Ciências ULisboa | Informática

2015/16

Sistemas Distribuídos

Projeto 4

1. Descrição geral

A componente teórico-prática da disciplina de sistemas distribuídos está divida em cinco projetos, 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 hipotecarem os projetos seguintes.

O objetivo geral do projeto será concretizar um serviço de armazenamento de pares chave-valor similar ao utilizado pela *Amazon* para dar suporte aos seus serviços Web. A estrutura de dados utilizada para armazenar esta informação é uma **tabela *hash*** com ***chaining***.

No projeto 1 foram definidas algumas estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na tabela, bem como para implementar a técnica de *chaining* utilizando uma lista ligada. No projeto 2 implementaram-se as funções necessárias para construir a tabela *hash* e para codificar e descodificar estruturas complexas em mensagens. No projeto 3 concretizou-se a tabela *hash* num servidor, oferecendo ao cliente uma interface similar à que foi concretizada no projeto 2 para a tabela, e implementou-se um programa cliente que invoca operações para enviar ao servidor.

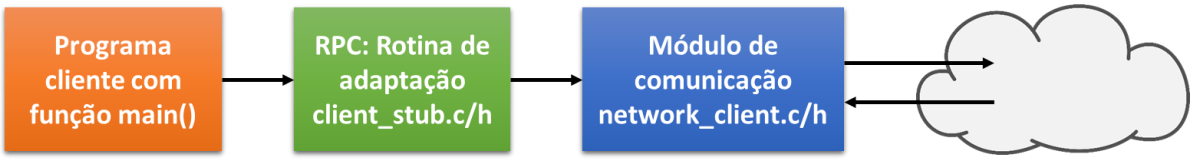
O projeto 4 tem 3 objetivos:

1. garantir que o servidor suporta múltiplos clientes simultaneamente através de multiplexagem de I/O (usando a chamada ao sistema, poll). Além disso também deve ser concretizado um mecanismo muito simples de tolerância a faltas;
2. fornecer um modelo de comunicação **tipo RPC** (*Remote Procedure Call*), às aplicações cliente que usam a tabela *hash* do servidor;
3. iniciar a implementação de uma tabela persistente em disco. Uma vez que a tabela é mantida em memória volátil, ela pode ser perdida em caso de paragem do servidor. Através da tabela persistente o servidor será capaz de recuperar de uma falha por paragem, obtendo o seu estado pré-falha, desde que o disco da máquina onde ele está a executar se mantenha correto.

Espera-se de novo uma grande fiabilidade por parte do servidor, portanto não pode 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 tabela partilhada (no caso da Amazon, se o serviço que mantém as cestas de compras dos clientes não funciona, a empresa não vende, perde milhões de dólares por hora… e os programadores são despedidos!).

2. Restruturação do cliente

Além das funcionalidades definidas no projeto anterior, em que o cliente é composto por uma aplicação cliente interativa (table_client.c) com a função main() e uma biblioteca de comunicação (network_client.h/c), de forma a construir um modelo de comunicação tipo RPC, **vai-se agora incluir uma rotina de adaptação** do cliente, isto é, **um RPC *stub*** (client_stub.h/c) **entre a aplicação cliente e a biblioteca de comunicação**.



A funcionalidade que até agora estava implementada na aplicação cliente interativa vai agora ser dividida entre esta e o RPC stub. Notem que isto vai implicar, naturalmente, alterações no ficheiro fonte table_client.c. O ***stub* serve de intermediário** para o cliente e por isso a sua função é **"esconder" deste todos os detalhes relativos à comunicação**.

2.1. RPC stub (client_stub.c)

Além das funções para se ligar e desligar de um servidor, **o *stub* concretiza funções que possibilitem todas as operações que podem ser executadas na tabela**. Cada função vai preencher uma estrutura *message_t* com o *opcode* (tipo da operação) e os restantes campos usados de acordo com o conteúdo requerido pela mensagem, e vai passar esta estrutura para o módulo de comunicação, recebendo depois a(s) resposta(s) deste módulo.

A interface a ser oferecida ao cliente é a seguinte:

```
#ifndef _CLIENT_STUB_H
#define _CLIENT_STUB_H

#include "data.h"

/* Remote table. A definir pelo grupo em client_stub-private.h
 */
struct rtable_t;

/* Função para estabelecer uma associação entre o cliente e uma tabela
 * remota num servidor.
 * address_port é uma string no formato <hostname>:<port>.
 * retorna NULL em caso de erro .
 */
struct rtable_t *rtable_bind(const char *address_port);

/* Termina a associação entre o cliente e a tabela remota, e liberta
 * toda a memória local.
 * Retorna 0 se tudo correr bem e -1 em caso de erro.
 */
int rtable_unbind(struct rtable_t *rtable);

/* Função para adicionar um par chave valor na tabela remota.
 * Devolve 0 (ok) ou -1 (problemas).
 */
int rtable_put(struct rtable_t *rtable, char *key, struct data_t *value);

/* Função para substituir na tabela remota, o valor associado à chave key.
 * Devolve 0 (OK) ou -1 em caso de erros.
 */
int rable_update(struct rtable_t *rtable, char *key, struct data_t *value);

/* Função para obter da tabela remota o valor associado à chave key.
 * Devolve NULL em caso de erro.
 */
struct data_t *rtable_get(struct rtable_t *table, char *key);

/* Função para remover um par chave valor da tabela remota, especificado
 * pela chave key.
 * Devolve: 0 (OK) ou -1 em caso de erros.
```

```
 */
int rtable_del(struct rtable_t *table, char *key);


/* Devolve número de elementos na tabela remota.
 */
int rtable_size(struct rtable_t *rtable);

/* Devolve um array de char * com a cópia de todas as keys da
 * tabela remota, e um último elemento a NULL.
 */
char **rtable_get_keys(struct rtable_t *rtable);

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

#endif
```

2.2. Biblioteca de comunicação do cliente: network_client.c

Este módulo deve agora concretizar também um mecanismo simples de **tolerância a faltas**. Se o envio ou a receção dos dados na função network_send_receive falha, a função deve dormir por RETRY_TIME segundos (constante a definir em network_client-private.h) e tentar uma única vez **restabelecer a ligação**, **reenviar o pedido** e **esperar pela resposta**. Se falhar novamente nalgum destes passos, deve retornar NULL para indicar o erro e o programa cliente deve apresentar uma mensagem adequada para o utilizador, continuando posteriormente a aceitar novos comandos.

3. Reestruturação do servidor

Além das funcionalidades definidas nos projetos 1 e 2, o servidor é composto por uma aplicação servidor (table_server.c) com a função main(). De forma a construir um modelo de comunicação tipo RPC, vai-se agora incluir uma rotina de adaptação do servidor, isto é, um RPC *skeleton* (table_skel.h/c). A funcionalidade que até agora estava implementada na aplicação servidor vai agora ser dividida entre esta e o RPC *skeleton*. A função do *skeleton* é "esconder" do servidor todos os detalhes relativos à tabela.

3.1. table_server.c

O servidor deverá suportar vários clientes em simultâneo. Como o servidor apenas necessita de tratar um pedido de cada vez, **não será** necessário recorrer à bifurcação do programa em *threads* ou processos filho (e.g., através da chamada de sistema *fork*). No entanto, como o protocolo TCP é um protocolo com ligação, será necessário que o servidor seja capaz de escutar simultaneamente várias *sockets* (ver página de manual da chamada ao sistema ***poll***). Esta função permite que se especifique um conjunto de descritores de ficheiros (e.g., *sockets*) e que se espere até que surja uma operação de I/O (e.g., dados disponíveis para leitura) num deles.

3.2. Skeleton (table_skel.c)

O *skeleton*, a concretizar em table_skel.c, serve para transformar uma mensagem do cliente (a qual foi de-serializada no servidor) numa chamada da respetiva função do módulo table. A interface é propositadamente muito simples:

```
#ifndef _TABLE_SKEL_H
#define _TABLE_SKEL_H

#include "message.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 0 (OK) ou -1 (erro, por exemplo OUT OF MEMORY)
 */
int table_skel_init(int n_lists);

/* Libertar toda a memória e recursos alocados pela função anterior.
 */
int table_skel_destroy();

/* Executa uma operação (indicada pelo opcode na msg_in) e retorna o
 * resultado numa mensagem de resposta ou NULL em caso de erro.
 */
struct message_t *invoke(struct message_t *msg_in);

#endif
```

As funções *table_skel_init* e *table_skel_destroy* servem fundamentalmente para criar e destruir a tabela a ser mantida pelo servidor. Já a função *invoke* interpreta o campo *opcode* de *msg_in* para selecionar a operação a executar nessa tabela. Depois de executar a operação, *invoke* aloca memória para a *struct message_t* de retorno que conterá o resultado da operação a enviar ao cliente.

3.3. Esqueleto da concretização do servidor

Nesta secção damos uma breve descrição da estrutura a ser concretizada no programa principal (*main*) do servidor (alterações ao ficheiro fonte table_server.c). A ideia é conjugar os métodos da API de *sockets POSIX* com a chamada ao sistema *poll*. A descrição não está escrita em código C compilável.

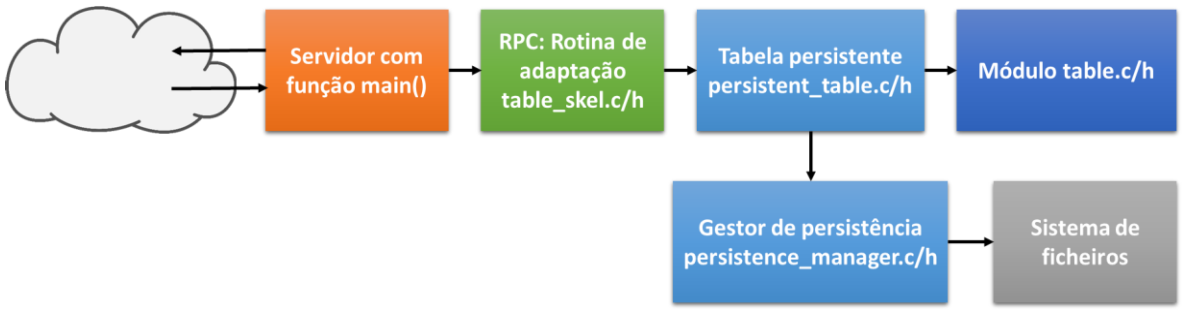
```
/* inicialização */
welcome_sockfd = socket(…);
bind(sockfd, porto obtido na linha de comando);
listen(sockfd);
table_skel_init();
adiciona sockfd a descriptor_set
while(not fatal_error){ /* espera por dados nos sockets abertos */
      res = poll(descriptor_set)
      if (res<0){
            if (errno != EINTR) fatal_error = TRUE
            continue;
      }
      if(welcome_sockfd tem dados para ler) { /* novo pedido de conexão */
            connection_sockfd = accept(welcome_sockfd);
            adiciona connection_sockfd a descriptor_set
      }
```

```
        /* um dos sockets de ligação tem dados para ler */
        for all socket s em descriptor_set (excluindo welcome_sockfd) {
              if (s tem dados para ler) {
                    nbytes = read_all(s, …);
                    if(read returns 0 bytes) {
                    /* sinal de que a conexão foi fechada pelo cliente */
                                close(s);
                                remove s de descriptor_set
                    } else {/* processamento da requisição e da resposta */
                          message = buffer_to_message(buffer);
                          msg_out = invoke(message);
                          buffer = message_to_buffer(msg_out);
                          write_all(s, buffer);
                    }
              }
              if (s com erro ou POLLHUP) {
                    close(s);
                    remove s de descriptor_set
              }
        }
}
table_skel_destroy();
/* fechar as ligações */
for all socket s em descriptor_set (incluindo sockfd) {
      close(s);
}
```

De notar que o algoritmo anterior apenas apresenta a ideia geral de como deve ser o main do servidor. Cabe aos alunos traduzir essa lógica para código C (ou inventar outro algoritmo).

3.4. A tabela persistente

A construção da tabela persistente passa pela concretização de dois novos módulos no servidor: a tabela persistente (`persistent_table.c`) e o gestor de persistência (`persistence_manager.c`).



O módulo `persistent_table.c`, de interface bastante similar a `table.c` e `remote_table.c`, desempenha a tarefa de agregar uma tabela em memória e um gestor de persistência, este último responsável por garantir que os dados desta tabela são persistidos em disco, e portanto sobrevivem a falhas do servidor.
O gestor de persistência encapsula todos os procedimentos relativos à manipulação de ficheiros de *log* e de armazenamento do estado da tabela. Neste projeto será apenas considerado um ficheiro de *log* onde são armazenadas as operações executadas na tabela, que modificaram o estado desta. Note que não é imposto nenhum formato dos *logs*, porém é sugerido que se usem os *arrays* de carateres das mensagens. Assim, quando o servidor recuperar de uma paragem, pode simplesmente ler do ficheiro de *log* os os *arrays* de carateres das mensagens, um a um, e para cada um deles chamar a função *invoke* do módulo `table_skel`, da mesma forma que acontece quando a mensagem chega através de uma *socket*.

3.4.1 Módulo persistent_table

A tabela persistente tem a interface definida em `persistent_table.h`.

```
#ifndef _PERSISTENT_TABLE_H
#define _PERSISTENT_TABLE_H

#include "data.h"
#include "table-private.h"
#include "persistence_manager-private.h"

struct ptable_t; /* A definir em persistent_table-private.h */

/* Abre o acesso a uma tabela persistente, passando como parâmetros a
 * tabela a ser mantida em memória e o gestor de persistência a ser usado
 * para manter logs e checkpoints. Retorna a tabela persistente criada ou
 * NULL em caso de erro.
 */
struct ptable_t *ptable_open(struct table_t *table,
                             struct pmanager_t *pmanager);

/* Fecha o acesso a uma tabela persistente. Todas as operações em table
 * devem falhar após um ptable_close.
 */
void ptable_close(struct ptable_t *ptable);

/* Liberta toda a memória e apaga todos os ficheiros utilizados pela
 * tabela persistente.
 */
void ptable_destroy(struct ptable_t *ptable);

/* Função para adicionar um par chave valor na tabela.
 * Devolve 0 (ok) ou -1 (problemas).
 */
int ptable_put(struct ptable_t *ptable, char *key, struct data_t *value);

/* Função para substituir na tabela, o valor associado à chave key.
 * Devolve 0 (OK) ou -1 em caso de erros.
 */
int pable_update(struct ptable_t *ptable, char *key, struct data_t *value);

/* Função para obter da tabela o valor associado à chave key.
 * Devolve NULL em caso de erro.
 */
struct data_t *ptable_get(struct ptable_t *ptable, char *key);
```

```
/* Função para remover um par chave valor da tabela, especificado pela
 * chave key.
 * Devolve: 0 (OK) ou -1 em caso de erros
 */
int ptable_del(struct ptable_t *ptable, char *key);

/* Devolve número de elementos na tabela.
 */
int ptable_size(struct ptable_t *ptable);

/* Devolve um array de char * com a cópia de todas as keys da tabela
 * e um último elemento a NULL.
 */
char **ptable_get_keys(struct ptable_t *ptable);

/* Liberta a memória alocada por ptable_get_keys().
 */
void ptable_free_keys(char **keys);

#endif
```

Note que as funções que não alteram o estado da tabela (*get*, *get_keys*, *size* e *free_keys*) devem ser executadas diretamente na tabela em memória (usando simplesmente as chamadas do módulo `table.c`). Adicionalmente, as demais funções (*put*, *update*, e *del*), devem acionar o gestor de persistência (ver a seguir) para fazer *log* das alterações na tabela.

3.4.2 Módulo persistence_manager

O gestor de persistência é definido pela interface `persistence_manager.h`.

```
#ifndef _PERSISTENCE_MANAGER_H
#define _PERSISTENCE_MANAGER_H

#include "table.h"

struct pmanager_t; /* A definir em persistence_manager-private.h */

/* Cria um gestor de persistência que armazena logs em filename+".log".
 * O parâmetro logsize define o tamanho máximo em bytes que o ficheiro de
 * log pode ter.
 * Note que filename pode ser um path completo. Retorna o pmanager criado
 * ou NULL em caso de erro.
 */
struct pmanager_t *pmanager_create(char *filename, int logsize);

/* Destrói o gestor de persistência pmanager. Retorna 0 se tudo estiver OK
 * ou -1 em caso de erro. Esta função não limpa o ficheiro de log.
 */
int pmanager_destroy(struct pmanager_t *pmanager);

/* Apaga o ficheiro de log gerido pelo gestor de persistência.
 * Retorna 0 se tudo estiver OK ou -1 em caso de erro.
 */
int pmanager_destroy_clear(struct pmanager_t *pmanager);

/* Retorna 1 caso existam dados no ficheiro de log e 0 caso contrário.
 */
int pmanager_has_data(struct pmanager_t *pmanager);

/* Adiciona uma string msg no fim do ficheiro de log associado a pmanager.
 * Retorna o numero de bytes escritos no log ou -1 em caso de problemas na
```

```
 * escrita (e.g., erro no write()), ou no caso em que o tamanho do ficheiro
 * de log após o armazenamento da mensagem seja maior que logsize (neste
 * caso msg não é escrita no log).
 */
int pmanager_log(struct pmanager_t *pmanager, char *msg);

/* Recupera o estado contido no ficheiro de log, na tabela passada como
 * argumento.
 */
int pmanager_fill_state(struct pmanager_t *pmanager,
                        struct table_t *table);

#endif
```

4. Observações

Algumas observações e dicas úteis:

- Sugere-se a seguinte sequência de desenvolvimento:
  - Alterar o servidor do projeto 3 para suportar simultaneamente vários clientes (secção 3.1). Podem testar com o cliente anterior;
  - Fazer a reestruturação do cliente (secção 2);
  - Implementar o módulo `table_skel` (secção 3.2; ver também pseudocódigo em 3.3);
  - Implementar o módulo `persistent_table` (seções 3.4 e 3.4.1);
  - Implementar o módulo `persistence_manager` (secções 3.4 e 3.4.2).
- Usar a função setsockopt(…, SO_REUSEADDR, …) para fazer com que o servidor consiga fazer *bind* a um porto registado como usado no kernel. Isto permite que o servidor seja reinicializado rapidamente, sem ter de esperar o tempo de limpeza das tabelas de portos usados do kernel.
- O servidor deve receber como parâmetro, além do porto, a string *filename* a ser usada como base para o ficheiro de *log*.
- Quando o servidor se inicia terá de executar a função `table_skel_init()` para que a tabela seja criada. Nessa função deverá ser aberta também a tabela persistente através da função `ptable_open()`. Na abertura da tabela persistente deverá ser criado o gestor de persistência usando a função `pmanager_create()`. Os alunos podem alterar a função table_skel_init de forma a que esta possa receber todos os parâmetros necessários. Podem fazer uma redefinição em table_skel-private.h.
- Após a criação do gestor de persistência deve-se verificar se o ficheiro de *log* existe e tem entradas (`pmanager_has_data()`):
  - Se existir e tiver entradas, significa que houve uma paragem do servidor. Devem então ser processadas as entradas no ficheiro, para que se recupere o estado da tabela anterior à paragem(`pmanager_fill_state()`);
  - caso contrário, se não existir deverá ser criado.
- Estando a infraestrutura do servidor criada, sempre que o servidor recebe um pedido, a função `invoke` de `table_skel` será executada. Esta, de acordo com o *opcode* na mensagem, invocará um dos métodos da tabela persistente. Nestes, se a operação não altera o estado da tabela, será apenas chamada a função correspondente do módulo `table`. Se a operação altera o estado da tabela, deverá ser feita a operação correspondente na tabela e, em caso de sucesso, deverá ser adicionada uma entrada ao ficheiro de *log* (`pmanager_log()`).

5. Entrega

A entrega do projeto 4 consiste em colocar todos os ficheiros do projeto, bem como o ficheiro README mencionado abaixo, num ficheiro com compressão no formato ZIP. Este ficheiro será depois entregue na página da disciplina, no moodle da FCUL.

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 devem explicar como executar o projeto e incluir outras informações que julguem necessárias (e.g., limitações na implementação);
- diretorias adicionais para armazenar os ficheiros *.c* e *.h* correspondentes a cada módulo;
- um ficheiro `Makefile` que permita a correta compilação de todos os ficheiros entregues. **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), o trabalho é considerado nulo**.
  Na página da cadeira 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 três linhas de comentários a dizer o número do grupo e o nome e número de seus elementos.

O prazo de entrega é domingo, dia 15/11/2015, até às 22:00hs.