



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 **árvore binária de pesquisa** [2], 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 do tipo de dados par chave-valor;
- Definição da estrutura de dados árvore binária de pesquisa, onde serão armazenados pares (chave, valor) nos servidores;
- 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 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);

/* Função que substitui o conteúdo de um elemento de dados data_t.
 * Deve assegurar que destrói o conteúdo antigo do mesmo.
 */
void data_replace(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.

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

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

Para este efeito, é dado o ficheiro *entry.h* que define a estrutura de uma entrada na árvore, 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 árvore
 */
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);

/* Função que substitui o conteúdo de uma entrada entry_t.
 * Deve assegurar que destrói o conteúdo antigo da mesma.
 */
void entry_replace(struct entry_t *entry, char *new_key, struct data_t
*new_value);

/* Função que compara duas entradas e retorna a ordem das mesmas.
 * Ordem das entradas é definida pela ordem das suas chaves.
 * A função devolve 0 se forem iguais, -1 se entry1 < entry2, e 1 caso
 * contrário.
 */
int entry_compare(struct entry_t *entry1, struct entry_t entry2);

#endif

```

2.3. Árvore Binária de Pesquisa

A árvore binária de pesquisa [3] deve armazenar os dados e oferecer operações do tipo **put**, **get**, **delete**, **get_keys**, **size** e **height**. A árvore deve usar a função **entry_compare** de *entry.h* para conseguir ordenar entradas e saber onde inserir/pesquisar entradas.

O ficheiro *tree.h* define as estruturas e as funções a serem concretizadas neste módulo:

```

#ifndef _TREE_H
#define _TREE_H /* Módulo tree */

#include "data.h"

struct tree_t; /* A definir pelo grupo em tree-private.h */

/* Função para criar uma nova árvore tree vazia.
 * Em caso de erro retorna NULL.
 */
struct tree_t *tree_create();

/* Função para libertar toda a memória ocupada por uma árvore.

```

```

*/
void tree_destroy(struct tree_t *tree);

/* Função para adicionar um par chave-valor à árvore.
 * 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 árvore,
 * a função tem de substituir a entrada existente pela nova, fazendo
 * a necessária gestão da memória para armazenar os novos dados.
 * Retorna 0 (ok) ou -1 em caso de erro.
 */
int tree_put(struct tree_t *tree, char *key, struct data_t *value);

/* Função para obter da árvore 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 tree_get, ou seja, a
 * função aloca memória para armazenar uma *CÓPIA* dos dados da árvore,
 * 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 *tree_get(struct tree_t *tree, char *key);

/* Função para remover um elemento da árvore, indicado pela chave key,
 * libertando toda a memória alocada na respetiva operação tree_put.
 * Retorna 0 (ok) ou -1 (key not found).
 */
int tree_del(struct tree_t *tree, char *key);

/* Função que devolve o número de elementos contidos na árvore.
 */
int tree_size(struct tree_t *tree);

/* Função que devolve a altura da árvore.
 */
int tree_height(struct tree_t *tree);

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

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

#endif

```

2.4. 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"

```

```

#include "tree.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 espaço em memória 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 será 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 espaço em memória deve ser reservado.
 * Devolve NULL em caso de erro.
 */
struct entry_t *buffer_to_entry(char *entry_buf, int entry_buf_size);

/* Serializa uma estrutura tree num buffer que será alocado
 * dentro da função. Além disso, retorna o tamanho deste
 * buffer ou -1 em caso de erro.
 */
int tree_to_buffer(struct tree_t *tree, char **tree_buf);

/* De-serializa a mensagem contida em tree_buf, com tamanho
 * tree_buf_size, colocando-a e retornando-a numa struct
 * tree_t, cujo espaço em memória deve ser reservado.
 * Devolve NULL em caso de erro.
 */
struct entry_t *buffer_to_tree(char *tree_buf, int tree_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_tree** 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 11/10/2020 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 . *Binary Search Tree*. https://en.wikipedia.org/wiki/Binary_search_tree
- [3] B. W. Kernighan, D. M. Ritchie, *C Programming Language*, 2nd Ed, Prentice-Hall, 1988.