (**data** e **entry**) que definem o tipo de dados a armazenar e as operações sobre os mesmos;
- Criação de um módulo (**list**) que define uma lista encadeada simples e as operações sobre a mesma;
- Criação de um módulo (**table**) que define uma estrutura de dados (tabela *hash*) baseada em listas encadeadas simples, onde serão armazenados pares (chave, valor) nos servidores, e que defina também das operações sobre a mesma;
- Criação de um módulo (**serialization**) que define funções para serializar (codificar) uma estrutura de dados complexa (neste caso, um *array* de *strings*) numa sequência de bytes (a mensagem a enviar), e de-serializar (descodificar) a mensagem para recriar a estrutura complexa correspondente, 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 um ficheiro ZIP contendo todos os *headers* definidos neste enunciado, bem como *templates* de ficheiros "-private.h" que deverão ser editados pelos grupos.

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 e que inicializa
 * os dados de acordo com os argumentos recebidos, sem necessidade de
 * reservar memória para os dados.
 * Retorna a nova estrutura ou NULL em caso de erro.
 */
struct data_t *data_create(int size, void *data);

/* Função que elimina um bloco de dados, apontado pelo parâmetro data,
 * libertando toda a memória por ele ocupada.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int data_destroy(struct data_t *data);

/* Função que duplica uma estrutura data_t, reservando a memória
 * necessária para a nova estrutura.
 * Retorna a nova estrutura ou NULL em caso de erro.
 */
struct data_t *data_dup(struct data_t *data);

/* Função que substitui o conteúdo de um elemento de dados data_t.
 * Deve assegurar que liberta o espaço ocupado pelo conteúdo antigo.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int data_replace(struct data_t *data, int new_size, void *new_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***. Na implementação das funções, não pode ser assumido que os dados são terminados com '\0'.

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 para manipular esta estrutura. 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 e
 * inicializando-a com a string e o bloco de dados de entrada.
 * Retorna a nova entry ou NULL em caso de erro.
 */
struct entry_t *entry_create(char *key, struct data_t *data);

/* Função que elimina uma entry, libertando a memória por ela ocupada.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int entry_destroy(struct entry_t *entry);

/* Função que duplica uma entry, reservando a memória necessária para a
 * nova estrutura.
 * Retorna a nova entry ou NULL em caso de erro.
 */
struct entry_t *entry_dup(struct entry_t *entry);

/* Função que substitui o conteúdo de uma entry, usando a nova chave e
 * o novo valor passados como argumentos, e eliminando a memória ocupada
 * pelos conteúdos antigos da mesma.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int entry_replace(struct entry_t *entry, char *new_key, struct data_t
*new_value);

/* Função que compara duas entries e retorna a ordem das mesmas, sendo esta
 * ordem definida pela ordem das suas chaves.
 * Retorna 0 se as chaves forem iguais, -1 se entry1 < entry2,
 * 1 se entry1 > entry2 ou -2 em caso de erro.
 */
int entry_compare(struct entry_t *entry1, struct entry_t *entry2);

#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 e inicializa uma nova lista (estrutura list_t a
 * ser definida pelo grupo no ficheiro list-private.h).
 * Retorna a lista ou NULL em caso de erro.
 */
struct list_t *list_create();

/* Função que elimina uma lista, libertando *toda* a memória utilizada
 * pela lista.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int list_destroy(struct list_t *list);

/* Função que adiciona à lista a entry passada como argumento.
 * A entry é inserida de forma ordenada, tendo por base a comparação
 * de entries feita pela função entry_compare do módulo entry e
 * considerando que a entry menor deve ficar na cabeça da lista.
 * Se já existir uma entry igual (com a mesma chave), a entry
 * já existente na lista será substituída pela nova entry,
 * sendo libertada a memória ocupada pela entry antiga.
 * Retorna 0 se a entry ainda não existia, 1 se já existia e foi
 * substituída, ou -1 em caso de erro.
 */
int list_add(struct list_t *list, struct entry_t *entry);

/* Função que elimina da lista a entry com a chave key, libertando a
 * memória ocupada pela entry.
 * Retorna 0 se encontrou e removeu a entry, 1 se não encontrou a entry,
 * ou -1 em caso de 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 se não encontrar a
 * entry ou em caso de erro.
*/
struct entry_t *list_get(struct list_t *list, char *key);

/* Função que conta o número de entries na lista passada como argumento.
 * Retorna o tamanho da lista ou -1 em caso de erro.
 */
int list_size(struct list_t *list);

/* Função que constrói um array de char* com a cópia de todas as keys na
 * lista, colocando o último elemento do array com o valor NULL e
 * reservando toda a memória necessária.
 * Retorna o array de strings ou NULL em caso de erro.
 */
char **list_get_keys(struct list_t *list);

/* Função que liberta a memória ocupada pelo array de keys obtido pela
 * função list_get_keys.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int 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, size, get_keys** e **free_keys**. 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*, o que facilitará o acesso ao 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 *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"

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

/* Função para criar e inicializar uma nova tabela hash, com n
 * linhas (n = módulo da função hash).
 * Retorna a tabela ou NULL em caso de erro.
 */
struct table_t *table_create(int n);

/* Função que elimina uma tabela, libertando *toda* a memória utilizada
 * pela tabela.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int 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 criar uma nova
 * entry com *CÓPIAS* da key (string) e dos dados. Se a key já existir na
 * tabela, a função tem de substituir a entry existente na tabela pela
 * nova, fazendo a necessária gestão da memória.
 * Retorna 0 (ok) ou -1 em caso de erro.
 */
int table_put(struct table_t *table, char *key, struct data_t *value);

/* Função que procura na tabela uma entry com a chave key.
 * Retorna uma *CÓPIA* dos dados (estrutura data_t) nessa entry ou
 * NULL se não encontrar a entry ou em caso de erro.
 */
struct data_t *table_get(struct table_t *table, char *key);

/* Função que remove da lista a entry com a chave key, libertando a
 * memória ocupada pela entry.
```

```
 * Retorna 0 se encontrou e removeu a entry, 1 se não encontrou a entry,
 * ou -1 em caso de erro.
 */
int table_remove(struct table_t *table, char *key);

/* Função que conta o número de entries na tabela passada como argumento.
 * Retorna o tamanho da tabela ou -1 em caso de erro.
 */
int table_size(struct table_t *table);

/* Função que constrói um array de char* com a cópia de todas as keys na
 * tabela, colocando o último elemento do array com o valor NULL e
 * reservando toda a memória necessária.
 * Retorna o array de strings ou NULL em caso de erro.
 */
char **table_get_keys(struct table_t *table);

/* Função que liberta a memória ocupada pelo array de keys obtido pela
 * função table_get_keys.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int table_free_keys(char **keys);

#endif
```

2.5. Serialização e de-serialização de estruturas de dados

Este módulo do projeto consiste em transformar uma estrutura de dados complexa num formato que possa ser enviado pela rede (isto é, converter uma estrutura com vários elementos num formato "unidimensional"), e vice-versa. O ficheiro *serialization.h* define as funções a serem concretizadas neste módulo. De salienta que a serialização tem de transformar o valor inteiro em *network byte order* e vice-versa.

O formato dos dados serializados está indicado na descrição das funções no ficheiro *serialization.h*.

```
#ifndef _SERIALIZATION_H
#define _SERIALIZATION_H

/* Serializa todas as chaves presentes no array de strings keys para o
 * buffer keys_buf, que será alocado dentro da função. A serialização
 * deve ser feita de acordo com o seguinte formato:
 *    | int   | string | string | string |
 *    | nkeys | key1   | key2   | key3   |
 * Retorna o tamanho do buffer alocado ou -1 em caso de erro.
 */
int keyArray_to_buffer(char **keys, char **keys_buf);

/* De-serializa a mensagem contida em keys_buf, colocando-a num array de
 * strings cujo espaco em memória deve ser reservado. A mensagem contida
 * em keys_buf deverá ter o seguinte formato:
 *    | int   | string | string | string |
 *    | nkeys | key1   | key2   | key3   |
 * Retorna o array de strings ou NULL em caso de erro.
 */
char** buffer_to_keyArray(char *keys_buf);

#endif
```

3. Teste dos módulos desenvolvidos

A realização de testes para verificar que os módulos foram corretamente implementados e que as funções satisfazem os requisitos é muito importante, dado que os módulos serão usados nos projetos seguintes.

Assim, são disponibilizados testes que os grupos poderão usar para verificar a correção do seu código. De notar que os testes disponibilizados não são exaustivos, pelo que o facto do código passar os testes não significa que esteja 100% correto e não tenha *bugs*. Por exemplo, aspetos como libertação de memória previamente reservada não são testados nestes testes.

Assim, os testes devem ser considerados uma ajuda, mas é da responsabilidade dos grupos assegurar que a máxima correção possível do código desenvolvido e entregue.

4. 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 9/10/2023, até às 23:59 horas.

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

5. Autoavaliação de contribuições

Cada aluno tem de preencher no Moodle um formulário de autoavaliação das contribuições individuais de cada elemento do grupo para o projeto. Por exemplo, se todos os elementos colaboraram de forma idêntica, bastará que todos indiquem que cada um contribuiu 33%. Aplicam-se as seguintes regras e penalizações:

- Alunos que não preencham o formulário **sofrem uma penalização na nota de 10%.**
- Caso existam assimetrias significativas entre as respostas de cada elemento do grupo, o grupo poderá ser chamado para as explicar.
- Se as contribuições individuais forem diferentes, isso será refletido na nota de cada elemento do grupo, levando à atribuição de notas individuais diferentes.

O prazo de preenchimento desde formulário é o mesmo que a entrega do projeto (9/10/2023, até às 23:59 horas).

6. 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.