Ciências ULisboa | Informática

2020/21

Sistemas Distribuídos

Projeto 2

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 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.

No Projeto 1 foram definidas estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na árvore binária de pesquisa, bem como para gerir uma árvore local que suporte um subconjunto dos serviços definidos pela interface *Map*.

O projeto 2 tem como objetivo a implementação das funções necessárias à **serialização** (codificar) e **de-serialização** (descodificar) usando o protocolo buffer [5]; à concretização da árvore binária de pesquisa num servidor, e à implementação de um cliente com uma interface similar àquela concretizada no Projeto 1 para a árvore (definida pelo ficheiro `tree.h`), ou seja, às funções que interagem com a àrvore na memória. 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 na árvore local concretizada por ele;
    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.

O objetivo final é fornecer às aplicações que usariam esta árvore um modelo de comunicação tipo RPC (*Remote Procedure Call*)[1], onde vários clientes acedem a uma mesma árvore binária de pesquisa partilhada.

O ponto inicial para a concretização do Projecto 2 é a combinação do código desenvolvido no Projecto 1 com as técnicas para comunicação por *sockets* TCP.

Espera-se uma grande fiabilidade por parte do servidor, portanto não podem 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 a árvore partilhada (no caso da *Amazon*, se o serviço que mantém as cestas de compras dos clientes não funciona, a empresa não vende, e perde milhões por hora).

---

[1] Ver aula teórica sobre RPC e capítulo 4 do livro texto da cadeira.

2. Descrição específica

O objetivo específico do projeto 2 é desenvolver uma aplicação do tipo cliente-servidor, utilizando o paradigma das chamadas a procedimentos remotos (RPC – *Remote Procedure Calls*) para concretizar a interação entre o cliente e o servidor. O programa cliente deverá implementar uma interface com o utilizador, permitindo que este solicite a execução de operações numa árvore binária de pesquisa. Esta árvore será implementada no programa servidor (utilizando o código desenvolvido no projeto 1), que deverá oferecer um conjunto de procedimentos acessíveis remotamente, para executar determinadas operações na árvore. Assim, o cliente terá de ser programado para enviar para o servidor as operações solicitadas pelo utilizador, recebendo e apresentando as respostas. Por seu lado, o servidor terá de ser programado para esperar um pedido de ligação de um cliente e para receber, executar e responder às operações enviadas por esse cliente, até que este se desligue (podendo depois atender um novo cliente).

Em termos práticos, o projeto 2 consiste na concretização em C de:

1. um módulo que define funções para serializar (codificar) e de-serializar (descodificar) de uma mensagem usando o protocolo buffer [5]. Estas mensagens são necessárias para a comunicação entre cliente e servidor;
2. dois programas, a serem executados da seguinte forma:
    a. **tree-server** `<port>`
    `<port>` é o número do porto TCP ao qual o servidor se deve associar (fazer *bind*).

    b. **tree-client** `<server>:<port>`
    `<server>` é o endereço IP ou nome do servidor da árvore.
    `<port>` é o número do porto TCP onde o servidor está à espera de ligações.

O programa cliente (**tree-client**) permitirá o envio de comandos via rede para invocar operações (na árvore binária de pesquisa) implementadas pelo servidor. Por outro lado, o programa servidor (**tree-server**) receberá os comandos vindos dos clientes, executando-os sobre a árvore e devolvendo a resposta ao cliente. Estes comandos são constituídos por *opcodes* representativos de *operações* e *tipo de conteúdo* a operar na árvore binária de pesquisa. A Árvore 1 apresenta os comandos e respetivas sintaxes, que os clientes podem requerer e a que o servidor pode responder. A secção seguinte apresenta a informação detalhada sobre a constituição e o formato das mensagens.

COMANDO UTILIZADOR	**MENSAGEM DE PEDIDO**	**MENSAGEM DE RESPOSTA**
size	OP_SIZE CT_NONE <none>	OP_SIZE+1 CT_RESULT <result>
del <key>	OP_DEL CT_KEY <key>	OP_DEL+1 CT_NONE <none> OP_ERROR CT_NONE <none>
get <key>	OP_GET CT_KEY <key>	OP_GET+1 CT_VALUE <value> OP_ERROR CT_NONE <none>
put <key> <data>	OP_PUT CT_ENTRY <entry>	OP_PUT+1 CT_NONE <none> OP_ERROR CT_NONE <none>
getkeys	OP_GETKEYS CT_NONE <none>	OP_GETKEYS+1 CT_KEYS <keys>
quit	--	--

Árvore 1: Definição dos comandos e respetivas mensagens de pedidos e de resposta.

Caso a operação sobre a árvore seja bem-sucedida, o *opcode* de resposta é igual ao *opcode* do pedido incrementado de uma unidade. Por seu turno, caso a operação no servidor não seja bem-sucedida, o *opcode* da resposta será OP_ERROR.

Os *opcode* utilizados são listados de seguida:

```
/* Define os possíveis opcodes da mensagem */
OP_SIZE          10
OP_DEL           20
OP_GET           30
OP_PUT           40
OP_GETKEYS       50
/* opcode para representar retorno de erro da execução da operação*/
OP_ERROR         99

/* Define códigos para os possíveis conteúdos da mensagem (c_type) */
CT_KEY    10
CT_VALUE  20
CT_ENTRY  30
CT_KEYS   40
CT_RESULT 50
/* Opcode representativo de inexistência de content
 * (e.g., getkeys e size */
CT_NONE   60
```

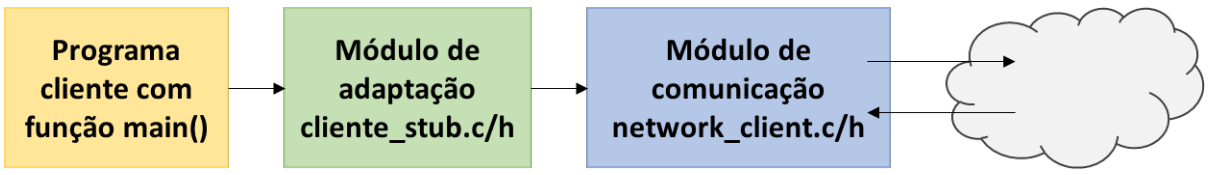
Para a concretização do módulo e dos programas referidos anteriormente, é fornecido um ficheiro *.h* com os cabeçalhos das funções, que **não pode ser alterado**. As concretizações das funções definidas nos ficheiros *X.h* devem ser feitas num ficheiro *X.c*, utilizando os algoritmos e métodos que o 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 serão disponibilizados na página da disciplina.

3. Cliente

O cliente usa alguns dos módulos já desenvolvidos no Projeto 1 (`data.c/h` e `entry.c/h`), juntamente com três módulos adicionais:

- Programa cliente com a função `main()` (`tree_client.c`)
- Módulo de adaptação das chamadas do cliente, RPC *stub* (`client_stub.c/h`)
- Módulo de comunicação (`network_client.c/h`)

A figura abaixo ilustra a interação entre os três módulos, onde o **módulo de adaptação** do cliente, isto é, o **RPC *stub*** (client_stub.c/h) permite a **interação entre o programa cliente e o módulo de comunicação**, tendo como função fazer com que **todos os detalhes relativos à comunicação sejam transparentes para o cliente**, ou seja, escondidos deste.



3.1. Programa cliente: `tree_client.c`

O programa cliente consiste num programa interativo simples, que quando executado aceita um comando (uma linha) do utilizador no *stdin*, invoca a chamada remota através da respetiva função do *stub* (Secção 3.2), imprime a resposta recebida no ecrã e volta a aceitar um novo comando. Uma boa forma de ler e tratar os comandos inseridos é usando as funções *fgets* e *strtok*. Cada comando vai ser inserido pelo utilizador numa única linha, devendo ser aceites os seguintes comandos (ver 1ª coluna da Árvore 1):

```
put <key> <data>
get <key>
del <key>
size
height
getkeys
quit
```

Note que a aplicação cliente vai usar o campo `'data'` da estrutura `data_t` do Projecto 1 para guardar a *string* `<data>` introduzida pelo utilizador (que pode conter espaços, i.e., a *string* completa após a chave (`<key>`) até ao fim da linha). No entanto, na definição de todos os restantes módulos (*client_stub.c*, *network_client.c* e módulos do servidor) deve-se assumir que o campo dados pode conter dados arbitrários (e.g., imagens, som, etc.).

3.2. RPC *stub*: `client_stub.c`

O *stub* implementa e disponibiliza uma função de adaptação para cada operação remota que o cliente possa efetuar no servidor, bem como funções para estabelecer e terminar a ligação a um servidor. Cada função de adaptação vai preencher uma estrutura `message_t` com o `opcode` (tipo da operação), com o tipo de conteúdo (`c_type`) necessário para realizar a operação, e com esse conteúdo. Esta estrutura é então passada para o módulo de comunicação, que a irá processar e devolver uma resposta (veja a próxima secção).

A interface a ser oferecida ao cliente é a seguinte:

```
#ifndef _CLIENT_STUB_H
#define _CLIENT_STUB_H

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

/* Remote tree. A definir pelo grupo em client_stub-private.h
 */
struct rtree_t;

/* Função para estabelecer uma associação entre o cliente e o servidor,
 * em que address_port é uma string no formato <hostname>:<port>.
 * Retorna NULL em caso de erro.
 */
struct rtree_t *rtree_connect(const char *address_port);
```

```
/* Termina a associação entre o cliente e o servidor, fechando a
 * ligação com o servidor e libertando toda a memória local.
 * Retorna 0 se tudo correr bem e -1 em caso de erro.
 */
int rtree_disconnect(struct rtree_t *rtree);

/* Função para adicionar um elemento na árvore.
 * Se a key já existe, vai substituir essa entrada pelos novos dados.
 * Devolve 0 (ok, em adição/substituição) ou -1 (problemas).
 */
int rtree_put(struct rtree_t *rtree, struct entry_t *entry);

/* Função para obter um elemento da árvore.
 * Em caso de erro, devolve NULL.
 */
struct data_t *rtree_get(struct rtree_t *rtree, char *key);

/* Função para remover um elemento da árvore. Vai libertar
 * toda a memoria alocada na respetiva operação rtree_put().
 * Devolve: 0 (ok), -1 (key not found ou problemas).
 */
int rtree_del(struct rtree_t *rtree, char *key);

/* Devolve o número de elementos contidos na árvore.
 */
int rtree_size(struct rtree_t *rtree);

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

/* Devolve um array de char* com a cópia de todas as keys da árvore,
 * colocando um último elemento a NULL.
 */
char **rtree_get_keys(struct rtree_t *rtree);

/* Liberta a memória alocada por rtree_get_keys().
 */
void rtree_free_keys(char **keys);

#endif
```

É importante relembrar que quando uma operação é executada pelo servidor com sucesso, o *opcode* para a mensagem de resposta deve ter o valor (*opcode* do pedido + 1).

3.3. Módulo de comunicação do cliente: `network_client.c`

O módulo *network-client.c* vai serializar a mensagem, enviá-la ao servidor e esperar pela resposta. Do mesmo modo, o módulo vai de-serializar a resposta vinda do servidor, e devolver a mensagem de resposta ao cliente. A biblioteca de comunicação *network_client.c* tem a seguinte interface:

```
#ifndef _NETWORK_CLIENT_H
#define _NETWORK_CLIENT_H

#include "client_stub.h"
#include "sdmessage.pb-c.h"

/* Esta função deve:
```

```
 * - Obter o endereço do servidor (struct sockaddr_in) a base da
 *   informação guardada na estrutura rtree;
 * - Estabelecer a ligação com o servidor;
 * - Guardar toda a informação necessária (e.g., descritor do socket)
 *   na estrutura rtree;
 * - Retornar 0 (OK) ou -1 (erro).
 */
int network_connect(struct rtree_t *rtree);

/* Esta função deve:
 * - Obter o descritor da ligação (socket) da estrutura rtree_t;
 * - Serializar a mensagem contida em msg;
 * - Enviar a mensagem serializada para o servidor;
 * - Esperar a resposta do servidor;
 * - De-serializar a mensagem de resposta;
 * - Retornar a mensagem de-serializada ou NULL em caso de erro.
 */
struct message_t *network_send_receive(struct rtree_t * rtree,
                                       struct message_t *msg);

/* A função network_close() fecha a ligação estabelecida por
 * network_connect().
 */
int network_close(struct rtree_t * rtree);

#endif
```

As mensagens na rede devem usar o formato definido no ficheiro .proto utilizado pelo protocolo buffer e devem ter o nome de `message_t` . De seguida é apresentado um formato possível para a `message`. Este formato pode ser alterado de acordo com a vossa realização do projeto. **De relembrar que apenas o ficheiro .proto pode ser alterado, os restantes ficheiros .h devem permanecer inalterados.**

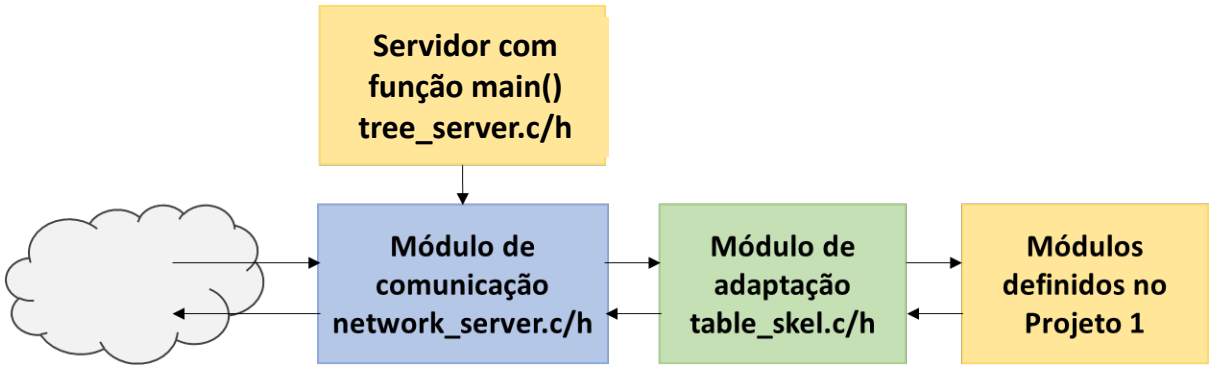
```
syntax = "proto3";
message message_t
 {
        enum Opcode {
                OP_BAD     = 0;
                OP_SIZE    = 10;
                OP_DEL     = 20;
                OP_GET     = 30;
                OP_PUT     = 40;
                OP_GETKEYS = 50;
                OP_ERROR   = 99;
        }
        Opcode opcode = 1;

        enum C_type {
                CT_BAD    = 0;
                CT_KEY    = 10;
                CT_VALUE  = 20;
                CT_ENTRY  = 30;
                CT_KEYS   = 40;
                CT_RESULT = 50;
                CT_NONE   = 60;
        }
        C_type c_type = 2;
        sint32 data_size = 3;
        string data = 4;

};
```

4. Servidor

O **servidor** a realizar usa todas as funcionalidades do projecto 1, e é composto por três partes adicionais:

- Aplicação servidor (`tree_server.c`) com a função `main()`
- Módulo de comunicação (`network_server.c/h`)
- Módulo de adaptação do servidor, i.e., RPC *skeleton* (`tree_skel.h/c`)

A figura abaixo ilustra a interação entre os três módulos, onde o **módulo de adaptação** do servidor, isto é, o **RPC *skeleton*** (tree_skel.c/h), permite a **interação entre o módulo de comunicação e a árvore binária de pesquisa**, permitindo também a inicialização da árvore (feita na função main().



4.1. Aplicação servidor: `tree_server.c`

O servidor concretiza uma árvore binária de pesquisa que pode ser acedida no porto definido na linha de comando. O servidor deverá suportar apenas um cliente de cada vez. Como o servidor apenas necessita de interagir com um cliente de cada vez, não será necessário, nesta fase, recorrer à criação de processos ligeiros (*threads*) para atendimento simultâneo de vários clientes.

4.2. Módulo de Comunicação do servidor: `network_server.c`

O módulo *network_server.c* vai permitir colocar o servidor à escuta de pedidos de conexão de clientes (serão usados *sockets* TCP) e eliminar o *socket* do servidor, através das funções *network_server_init* e *network_server_close*. A função *network_main_loop* é responsável pela interação com a rede (receber pedidos de ligação, receber dados e enviar dados) e pela interação com o *skeleton* (*tree_skel*). Assim sendo, a função recebe sequências de bytes que constituem as mensagens com os pedidos dos clientes, realiza a de-serialização destes bytes, construindo uma mensagem (estrutura *message_t*), a qual é entregue ao *skeleton* para processamento. Em resposta, a função recebe do *skeleton* uma mensagem com o resultado da operação, serializa esta mensagem numa sequência de bytes, enviando estes bytes de resposta ao cliente através da ligação TCP. A biblioteca de comunicação *network_server.c* tem a seguinte interface:

```
#ifndef _NETWORK_SERVER_H
#define _NETWORK_SERVER_H

#include "tree_skel.h"
```

```
/* Função para preparar uma socket de receção de pedidos de ligação
 * num determinado porto.
 * Retornar descritor do socket (OK) ou -1 (erro).
 */
int network_server_init(short port);

/* Esta função deve:
 * - Aceitar uma conexão de um cliente;
 * - Receber uma mensagem usando a função network_receive;
 * - Entregar a mensagem de-serializada ao skeleton para ser processada;
 * - Esperar a resposta do skeleton;
 * - Enviar a resposta ao cliente usando a função network_send.
 */
int network_main_loop(int listening_socket);

/* Esta função deve:
 * - Ler os bytes da rede, a partir do client_socket indicado;
 * - De-serializar estes bytes e construir a mensagem com o pedido,
 *   reservando a memória necessária para a estrutura message_t.
 */
struct message_t *network_receive(int client_socket);

/* Esta função deve:
 * - Serializar a mensagem de resposta contida em msg;
 * - Libertar a memória ocupada por esta mensagem;
 * - Enviar a mensagem serializada, através do client_socket.
 */
int network_send(int client_socket, struct message_t *msg);

/* A função network_server_close() liberta os recursos alocados por
 * network_server_init().
 */
int network_server_close();

#endif
```

4.3. RPC skeleton: `tree_skel.c`

O *skeleton*, a concretizar em `tree_skel.c`, serve para transformar uma mensagem do cliente (a qual foi de-serializada pela camada de rede) numa chamada da respetiva função do módulo tree (`tree.c`). A interface é propositadamente muito simples:

```
#ifndef _TREE_SKEL_H
#define _TREE_SKEL_H

#include "sdmessage.pb-c.h"
#include "tree.h"

/* Inicia o skeleton da árvore.
 * O main() do servidor deve chamar esta função antes de poder usar a
 * função invoke().
 * Retorna 0 (OK) ou -1 (erro, por exemplo OUT OF MEMORY)
 */
int tree_skel_init();

/* Liberta toda a memória e recursos alocados pela função tree_skel_init.
 */
void tree_skel_destroy();

/* Executa uma operação na árvore (indicada pelo opcode contido em msg)
```

```
 * e utiliza a mesma estrutura message_t para devolver o resultado.
 * Retorna 0 (OK) ou -1 (erro, por exemplo, árvore nao incializada)
*/
int invoke(struct message_t *msg);

#endif
```

As funções `tree_skel_init` e `tree_skel_destroy` servem fundamentalmente para criar e destruir a árvore a ser mantida pelo servidor. Já a função `invoke` interpreta o campo *opcode* de `msg` para seleccionar a operação a executar nessa árvore. Depois de executar a operação pedida na árvore local, coloca-se o resultado na mesma estrutura `msg`, alterando-se os campos `opcode`, `c_type` e `content`. Caso a operação na árvore tenha sido executada com sucesso, o *opcode* da estrutura `msg` será o valor do *opcode* do pedido incrementado de uma unidade e o campo `c_type` poderá ser CT_RESULT ou CT_NONE. Por outro lado, caso ocorra erro na operação executada na árvore, o campo `opcode` receberá o valor de OP_ERROR e o cmpo `c_type` o valor de CT_NONE (ver Árvore 1).

4.4. Esqueleto da função main() do servidor

A descrição seguinte não está escrita em código C compilável, apenas apresenta a ideia geral de como deve ser o `main` do servidor. Cabe aos alunos traduzir essa lógica (ou outra que entendam definir) para código C.

```
/* Testar os argumentos de entrada */
/* inicialização da camada de rede */
socket_de_escuta = network_server_init(/* */)
tree_skel_init();
int result = network_main_loop(socket_de_escuta)
tree_skel_destroy();
```

5. Observações

Algumas observações e dicas úteis:

- Recomenda-se a criação de funções *read_all* e *write_all* que vão receber e enviar *strings* inteiras pela rede (lembrar que as funções read/write em sockets nem sempre lêem/escrevem tudo o que pedimos). Um bom sítio para concretizar essas funções é num modulo 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 quanto no servidor, evitando que um programa termine abruptamente (*crash*) quando a outra parte é desligada.
- Usar a função *setsockopt(..., SO_REUSEADDR, ...)* para fazer com que o servidor consiga fazer *bind* a um porto usado anteriormente e registado pelo kernel como ainda ocupado. Isto permite que o servidor seja reinicializado rapidamente, sem ter de esperar o tempo de limpeza da árvore de portos usados, mantida pelo kernel.

- Caso algum dos pedidos não possa ser atendido devido a um erro, o servidor vai retornar {OP_ERROR, CT_RESULT, *errcode*} ou {OP_ERROR, CT_NONE} (dependendo do pedido), onde *errcode* é o código do erro retornado ao executar a operação na árvore do servidor (em geral, -1).

  Note que o caso em que uma chave não é encontrada no *get* não deve ser considerado como erro. Neste caso o servidor deve responder com uma resposta normal (OP_GET+1) mas definindo um `data_t` com `size=0` e `data=NULL`.

6. Makefile

Deve-se também escrever um `Makefile` que permita compilar os dois programas, com os seguintes *targets*:

- **client-lib.o**: Compilar os ficheiros `client_stub.c`, `network-client.c`, `data.c` e `entry.c` (e eventuais ficheiros adicionais "`.c`" criados na concretização do cliente), sem a aplicação interactiva com o *main()*, criando uma biblioteca **client-lib.o** (Usar linker *ld* com opção *-r*: `ld -r in1.o in2.o [...] -o client-lib.o`);
- **tree-client**: Compilar todo o código que pertence ao cliente, criando a aplicação cliente **tree-client**;
- **tree-server**: Compilar todo o código que pertence ao servidor, criando a aplicação servidor **tree-server**;
- **clean**: Remover todos os ficheiros criados pelos *targets* acima.

7. Entrega

A entrega do projeto 2 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-projeto2.zip`** (XX é o número do grupo).
2. Submeter o ficheiro **`grupoXX-projeto2.zip`** na página da disciplina no moodle da FCUL, utilizando a atividade disponibilizada para tal. Apenas um dos elementos do grupo deve submeter e todos os elementos têm de confirmar a submissão.

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;
  - lib: para armazenar bibliotecas;
  - binary: para armazenar os ficheiros executáveis.
- um ficheiro `Makefile` que satisfaça os requisitos descritos na Secção 7. Não devem ser incluídos no ficheiro ZIP os ficheiros objeto (.o) ou executáveis. Quaisquer outros ficheiros (por exemplo, de teste) também não deverão ser incluídos no ficheiro ZIP.

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

- **Se não for incluído um `Makefile`, se o mesmo não satisfizer os requisitos indicados, 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 01/11/2020 até às 23:59hs.

Após esta data, a submissão do trabalho através do Moodle deixará de ser permitida. Também, cada grupo ficará sem acesso de escrita à diretoria de entrega.

8. 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.

[4] https://developers.google.com/protocol-buffers/docs/overview