Ciências ULisboa | Informática

2015/16

Sistemas Distribuídos

Projeto 5

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. No projeto 4 reestruturaram-se o cliente e o servidor segundo um mecanismo de comunicação do tipo RPC, alterou-se o servidor de forma a suportar múltiplos clientes simultaneamente, e iniciou-se a implementação de uma tabela persistente.

O projeto 5 tem 2 objetivos:

1. **Finalizar a implementação de uma tabela persistente em disco**. No projeto anterior, o servidor regista num ficheiro de *log*, as operações que alteram a tabela. Como sabemos, com o funcionamento contínuo do servidor, o ficheiro de log poderá crescer até se tornar demasiado grande para o sistema de armazenamento, e também fazer com que a recuperação seja lenta. Assim vamos complementar o funcionamento com a adição de um ficheiro de *checkpoint* que é criado sempre que o ficheiro de *log* atinge uma determinada dimensão.
2. **Aumentar a disponibilidade do sistema e melhorar a tolerância a faltas**. Um problema fundamental que ainda persiste é a indisponibilidade do sistema desde que o servidor falha até à sua recuperação. Para tratar este problema, o projeto 5 propõe a utilização de réplicas do servidor, utilizando um sistema de quóruns para controlo de consistência. A ideia fundamental é alterar o cliente para que este escreva e leia pares chave/valor num conjunto de servidores, em vez de o fazer num servidor apenas. Assim, o problema de programação a ser resolvido consiste em invocar operações em diversas tabelas remotas e retornar apenas as respostas de um subconjunto maioritário de servidores que responderam primeiro.

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. Finalização da tabela persistente

2.1. Alterações ao módulo *table*

A finalização da implementação da tabela persistente requer alterações ao módulo *table* realizado anteriormente. Devem acrescentar ao módulo, em *table-private.h*, as seguintes funções:

```
/* Retorna o número de alterações realizadas na tabela.
*/
int table_get_num_change_ops(struct table_t *table);

/* Devolve um array de entry_t* com cópias de todas as entries
 * da tabela, e um último elemento a NULL.
 */
struct entry_t **table_get_entries(struct table_t *table);

/* Liberta a memória alocada por table_get_entries().
 */
void table_free_entries(struct entry_t **entries);
```

2.2. Alterações ao módulo *persistence_manager*

O módulo *persistence_manager* passa agora a gerir três tipos de ficheiros, a saber:

- *.log*: ficheiro de *log* onde são armazenadas as operações executadas na tabela, que alteram o estado desta (já abordado no projeto 4);
- *.ckp*: ficheiro que contém um *checkpoint* da tabela, com todo o seu estado no momento em que o ficheiro foi criado;
- *.stt*: ficheiro similar ao anterior, porém de cariz temporário, usado apenas para criar um novo checkpoint e garantir que não estragamos o ficheiro de *checkpoint* antigo antes do novo estar completamente escrito.

Note que nenhum formato dos *logs* é exigido, porém, tal como no projeto anterior, é sugerido que se usem os arrays de carateres nos ficheiros de *checkpoint*. Ou seja, os ficheiros de checkpoint podem ser definidos através de uma sequência de mensagens de inserção de entradas na tabela.
As seguintes funções devem ser adicionadas ao módulo *persistence_manager*, em persistence_manager-private.h.

```
/* Cria um ficheiro filename+".stt" com o estado atual da tabela table.
 * Retorna o tamanho do ficheiro criado ou -1 em caso de erro.
 */
int pmanager_store_table(struct pmanager_t *pmanager,
                         struct table_t *table);

/* Limpa o conteúdo do ficheiro ".log" e copia o ficheiro ".stt" para ".ckp".
 * Retorna 0 se tudo correr bem ou -1 em caso de erro.
 */
int pmanager_rotate_log(struct pmanager_t *pmanager);
```

O nome base dos ficheiros *.stt* e *.ckp* deve ser o mesmo do ficheiro *.log*. É de notar que na versão concluída da tabela persistente, ao reiniciar após uma falha o servidor deve verificar que ficheiros existem e assim determinar como deverá ser feita a recuperação. O caso específico em que exista um ficheiro ".stt" é de particular interesse. Significa que houve uma falha antes de o ficheiro ".stt" ser copiado para ".ckp". A questão que se coloca então é a de saber se o fiocheiro ".stt" estará ou não completo. Mediante a conclusão deve-se escolher se a recuperação

é feita através do ficheiro ".stt", caso esteja completo, ou, caso contrário, através do *checkpoint* anterior e do log correspondente.

3. Replicação com quóruns

O algoritmo a concretizar baseia-se na implementação clássica de registos atómicos partilhados sobre um conjunto de servidores, numa rede de computadores.

3.1. Hipóteses

O sistema não requer nenhum pressuposto de tempos na rede nem a deteção de falhas dos servidores (quem falham apenas por paragem), e funciona desde que uma maioria de servidores esteja correta. Isto é, num sistema com *n* servidores, são toleradas até *t* faltas simultâneas por paragem, sendo $t \leq \lfloor n - 1/2 \rfloor$.

Note que este sistema de quóruns garante que as restrições de intersecção vistas nas aulas teóricas são respeitadas, já que quaisquer dois quóruns maioritários (de leitura ou escrita) se intersectam em pelo menos um servidor correto.

3.2. Timestamps

Juntamente com o nome do elemento de dados (chave na tabela) e o dado (valor), armazenamos também um *timestamp* que deve sempre ser composto por um contador e pelo *id* do processo cliente. Assim, a forma mais simples de um cliente com um id *i*, incrementar um *timestamp* com valor *ts* seria:

1.) Dividir *ts* por 1000 e fazer *cast* para *long* (para ignorar o valor do id anterior).
2.) Incrementar *ts* em 1 (incremento).
3.) Multiplicar *ts* por 1000 (para criar o espaço para o *id* do cliente).
4.) Somar *i* a *ts* (marcando-o com o *id* do cliente).

Outras concretizações mais eficientes são possíveis, mas esta é suficiente para criar *timestamps* únicos.

3.3. Operações

Escrita (put). Representada pela função *table_put(key, value)*, que deve ser implementada segundo o algoritmo que se segue:

1.) Gera um *timestamp ts* no formato apresentado, para a *key*.
2.) Escreve *key*, *ts* e *value*, num quórum de servidores.

Alteração (update). Representada pela função *table_update(key, value)*, que deve ser concretizada usando o seguinte algoritmo:

1.) Lê o *timestamp* associado a *key* de um quórum de servidores. Para isso, será usada uma nova operação, descrita mais abaixo.
2.) Escolhe o maior *timestamp ts* do quórum de servidores.
3.) Escreve *k*, *incremento(ts)* e *value* num quórum de servidores.

Leitura (get). Representada pela função *table_get(key)*, concretizada pelo seguinte algoritmo:

1.) Lê os dados e o *timestamp* associado a *key* num quórum de servidores. Para isso, deve ser feito um table_get aos servidores.
2.) Escolhe os dados associados ao maior *timestamp*.
3.) Se algum dos servidores retornou um *timestamp* menor que o escolhido (i.e., este servidor não tem a versão mais nova), escrevem-se de volta nesse servidor os dados escolhidos com a *timestamp* correspondente (esta fase é normalmente chamada *writeback*). Este procedimento tende a aumentar a consistência dos dados de um servidor que já teve falhas.

Outras operações. A operação *table_del(key)* nada mais é que um *table_update(key, NULL)*. Ou seja, a chave *key* fica com 0 bytes de dados associados. Desta forma precavemos a possibilidade de uma operação *table_del*, não ser propagada para servidores que poderão estar em falha. Resta ao grupo definir como concretizar as funções *table_size* e *table_get_keys*.

3.4. Modificações necessárias em módulos implementados anteriormente

Os módulos concretizados nos projetos anteriores devem ser modificados da seguinte forma:

Módulo message e data:

- Adicionar um campo *long timestamp* na *struct data_t*;
- Adicionar um novo campo na *union* de *message_t.content* para permitir enviar uma mensagem com apenas um *long timestamp*. O *c_type* desse campo será CT_TIMESTAMP = 60 (a definir em message-private.h);
- Modificar as funções *buffer_to_message* e *message_to_buffer* para tratar as modificações anteriores. Os *timestamps* devem ser serializados em 64 bits, e, no caso de *content* ser uma *data_t*, devem aparecer antes dos bytes de *data_t.data*. A conversão para formato de rede será feita com uma função disponibilizada na página de SD.

Módulo table e list.

- Devem ser capazes de guardar um valor com o campo *data_t.data = NULL* e *data_t.datasize = 0* (mas com *timestamp* válido).
- Não modificam os dados associados a uma chave se o *timestamp* da atualização for menor ou igual ao *timestamp* dos dados que estão associados à chave na tabela. Isto significa que não se aceitam *updates* para uma chave, que diminuam o *timestamp* desta.

Módulo network_client.

- O sistema deve tentar reestabelecer a ligação com o servidor não apenas uma vez após 5 segundos, mas a cada 5 segundos (para sempre).

Módulos table, persistent_table e client_stub. Criar funções para retornar apenas o *timestamp* do valor associado a uma chave.

Adicionar a *table-private.h*:

```
/* Função para obter o timestamp do valor associado a uma chave.
 * Em caso de erro devolve -1. Em caso de chave não encontrada devolve 0.
 */
long table_get_ts(struct table_t *table, char *key);
```

Adicionar a *persistent_table-private.h*:

```
/* Função para obter o timestamp do valor associado a uma chave.
 * Em caso de erro devolve -1. Em caso de chave não encontrada
 * devolve 0.
 */
long ptable_get_ts(struct ptable_t *ptable, char *key);
```

Adicionar a *client_stub-private.h*:

```
#define OC_RT_GETTS 60

/* Função para obter o timestamp do valor associado a essa chave.
 * Em caso de erro devolve -1. Em caso de chave não encontrada
 * devolve 0.
 */
long rtable_get_ts(struct rtable_t *rtable, char *key);
```

4. Cliente

A maior parte da concretização deste trabalho será feita no cliente. Um aspeto que terá de ser complementado, é o dos parâmetros definidos pela linha de comando. Devemos agora esperar pelo *id* do cliente (um número inteiro entre 1 e 999), e pelo endereço dos *n* servidores de tabela, doravante denominados individualmente por servidor *0, …, n-1*. O comando será agora:

`table-client <id-cliente> <servidor`$_0$`:porto`$_0$`> … <servidor`$_{n-1}$`:porto`$_{n-1}$`>`

O programa cliente deve ser capaz de criar uma *remote_table* para aceder a cada um destes servidores, separadamente e concorrentemente. Esse acesso concorrente a um conjunto de servidores é encapsulado no módulo *quorum_table*, descrito a seguir.

4.1. QRPC stub (quorum_table.c)

O módulo `quorum_table.c` define um *stub* para acesso à tabela persistente concretizada por um conjunto de servidores. A ideia fundamental deste módulo é concretizar os algoritmos definidos na secção 3.3 utilizando o módulo `quorum_access.c`, definido na próxima secção.

A interface a ser oferecida ao cliente é a seguinte:

```
#ifndef _QUORUM_TABLE_H
#define _QUORUM_TABLE_H

#include "data.h"

/* A definir pelo grupo em quorum_table-private.h */
struct qtable_t;

/* Função para estabelecer uma associação entre uma tabela qtable_t e
 * um array de n servidores.
 * addresses_ports é um array de strings e n é o tamanho deste array.
 * Retorna NULL caso não consiga criar o qtable_t.
 */
struct qtable_t *qtable_bind(const char **addresses_ports, int n);

/* Fecha a ligação com os servidores do sistema e liberta a memoria alocada
 * para qtable.
 * Retorna 0 se tudo correr bem e -1 em caso de erro.
 */
int qtable_disconnect(struct qtable_t *qtable);

/* Função para adicionar um elemento na tabela.
 * Note que o timestamp em value será atribuído internamente a esta função,
 * como definido no algoritmo de escrita.
 * Devolve 0 (ok) ou -1 (problemas).
 */
int qtable_put(struct qtable_t *qtable, char *key, struct data_t *value);

/* Função para atualizar o valor associado a uma chave.
 * Note que o timestamp em value será atribuído internamente a esta função,
 * como definido no algoritmo de update.
 * Devolve 0 (ok) ou -1 (problemas).
 */
int qtable_update(struct qtable_t *qtable, char *key,
                  struct data_t *value);

/* Função para obter um elemento da tabela.
 * Em caso de erro ou elemento não existente, devolve NULL.
 */
struct data_t *qtable_get(struct qtable_t *qtable, char *key);
```

```
/* Função para remover um elemento da tabela. É equivalente à execução
 * put(k, NULL) se a chave existir. Se a chave não existir, nada acontece.
 * Devolve 0 (ok), -1 (chave não encontrada).
 */
int qtable_del(struct qtable_t *qtable, char *key);

/* Devolve número (aproximado) de elementos da tabela ou -1 em caso de
 * erro.
 */
int qtable_size(struct qtable_t *qtable);

/* Devolve um array de char* com a cópia de todas as keys da tabela,
 * e um último elemento a NULL. Esta função não deve retornar as
 * chaves removidas, i.e., a NULL.
 */
char **qtable_get_keys(struct qtable_t *rtable);

/* Liberta a memória alocada por qtable_get_keys().
 */
void qtable_free_keys(char **keys);

#endif
```

4.2. Comunicação entre o cliente e o conjunto de servidores: quorum_access.c

O módulo `quorum_access.c` vai utilizar *n threads* para invocar operações em *n* servidores diferentes e retornar um conjunto de respostas. A interface é a seguinte:

```
#ifndef _QUORUM_ACCESS_H
#define _QUORUM_ACCESS_H

#include "client-stub-private.h"

/* Estrutura que agrega a informação acerca da operação a ser executada
 * pelas threads através da tabela remota implementada no módulo
 * client_stub.
 */
struct quorum_op_t {
      int id; /* id único da operação (o mesmo no pedido e na resposta) */
      int sender; /* emissor da operação, ou da resposta */
      int opcode; /* o mesmo usado em message_t.opcode */
      union content_u { /* o mesmo usado em message_t.content */
            struct entry_t *entry;
            long timestamp;
            char *key;
            char **keys;
            struct data_t *value;
            int result;
      } content;
};

/* Esta função deve criar as threads e as filas de comunicação para que o
 * cliente invoque operações a um conjunto de tabelas em servidores
 * remotos. Recebe como parâmetro um array rtable de tamanho n.
 * Retorna 0 (OK) ou -1 (erro).
 */
int init_quorum_access(struct rtable_t *rtable, int n);

/* Função que envia um pedido a um conjunto de servidores e devolve
 * um array com o número esperado de respostas.
 * O parâmetro request é uma representação do pedido, enquanto
 * expected_replies representa a quantidade de respostas esperadas antes
 * da função retornar.
 * Note que os campos id e sender em request serão preenchidos dentro da
```

```
 * função. O array retornado é um array com k posições (0 a k-1), sendo cada
 * posição correspondente a um apontador para a resposta de um servidor
 * pertencente ao quórum que foi contactado.
 * Caso não se consigam respostas do quórum mínimo, deve-se retornar NULL.
 */
struct quorum_op_t **quorum_access(struct quorum_op_t *request,
                                   int expected_replies);

/* Liberta a memoria, destroi as rtables usadas e destroi as threads.
 */
int destroy_quorum_access();

#endif
```

Na primeira chamada á função *quorum_access*, deve ser determinada qual a dimensão *k* do quórum maioritário que deve ser usado para garantir o maior número de falhas simultâneas de servidores. De seguida, ainda na primeira chamada, deverão ser contactados todos os *n* servidores e esperadas as respostas destes. As primeiras *k* respostas, de servidores que podem estar com menos carga e para os quais a latência de rede deverá ser pequena, vão definir qual o quórum de servidores que deverá ser contactado em operações futuras. Assim, a comunicação necessária para realizar uma operação será reduzida na proporção de *n-k* servidores. Se não se conseguirem pelo menos k servidores deverá ser retornado NULL. Nesta situação o procedimento deverá ser repetido na próxima chamada à função. O procedimento de escolha do quórum de k servidores poderá ser implementado com recurso, por exemplo, à operação *size* (que não altera o estado da tabela), e concretizado numa função específica para esse propósito.

Se em qualquer operação após a primeira, um número *m* dos *k* servidores estiver em falha deve-se proceder da seguinte forma:

1.) Um a um, envia-se o pedido aos *n-k* servidores que não faziam parte do quórum atual, até se obterem as *m* respostas em falta. Se não se conseguiram obter as respostas necessárias, deve-se retornar NULL.
2.) Tendo-se conseguido as respostas em falta, a função devolve o array de respostas.
3.) Na próxima chamada à função `quórum_access`, antes de realizar a operação pretendida, deve-se executar o procedimento descrito acima para escolha do quórum de *k* servidores com resposta mais rápida.

4.3. Funcionamento do cliente ao aceder ao quórum de servidores

Nesta secção é apresentada uma breve descrição de como um *update* deve funcionar no sistema. As demais operações devem ser concretizadas de forma similar. Omite-se a operação relativa à escolha inicial do quórum e ao caso de falhas de servidores do quórum.

Primeiramente apresentamos o pseudo-código do que deve ser executado do lado do cliente. Alguns dos passos executados neste pseudo-código estão representados na figura 1, que aparece mais adiante.

```
quorum_update(key, value) {
      criar quorum_op_t request com {OC_RT_GETTS, key};
      respostas = quorum_access(request, ceil((n+1)/2));
      ts = incremento(maior  timestamp recebido entre as respostas);
      alterar request para {OC_UPDATE, key, value com timestamp = ts};
      respostas = quorum_access(request, ceil((n+1)/2));
      se respostas recebidas são todas OK, retorne 0, caso contrário retorne -1.
}
```

```
quorum_access(op, k) {
        op.id = id único (pode ser gerado usando um contador e o id do cliente);
        for (i=0; i<k; i++){
                servidor = quórum[i]
                put op na fila queue[servidor] /* passo 1 na figura */
        respostas = espaço para n apontadores para quorum_op_t iniciados a NULL;
        respostas_recebidas = 0;
        for(respostas_recebidas = 0; respostas_recebidas < k; ) {

                /* passo 5 na figura */
                resposta = consome elemento da fila reply_queue
                if(resposta.id == id){ /* é uma resposta a um pedido enviado */
                        respostas[resposta.sender] = resposta;
                        respostas_recebidas++;
                }
        }
        return respostas;
}

funcao a ser executada pela thread i {
        while(!desligar) {
                obtem op da fila queue[i] /* passo 2 na figura */

                /* passo 3 na figura */
                executa a operação definida em op em rtable[i] e obtem resposta

                criar quorum_op_t resposta com {op.id, i, op.opcode+1,
                                                conteúdo da resposta}
                coloca resposta na fila reply_queue /* passo 4 na figura */
        }
}
```

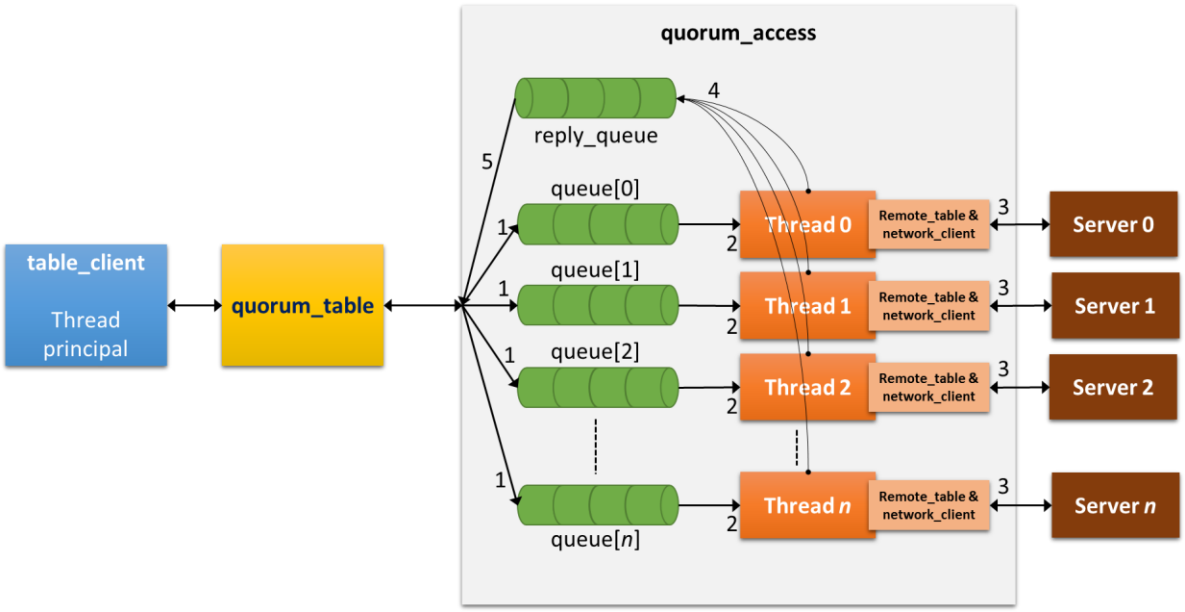

Figura 1. Arquitetura do sistema de replicação com quóruns.

É de notar que os algoritmos apresentados apenas ilustram a lógica geral da solução e omitem alguns detalhes conforme referido acima. Cabe aos alunos traduzir essa lógica, complementar os algoritmos, e traduzi-los para código C (ou inventar outro algoritmo).

Na figura pode-se observar que cada uma das filas usadas para comunicação entre as *threads* do cliente, são concretizações do problema produtor-consumidor. As filas serão implementadas através de variáveis em memória.

5. Pontos em aberto.

O projeto definido nas secções acima requer que os alunos trabalhem em alguns pontos em aberto, que valem aproximadamente 20% da nota do projeto 5 (i.e., quem os ignorar completamente ainda pode ter 16 valores neste projeto). Os pontos em aberto são:

- Como concretizar a função *size*? Em que condições a concretização proposta funciona?
- Como concretizar a função *getkeys*? Em que condições a concretização proposta funciona?
- Como limpar as chaves a NULL dos servidores (implementar *garbage collection*)? Em que condições a concretização proposta funciona?
- Como responder ao cliente quando não se conseguiu resposta do quórum de *k* servidores? Em que condições a implementação funciona e que impacto tem na consistência da tabela de quorum?

Tendo em conta que estes quatro pontos não podem ser concretizados de forma perfeita sem que se assumam algumas hipóteses extra no sistema, o grupo deve definir precisamente as condições necessárias para que a sua concretização funcione, e descrever com detalhe as soluções encontradas. Estas descrições devem ser incluídas no ficheiro REAME do projecto 5 (ver secção 7).

6. Sugestão de implementação

Sugere-se a seguinte sequência de implementação das tarefas relativas ao projeto 5:

1.) Finalização da tabela persistente, implementando as alterações apresentadas na secção 2 e alterando o procedimento de recuperação para que este atue de acordo com os ficheiros que forem encontrados na inicialização após falha.
2.) Implementar as alterações sugeridas na secção 3.4, bem como a função para incrementar uma timestamp, especificada na secção 3.2. Estas alterações são necessárias a implementação com êxito dos novos módulos.
3.) Alterar o processamento dos parâmetros de entrada do cliente, de acordo com a descrição apresentada no início da secção 4.
4.) Implementar o módulo `quorum_table` descrito na secção 4.1. A implementação pode ser testada parcialmente com um servidor, através do módulo `client_stub`.
5.) Implementar o módulo quorum_access, para acesso aos *n* servidores. Deverão implementar as funções pela ordem que são apresentadas no ficheiro ".h". As operações na tabela que devem implementar em primeiro lugar são as que se apresentam no início da secção 4.1.
6.) Implementar as operações e resolver as questões em aberto, que se apresentam na secção 5.

7. Entrega

A entrega do projeto 5 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 6/12/2015, até às 22:00h.