Trabalho de Projeto

2ª Fase

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 cumprir os objetivos de cada projeto, de forma a não hipotecar os projetos seguintes**.

O objetivo geral do projeto será concretizar um serviço de armazenamento baseado em pares chave-valor, nos moldes da interface *java.util.Map* da API Java e similar ao utilizado pela *Amazon DynamoDB* para dar suporte a muitos serviços Web [1]. Neste sentido, as estruturas de dados utilizadas para armazenar esta informação são uma **lista encadeada simples** [2] e uma ***hash table*** [3], dada a sua elevada eficiência ao nível da pesquisa.

Na 1ª fase do projeto foram definidas as estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na *hash table*. Também foi construído um módulo para a serialização de dados, que teve como objetivo familiarizar os alunos com a necessidade de serializar dados para a comunicação em aplicações distribuídas.

A 2ª fase do projeto tem como objetivo implementar um sistema cliente-servidor simples, sendo o servidor responsável por manter uma *hash table* (usando os módulos construídos na 1ª fase do projeto) e sendo o cliente responsável por comunicar com o servidor para realizar operações nesta tabela. Nesta fase do projeto, em vez de se utilizar o módulo de serialização desenvolvido na 1ª fase (que teria de ser consideravelmente estendido com outras funções), será utilizado o ***Protocol Buffers*** da Google [5]. Este é um mecanismo que permite automatizar, a partir de um ficheiro com a descrição dos dados a serem enviados e recebidos, a criação das funções para **serializar** (codificar) e **de-serializar** (descodificar) estes dados.

Será, portanto, necessário programar duas aplicações, um servidor e um cliente. O servidor será responsável por manter uma *hash table* em memória, receber os pedidos dum cliente para executar operações na tabela e responder ao cliente. O cliente será responsável por apresentar uma interface ao utilizador, enviando comandos fornecidos por este para o servidor, e apresentando os resultados recebidos. Em mais detalhe, 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 *hash table* local;
    b. enviará depois ao cliente a resposta transformada em mensagem;
3. por sua vez, o cliente interpretará a mensagem de resposta e;
4. apresentará o resultado ao utilizador, ficando depois pronto para receber a próxima operação.

O objetivo final é fornecer às aplicações que usariam esta tabela um modelo de comunicação tipo RPC (*Remote Procedure Call*)[1], onde vários clientes acedem a uma mesma *hash table* partilhada. Nesta fase do projeto apenas será atendido um cliente de cada vez, sendo que nas próximas fases o sistema passará a suportar vários clientes em simultâneo. Para a concretização desta fase do projeto será necessário combinar o código desenvolvido na 1ª fase do projeto com o *Protocol Buffers* e com as funções de sistema para a comunicação usando *sockets* TCP.

Espera-se uma grande fiabilidade por parte do servidor, portanto **todas as condições de erro devem ser tratadas de alguma forma.** É igualmente necessário garantir que a gestão de memória no cliente e especialmente no servidor é feita de forma correta (evitando *memory leaks* ou acesso a zonas de memória inválidas). É importante notar que em sistemas cliente-servidor os servidores ficam em funcionamento permanente durante muito tempo (dias, meses ou até anos) e não é suposto que parem (*crash*) por causa de erros. Por exemplo, 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.

2 Descrição Específica

O objetivo específico da 2ª fase do projeto 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 *hash table*. Esta será implementada no programa servidor (utilizando o código

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

desenvolvido na 1ª fase do projeto), que deverá oferecer um conjunto de procedimentos acessíveis remotamente, para executar determinadas operações na tabela. Assim, o cliente terá de ser programado para enviar ao servidor as operações solicitadas pelo utilizador, recebendo e apresentando as respostas. Por sua vez, o servidor terá de ser programado para esperar um pedido de ligação de um cliente, passando a estar depois pronto 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, a 2ª fase do projeto consiste na concretização de:

1. Dois programas, escritos na linguagem C, a serem executados da seguinte forma:
   a. **`server_hashtable`** `<server>:<port> <n_lists>`
      `<server>` é o endereço IP ou nome ao qual o servidor da *hash table* deverá fazer *bind*.
      `<port>` é o número do porto TCP ao qual o servidor se deve associar (fazer *bind*).
      `<n_lists>` é o número de listas usado na criação da *hash table* no servidor.
   b. **`client_hashtable`** `<server>:<port>`
      `<server>` é o endereço IP ou nome do servidor da *hash table*.
      `<port>` é o número do porto TCP onde o servidor está à espera de ligações.
2. Módulos de adaptação do lado do cliente (*stub*) e do lado do servidor (*skeleton*) que permitam preparar as mensagens que são serializadas para envio, e processar o conteúdo das mensagens recebidas após terem sido de-serializadas. Para a serialização e de-serialização, será fornecido um ficheiro *.proto* com a especificação do formato genérico das mensagens que serão usadas para a comunicação entre o cliente e o servidor, a partir do qual serão gerados, usando o *Protocol Buffers*, os ficheiros `.c/.h` que definem, em C, o formato das mensagens e as funções necessárias para serializar (codificar) e de-serializar (descodificar) as mesmas.

O programa cliente (`client_hashtable`) permitirá o envio de comandos via rede para invocar operações (na *hash table*) implementadas pelo servidor. Por outro lado, o programa servidor (`server_hashtable`) receberá os comandos vindos do(s) cliente(s), executando-os sobre a tabela e devolvendo a resposta ao(s) cliente(s). Estes comandos são constituídos por *opcodes* representativos de operações (`OP_***`) e tipo de conteúdo (`CT_***`) a operar na *hash table*.

A Tabela 1 apresenta os comandos e as respetivas sintaxes, que os clientes podem requerer e a que o servidor pode responder.

COMANDO UTILIZADOR	MENSAGEM DE PEDIDO	MENSAGEM DE RESPOSTA
put <key> <value>	`OP_PUT CT_ENTRY <entry>`	`OP_PUT+1 CT_NONE` `OP_ERROR CT_NONE`
get <key>	`OP_GET CT_KEY <key>`	`OP_GET+1 CT_VALUE <value>` `OP_ERROR CT_NONE`
del <key>	`OP_DEL CT_KEY <key>`	`OP_DEL+1 CT_NONE` `OP_ERROR CT_NONE`
size	`OP_SIZE CT_NONE`	`OP_SIZE+1 CT_RESULT <size>` `OP_ERROR CT_NONE`
getkeys	`OP_GETKEYS CT_NONE`	`OP_GETKEYS+1 CT_KEYS <keys>` `OP_ERROR CT_NONE`
gettable	`OP_GETTABLE CT_NONE`	`OP_GETTABLE+1 CT_TABLE <entries>` `OP_ERROR CT_NONE`
quit	`--`	`--`

Tabela 1: Definição dos comandos e as respetivas mensagens de pedido e de resposta.

Caso a operação sobre a *hash table* seja bem-sucedida, o *opcode* de resposta é igual ao *opcode* do pedido incrementado de uma unidade. Por outro lado, caso a operação no servidor não seja bem-sucedida, o *opcode* da resposta será `OP_ERROR`.
É assim necessário, nos programas cliente e servidor, construir mensagens que respeitem este formato. Para facilitar a definição das estruturas de dados que vão constituir uma mensagem, e para automatizar o processo de serialização dos dados (mensagem a enviar) num buffer, e de-serialização do buffer nos correspondentes dados (mensagem recebida), será usado o *Protocol Buffers*.
Em concreto, para se usar o *Protocol Buffers* é necessário, em primeiro lugar, definir o formato das mensagens a

enviar/receber. Esta definição é feita num ficheiro `.proto`, usando a sintaxe "`proto3`". Neste projeto, a definição será feita no ficheiro `htmessages.proto`, que será fornecido a todos os grupos. Desta forma, todos os grupos terão de seguir a mesma definição do formato das mensagens, pelo que **será de esperar que qualquer programa cliente consiga interagir com qualquer programa servidor**. Estando assegurada a interoperabilidade entre qualquer cliente e qualquer servidor que implemente corretamente a interface definida, serão também disponibilizados binários de um cliente e de um servidor, que os grupos poderão usar para testar, respetivamente, o seu próprio servidor ou cliente. Estes binários funcionarão no ambiente Linux dos laboratórios do DI.

No ficheiro `htmessages.proto` são especificados dois tipos de mensagem: uma mensagem com o nome `message_t`, que será usada para o envio de todos os pedidos e todas as respostas, e uma mensagem `entry_t`, que será usada apenas como submensagem de `message_t`. Dados estes nomes, as estruturas que serão criadas pelo *Protocol Buffers* serão do tipo `MessageT` e `EntryT`. É esta estrutura `MessageT` (que pode incluir estruturas `EntryT`) que os grupos terão de usar na implementação, preenchendo-a com os dados a enviar (e a serializar), e de onde terão de ler os dados recebidos (já de-serializados).

De seguida é apresentado o ficheiro `htmessages.proto`.

```
/* Ficheiro htmessages.proto
 */

syntax = "proto3";

message entry_t               /* Formato da mensagem EntryT */
{
      string key        = 1;
      bytes  value      = 2;
}

message message_t             /* Formato da mensagem MessageT */
{
      enum Opcode {           /* Opcodes da mensagem */
            OP_BAD      = 0;
            OP_PUT      = 10;
            OP_GET      = 20;
            OP_DEL      = 30;
            OP_SIZE     = 40;
            OP_GETKEYS  = 50;
            OP_GETTABLE = 60;
            OP_ERROR    = 99;
      }

      enum C_type {           /* Códigos para conteúdos da mensagem */
            CT_BAD      = 0;
            CT_ENTRY    = 10;
            CT_KEY      = 20;
            CT_VALUE    = 30;
            CT_RESULT   = 40;
            CT_KEYS     = 50;
            CT_TABLE    = 60;
            CT_NONE     = 70;
      }

/* Campos disponíveis na mensagem genérica (cada mensagem concreta, de
 * pedido ou de resposta, usa apenas os campos que necessitar)
 */
      Opcode            opcode      = 1;
      C_type            c_type      = 2;
      entry_t           entry       = 3;
      string            key         = 4;
      bytes             value       = 5;
      sint32            result      = 6;
      repeated string   keys        = 7;
      repeated entry_t  entries     = 8;
};
```

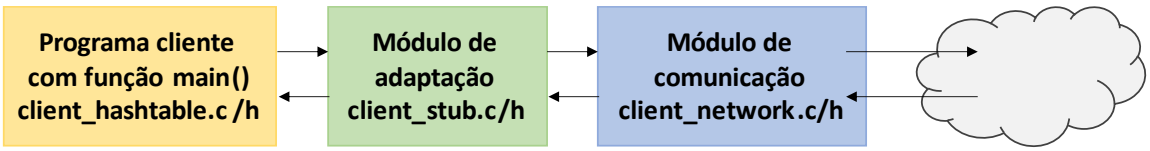
Para além de ser fornecido o ficheiro `htmessages.proto` e binários do cliente e do servidor, para guiar a concretização dos programas cliente e servidor são ainda fornecidos ficheiros `.h` com os cabeçalhos de algumas funções que devem ser concretizadas. Os ficheiros fornecidos, incluindo o ficheiro `htmessages.proto` e os ficheiros `htmessages.pb-c.c/h` que dele resultam, **não podem ser alterados**. As concretizações das funções definidas nos ficheiros `X.h` devem ser feitas nos respetivos ficheiros `X.c`, utilizando os algoritmos e métodos que o grupo achar convenientes. É possível criar ficheiros `X-private.h` para acrescentar outras definições que o grupo entenda necessárias, as quais serão também implementadas em `X.c`. Os ficheiros `.h` apresentados neste documento serão disponibilizados na página da disciplina.

3 Cliente

O cliente usa alguns dos módulos já desenvolvidos na 1ª fase do projeto (`block.c/h` e `entry.c/h`), juntamente com três módulos adicionais:

- Programa cliente com a função `main()` (`client_hashtable.c`)
- Módulo de adaptação das chamadas do cliente, RPC *stub* (`client_stub.c/h`)
- Módulo de comunicação (`client_network.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: `client_hashtable.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 a 1ª coluna da Tabela 1):

```
put <key> <value>
get <key>
del <key>
size
getkeys
gettable
quit
```

Note que o valor `<value>` no comando `put` deverá conter todos os caracteres após a chave (`<key>`), até ao fim da linha, **podendo por isso conter espaços**. Por outro lado, no desenvolvimento do código (nomeadamente módulos `client_stub.c` e `server_skeleton.c`) deve-se considerar que os dados `<value>` associados à chave `<key>` **podem não ser *strings*** e podem ser dados arbitrários (e.g., imagens, vídeos, executáveis, 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 `MessageT` com o tipo da operação (`opcode`), com o tipo de conteúdo (`c_type`) necessário para realizar a operação, e com esse conteúdo (`entry`, `value`, `keys`, etc.). Esta estrutura é então passada para o módulo de comunicação, que a irá processá-la e devolver uma resposta (veja a próxima secção).

De notar que a estrutura `MessageT` é definida automaticamente pelo *Protocol Buffers* a partir do ficheiro `htmessages.proto` e a definição estará no ficheiro `htmessages.pb-c.h`,

A interface a ser oferecida ao cliente é apresentada de seguida.

```
#ifndef _CLIENT_STUB_H
#define _CLIENT_STUB_H

#include "block.h"
#include "entry.h"

/* Remote table, que deve conter as informações necessárias para comunicar
 * com o servidor. A definir pelo grupo em client_stub-private.h
 */
struct rtable_t;

/* Função para estabelecer uma associação entre o cliente e o servidor,
 * em que address_port é uma string no formato <hostname>:<port>.
 * Retorna a estrutura rtable preenchida, ou NULL em caso de erro.
 */
struct rtable_t *rtable_connect(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, ou -1 em caso de erro.
 */
int rtable_disconnect(struct rtable_t *rtable);

/* Função para adicionar uma entrada na tabela.
 * Se a key já existe, vai substituir essa entrada pelos novos dados.
 * Retorna 0 (OK, em adição/substituição), ou -1 (erro).
 */
int rtable_put(struct rtable_t *rtable, struct entry_t *entry);

/* Retorna a entrada da tabela com chave key, ou NULL caso não exista
 * ou se ocorrer algum erro.
 */
struct block_t *rtable_get(struct rtable_t *rtable, char *key);

/* Função para remover um elemento da tabela. Vai libertar
 * toda a memoria alocada na respetiva operação rtable_put().
 * Retorna 0 (OK), ou -1 (chave não encontrada ou erro).
 */
int rtable_del(struct rtable_t *rtable, char *key);

/* Retorna o número de elementos contidos na tabela ou -1 em caso de erro.
 */
int rtable_size(struct rtable_t *rtable);

/* Retorna um array de char* com a cópia de todas as keys da tabela,
 * colocando um último elemento do array a NULL.
 * Retorna NULL em caso de erro.
 */
char **rtable_get_keys(struct rtable_t *rtable);

/* Liberta a memória alocada por rtable_get_keys().
 */
void rtable_free_keys(char **keys);

/* Retorna um array de entry_t* com todo o conteúdo da tabela, colocando
 * um último elemento do array a NULL. Retorna NULL em caso de erro.
 */
struct entry_t **rtable_get_table(struct rtable_t *rtable);

/* Liberta a memória alocada por rtable_get_table().
 */
void rtable_free_entries(struct entry_t **entries);

#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`). Sempre que ocorrer um erro do lado do servidor, este deverá enviar uma resposta com *opcode* `OP_ERROR` e conteúdo `CT_NONE`.

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

O módulo `client_network.c` vai concretizar as funções para estabelecer e terminar uma ligação ao servidor e, na função `network_send_receive()`, vai enviar a mensagem de pedido e receber a mensagem de resposta.
De forma mais detalhada, o que **tem de ser feito** nesta função é o seguinte:

1. Determinar a dimensão do buffer onde serão colocados aos dados a enviar, com base na mensagem contida na estrutura `MessageT` que é recebida como argumento;
2. Serializar a mensagem contida na estrutura, colocando-a num buffer com dimensão apropriada;
3. Enviar um ***short (2 bytes)*** indicando a dimensão do buffer que será enviado;
4. Enviar o buffer;
5. Receber um ***short*** indicando a dimensão do buffer onde será recebida a resposta;
6. Receber a resposta colocando-a num buffer de dimensão apropriada;
7. De-serializar a resposta para uma estrutura `MessageT` que será retornada.

O módulo de comunicação `client_network.c` tem a seguinte interface:

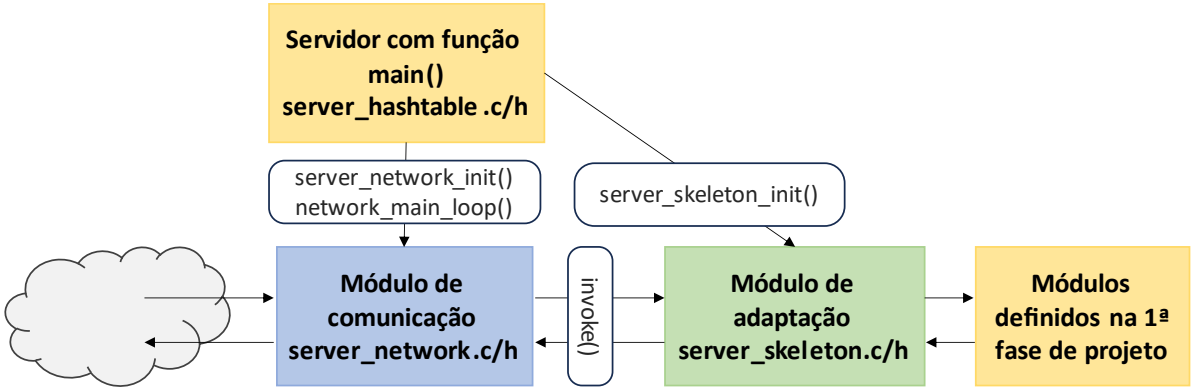
```
#ifndef _CLIENT_NETWORK_H
#define _CLIENT_NETWORK_H

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

/* Esta função deve:
 * - Obter o endereço do servidor (struct sockaddr_in) com base na
 *   informação guardada na estrutura rtable;
 * - Estabelecer a ligação com o servidor;
 * - Guardar toda a informação necessária (e.g., descritor do socket)
 *   na estrutura rtable;
 * - Retornar 0 (OK) ou -1 (erro).
 */
int network_connect(struct rtable_t *rtable);

/* Esta função deve:
 * - Obter o descritor da ligação (socket) da estrutura rtable_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;
 * - Tratar de forma apropriada erros de comunicação;
 * - Retornar a mensagem de-serializada ou NULL em caso de erro.
 */
MessageT *network_send_receive(struct rtable_t *rtable, MessageT *msg);

/* Fecha a ligação estabelecida por network_connect().
 * Retorna 0 (OK) ou -1 (erro).
 */
int network_close(struct rtable_t *rtable);

#endif
```

4 Servidor

O **servidor** a implementar usa os módulos `block.c/h`, `entry.c/h`, `list.c/h` e `table.c/h`, e é composto por três partes adicionais:

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

A figura abaixo ilustra a interação entre os três módulos, onde o **módulo de adaptação** do servidor permite a interação **entre o módulo de comunicação e a *hash table*, permitindo também a inicialização da tabela (feita na função** `main()`**)**.



4.1 Aplicação servidor: `server_hashtable.c`

O servidor concretiza uma *hash table* que pode ser acedida no porto definido através da 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 *threads* para o atendimento simultâneo de vários clientes.
A aplicação deve fazer essencialmente três coisas: inicializar o servidor, inicializar a tabela, e colocar o servidor em *loop* a receber pedidos de ligação e, após estabelecer uma ligação, receber e responder aos pedidos de operações na *hash table*.

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

O módulo `server_network.c` vai permitir inicializar o servidor (que deverá usar *sockets* TCP) e fechar o *socket* do servidor, respetivamente através das funções `server_network_init()` e `server_network_close()`. A função `network_main_loop()` é responsável pela interação com a rede (receber pedidos de ligação, receber pedidos de operações na tabela e enviar respostas) e pela interação com o *skeleton* (`server_skeleton.c`).
Assim sendo, a função recebe sequências de bytes que constituem as mensagens serializadas com os pedidos dos clientes, realiza a de-serialização destes bytes, construindo uma mensagem (estrutura `MessageT`), a qual é entregue ao *skeleton* para processamento. No retorno, a mensagem enviada ao *skeleton* vem preenchida com o resultado da operação (ou seja, a estrutura `MessageT` é reaproveitada e alterada pelo *skeleton*), tendo de ser serializada numa sequência de bytes que são enviados ao cliente através da ligação TCP.
Tal como do lado do cliente, também o servidor **tem de receber/enviar um short (2 bytes)** antes receber/enviar as respetivas mensagens, para se saber a dimensão das mensagens (e dos *buffers* onde serão recebidas). Não deve ser esquecida a necessidade de libertar toda a memória reservada que já não é necessária, para evitar *memory leaks*!
O módulo de comunicação `server_network.c` tem a seguinte interface:

```
#ifndef _SERVER_NETWORK_H
#define _SERVER_NETWORK_H

#include "table.h"
#include "htmessages.pb-c.h"

/* Função para preparar um socket de receção de pedidos de ligação
 * num determinado porto.
 * Retorna o descritor do socket ou -1 em caso de erro.
 */
int server_network_init(short port);

/* A função network_main_loop() 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
     na tabela table;
 * - Esperar a resposta do skeleton;
 * - Enviar a resposta ao cliente usando a função network_send.
 * A função não deve retornar, a menos que ocorra algum erro. Nesse
 * caso retorna -1.
 */
int network_main_loop(int listening_socket, struct table_t *table);
```

```
/* A função network_receive() 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 MessageT.
 * Retorna a mensagem com o pedido ou NULL em caso de erro.
 */
MessageT *network_receive(int client_socket);

/* A função network_send() deve:
 * - Serializar a mensagem de resposta contida em msg;
 * - Enviar a mensagem serializada, através do client_socket.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int network_send(int client_socket, MessageT *msg);

/* Liberta os recursos alocados por server_network_init(), nomeadamente
 * fechando o socket passado como argumento.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int server_network_close(int socket);

#endif
```

4.3 RPC skeleton: `server_skeleton.c`

O *skeleton*, a concretizar em `server_skeleton.c`, serve para transformar uma mensagem do cliente (a qual foi de-serializada pela camada de rede) numa chamada (ou possivelmente em várias, caso o pedido não possa ser realizado com apenas uma chamada) da respetiva função do módulo `table.c`. Depois de realizar o pedido e de, possivelmente, obter dados de resposta, o *skeleton* deve reutilizar a estrutura `MessageT` que recebeu, para preparar a resposta a enviar de volta à camada de rede.

As funções `server_skeleton_init()` e `server_skeleton_destroy()` servem fundamentalmente para criar e destruir a *hash table* a ser mantida pelo servidor. A *hash table* devolvida pela função `server_skeleton_init()` deve ser passada para a função `network_main_loop()`, que a utilizará quando invocar as operações através da função `invoke()`.

```
#ifndef _SERVER_SKELETON_H
#define _SERVER_SKELETON_H

#include "table.h"
#include "htmessages.pb-c.h"

/* Inicia o skeleton da tabela.
 * O main() do servidor deve chamar esta função antes de poder usar a
 * função invoke(). O parâmetro n_lists define o número de listas a
 * serem usadas pela tabela mantida no servidor.
 * Retorna a tabela criada ou NULL em caso de erro.
 */
struct table_t *server_skeleton_init(int n_lists);

/* Liberta toda a memória ocupada pela tabela e todos os recursos
 * e outros recursos usados pelo skeleton.
 * Retorna 0 (OK) ou -1 em caso de erro.
 */
int server_skeleton_destroy(struct table_t *table);

/* Executa na tabela table a operação indicada pelo opcode contido em msg
 * e utiliza a mesma estrutura MessageT para devolver o resultado.
 * Retorna 0 (OK) ou -1 em caso de erro.
*/
int invoke(MessageT *msg, struct table_t *table);

#endif
```

A função `invoke()` recebe a mensagem de-serializada numa estrutura `MessageT`, interpreta o campo `opcode` da mensagem para selecionar a operação a executar na tabela, e realiza a operação (ou as várias operações necessárias)

na tabela (também recebida como argumento). Depois de executar a operação, coloca o resultado na mesma estrutura `MessageT`, alterando os campos `opcode` e `c_type`, bem como os que forem necessários para responder ao cliente. Caso a operação na tabela tenha sido executada com sucesso, o `opcode` da estrutura `MessageT` será o valor do `opcode` do pedido incrementado de uma unidade e o campo `c_type` terá o valor adequado ao tipo de resultado retornado. Por outro lado, caso ocorra erro na operação executada na tabela, o campo `opcode` receberá o valor de `OP_ERROR` e o campo `c_type` o valor de `CT_NONE` (ver Tabela 1).

5 Observações

Algumas observações e dicas úteis:

- Recomenda-se a criação de funções `read_all()` e `write_all()` que vão receber e enviar *buffers* de qualquer dimensão pela rede, assegurando que só retornam quando o buffer for totalmente recebido ou enviado. Isto é necessário (ou pelo menos importante) dado que as funções `read()/write()` em *sockets* nem sempre leem/escrevem tudo o que lhes é pedido. Um bom sítio para concretizar essas funções é num modulo adicional `message.c` a ser usado quer do lado do cliente, quer do lado do servidor, com o protótipo das funções definido em `message-private.h`.
- 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 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 tabela 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_NONE`} independentemente do tipo de erro.

6 Ficheiros disponibilizados

Para a realização deste projeto serão disponibilizadas versões funcionais dos módulos que foram solicitados na 1ª fase do projeto: `block.o`, `entry.o`, `list.o` e `table.o`. Os grupos que não tenham conseguido concluir com sucesso o desenvolvimento da 1ª fase do projeto podem utilizar estes módulos se preferirem. Também são fornecidos os ficheiros `.h` apresentados neste enunciado, bem como o ficheiro `htmessages.proto`.
Finalmente, como já referido, serão ainda disponibilizados os binários de um cliente e de um servidor, que podem ser usados para testes. O protocolo de comunicação implementado nestes dois binários é o que está descrito neste enunciado. Assim, se os programas desenvolvidos pelos grupos respeitarem o protocolo, a interoperabilidade estará assegurada.

7 Makefile

Deve ser construído um `Makefile` que permita compilar os dois programas e que tenha, pelo menos, os seguintes *targets*:

- `all`: criar (pelo menos) os targets `libtable`, `client_hashtable` e `server_hashtable`;
- `libtable`: Compilar os ficheiros `block.c`, `entry.c`, `list.c`, e `table.c`, criando os respetivos ficheiro objeto, e colocando-os numa biblioteca estática `libtable.a` (usar o comando `ar` com as opções `-r`, `-c` e `-s`: `ar -rcs libtable.a block.o entry.o list.o table.o`. Caso sejam usados os ficheiros objeto da 1ª fase do projeto disponibilizados aos alunos, deverão ser estes a ser colocados na biblioteca;
- `client_hashtable`: Compilar todo o código que pertence ao cliente, criando a aplicação `client_hashtable` (ligar todos os ficheiros objetos necessários e a biblioteca `libtable.a`);
- `server_hashtable`: Compilar todo o código que pertence ao servidor, criando a aplicação `server_hashtable` ligar todos os ficheiros objetos necessários e a biblioteca `libtable.a`);
- `clean`: Remover todos os ficheiros criados pelos *targets* acima. Nota: se forem usados os ficheiros objeto da 1ª fase do projeto disponibilizados aos alunos, estes não precisam de ser removidos pelo `clean`.

8 Entrega

A entrega da 2ª fase do projeto 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 **`SD-XX-projeto2.zip`** (**`XX`** é o número do grupo).
2. Submeter o ficheiro **`SD-XX-projeto2.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 **`SD-XX`**, onde **`XX`** é o número do grupo. Nela 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`;
  - **`lib`**: para armazenar bibliotecas;
  - **`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 que são construídos pelo `Makefile`. Caso sejam usados os ficheiros objeto da 1ª fase do projeto disponibilizados aos grupos, estes **devem** 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 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 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 neste mesmo ambiente.

O prazo de entrega é dia 03/11/2024, até às 23:59 horas.

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

9 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 até a data limite de entrega do projeto **sofrem uma penalização na nota de 20%.**
- 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 (03/11/2024, até às 23:59 horas).

10 Plágios

Não é permitido aos alunos partilharem códigos com soluções, ainda que parciais, de nenhuma parte do projeto com outros alunos (nem através do Fórum da disciplina, nem por qualquer outro meio). Além disso, todos os códigos serão testados por um verificador de plágio. Caso alguma irregularidade seja encontrada, os projetos de todos os alunos envolvidos serão anulados e o caso será reportado aos órgãos responsáveis em Ciências@ULisboa.

Chamamos a atenção para o facto das plataformas generativas baseadas em Inteligência Artificial (e.g., o ChatGPT e o GitHub Co-Pilot) (1) gerarem um número limitado de soluções diferentes para o mesmo problema, (2) podem não resolver corretamente as alíneas descritas no projeto, e (3) incluem padrões característicos deste tipo de ferramenta, as quais podem vir a ser detetáveis pelos verificadores de plágio. Desta forma, recomendamos fortemente que os alunos não submetam trechos de código gerados por este tipo de ferramenta a fim de evitar riscos desnecessários.

Por fim, é responsabilidade de cada aluno garantir que a sua *home*, as suas diretorias e os seus ficheiros de código estão protegidos contra a leitura de outras pessoas (que não o utilizador dono dos mesmos). Por exemplo, se os ficheiros estiverem gravados na sua área de aluno nos servidores de Ciências@ULisboa, então todos os itens mencionados anteriormente devem ter as permissões de acesso `700`. Se os ficheiros estiverem no GitHub, garantam que o conteúdo do vosso repositório não esteja visível publicamente. Caso contrário, a sua participação num eventual plágio será considerada ativa.

11 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.

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