Ciências ULisboa | Informática

2022/23

Sistemas Distribuídos

Projeto 3

1. Descrição Geral

A componente teórico-prática da disciplina de sistemas distribuídos consiste no desenvolvimento de quatro projetos, utilizando a linguagem de programação C [3], 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, a estrutura de dados utilizada para armazenar esta informação é uma árvore de pesquisa binária [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. No Projeto 2 implementaram-se as funções necessárias para serializar e de-serializar estruturas complexas usando Protocol Buffer, um servidor concretizando a *árvore binária*, e um cliente com uma interface de gestão do conteúdo da *árvore binária*.

No Projeto 3 iremos criar um sistema concorrente que aceita pedidos de múltiplos clientes em simultâneo através do uso de multiplexagem de I/O [4] e que separa o tratamento de I/O do processamento de dados através do uso de Threads. Mais concretamente, vai ser preciso:

- Adaptar o servidor de modo a que este suporte pedidos de múltiplos clientes ligados em simultâneo, o que será feito através de multiplexagem de I/O (usando a chamada ao sistema `poll()`);
- Adaptar o servidor para dar respostas assíncronas aos pedidos de escrita dos clientes, ou seja, em vez de executar imediatamente pedidos de escrita, o servidor devolve aos clientes um identificador da operação e passa a guardar os pedidos numa fila temporária para serem executados por um conjunto de *threads*. Isto envolve:
    - Guardar no servidor um **contador de operações de escrita *last_assigned***, que é usado para atribuir um número de sequência único a cada operação de escrita e permite saber qual foi a última escrita recebida. Sempre que um cliente envia uma nova operação de escrita (*put* ou *delete*), o servidor responde com o valor atual de *last_assigned*, incrementando o contador logo de seguida. Atenção que pedidos de leitura (*get*, *size, height, getkeys* e *getvalues*) continuam a ser executados de forma síncrona pela *thread* principal.
    - Guardar no servidor uma **estrutura *op_proc*** que possui: a) um inteiro *max_proc* que regista o maior identificador das operações de escrita já concluídas; b) um array de inteiros *in_progress* que regista o identificador das operações de escrita que estão a ser atendidas por um conjunto de *threads* dedicadas às escritas. Ou seja, enquanto uma *thread* está a executar uma operação de escrita, o identificador da operação permanece armazenado em *in_progress*.
    - Implementar no cliente e no servidor uma **operação *verify***, que pode ser invocada pelos clientes, que leva como argumento o identificador de uma operação de escrita, e que tem como objetivo verificar se esta já foi executada pelo servidor (através da informação contida na estrutura *op_proc*).

- Implementar no servidor uma **Fila de Pedidos (Produtor/Consumidor)** onde os pedidos de escrita são guardados até serem executados;
- Adaptar o servidor para ter **dois tipos de threads**:
  - ***Main thread***: correspondente à *thread* principal do programa, que fica responsável por fazer a multiplexagem de novas ligações e de pedidos de clientes, por responder a pedidos de leitura e *verify*, e por inserir pedidos de escrita na Fila de Pedidos; e
  - ***Threads* secundárias**: deverá haver um conjunto de **N** *threads* secundárias, lançadas pela *main thread*, que retiram as operações a executar da Fila de Pedidos e as executam. O valor de N deverá ser indicado como argumento no comando de execução do servidor, que passa a ser executado da seguinte forma:
    `tree-server <port> <N>`
    Note que o array *in_progress*, que regista o número das operações em execução pelas *threads* secundárias, deve ter N posições.
- Garantir a **sincronização das *threads*** no acesso à árvore, à Fila de Pedidos, e à estrutura *op_proc,* através do uso de mecanismos de gestão da concorrência.

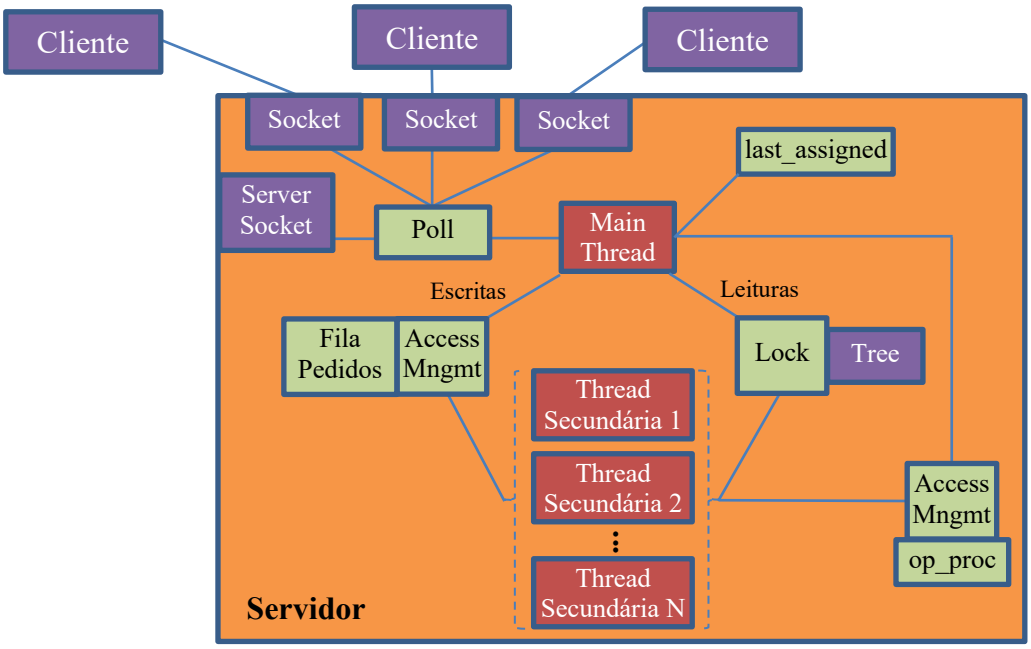
Como nos projetos anteriores, espera-se uma grande fiabilidade por parte do servidor e cliente, portanto não podem existir condições de erro não verificadas ou gestão de memória ineficiente a fim de evitar que estes sofram um *crash*.

2. Descrição Detalhada

O objetivo específico do projeto 3 é desenvolver uma aplicação do tipo cliente-servidor capaz de suportar múltiplos clientes de forma assíncrona. Para tal, para além de aproveitarem o código desenvolvido nos projetos 1 e 2, os alunos devem fazer uso de novas técnicas ensinadas nas aulas, incluindo técnicas para:

- multiplexagem de pedidos de clientes, utilizando a função `poll()`;
- execução paralela dos pedidos de escrita através de múltiplas *threads*, utilizando as primitivas para criação de *threads* fornecida pela biblioteca de *threads* `pthreads`;
- gestão da concorrência, de modo a garantir a sincronização de *threads* no acesso a variáveis e estruturas de dados na memória partilhada, utilizando *mutexes* e *condições*.

Devem também implementar a Fila de Pedidos, que será uma fila do tipo FIFO a ser usada pela *main thread* para registar os pedidos e pelas *threads* secundárias para os obter.



A figura anterior ilustra o novo modelo de comunicações que será usado no projeto 3. Atenção que este modelo abstrai os detalhes de comunicação implementados no projeto 2, ou seja, não mostra módulos *stub*, *skeleton* e *network* para simplificar a apresentação. A cor roxa representa o que já foi feito dos projetos 1 e 2, e as cores verde e vermelho representam o que é preciso fazer neste projeto 3.

2.1. Servidor com Multiplexagem I/O

Nesta secção apresenta-se uma breve descrição do novo código a ser desenvolvido na função `network_main_loop(int listening_socket)` do servidor, mais especificamente no *network_server*. Assume-se que o `listening_socket` recebido como argumento já foi preparado para receção de pedidos de ligação num determinado porto, tal como especificado no projeto 2.

```
/*
 * Esboço do algoritmo a ser implementado na função network_main_loop
 */
adiciona listening_socket a desc_set.       /* desc_set corresponde a um conjunto de file descriptors */

while (poll(desc_set) >= 0) {                          /* Espera por dados nos sockets abertos */
      if (listening_socket tem dados para ler) {       /* Verifica se tem novo pedido de conexão */
            connsockfd = accept(listening_socket);
            adiciona connsockfd a desc_set
      }
      for (all socket s em desc_set, excluindo listening_socket) {      /* Verifica restantes sockets */
            if (s tem dados para ler) {
                  message = network_receive(s);
                  if (message é NULL) {            /* Sinal de que a conexão foi fechada pelo cliente */
                        close(s);
                        remove s de desc_set
                  } else {
                        invoke(message);                    /* Executa pedido contido em message */
                        network_send(message);              /* Envia resposta contida em message */
                  }
            }
            if (s com erro ou POLLHUP) {
                  close(s);
                  remove s de desc_set
            }
      }
}
```

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

2.2. Pedidos de Escrita Assíncronos

Apesar do servidor fazer multiplexagem de pedidos dos clientes, ou seja, de conseguir atender pedidos de vários clientes, o tempo de espera por uma resposta a um pedido pode ser grande se este for complexo e se a computação necessária no lado do servidor for exigente. Neste trabalho vamos imaginar que os pedidos de escrita podem ser demorados (por exemplo, por requererem escritas síncronas no disco) e portanto, para evitar que o cliente fique à espera de uma resposta, vamos considerar que são executados de forma assíncrona. Ou seja, o servidor envia uma resposta ao cliente que apenas indica que o pedido foi recebido e qual o seu número de sequência, sendo o pedido guardado para ser executado mais tarde, logo que possível. Desta

forma o servidor pode atender rapidamente novos pedidos e não deixar os clientes à espera. No caso dos pedidos de leitura vamos considerar que são computacionalmente menos complexos e podem ser executados rapidamente, sendo por isso atendidos de forma síncrona, ou seja, o cliente fica à espera enquanto o servidor os executa, recebendo a resposta ao seu pedido.

2.2.1 Resposta a pedidos assíncronos

No caso dos pedidos de escrita, a serem executados de forma assíncrona, o servidor, mais concretamente o *tree_skel*, passa a guardar internamente o contador *last_assigned* e a estrutura *op_proc*. O contador *last_assigned* é um número inteiro inicializado com o valor 1 (um). Sempre que é recebido um novo pedido de escrita, o servidor responde com o valor atual de *last_assigned* e de seguida incrementa-o por uma unidade. Já a estrutura *op_proc* tem o inteiro *max_proc* e o array de inteiros *in_progress* inicializados a 0 (zero). Enquanto uma *thread* secundária está a executar uma operação de escrita, o identificador da respetiva operação permanece armazenado em *in_progress*. Quando o processamento da operação é concluído, se o identificador da operação for maior que *max_proc*, então o referido identificador é guardado em *max_proc*.

Adicionalmente, os alunos devem implementar um novo método *verify* que leva como argumento o número (de sequência) identificador de uma operação de escrita e verifica se esta foi executada.

Segue uma apresentação do novo formato das mensagens de resposta a implementar para pedidos de escrita, assim como do método *verify*:

COMANDO UTILIZADOR	**MENSAGEM DE PEDIDO**	**MENSAGEM DE RESPOSTA**
del <key>	OP_DEL CT_KEY <key>	OP_DEL+1 CT_RESULT <op_n> OP_ERROR CT_NONE <none>
put <key> <data>	OP_PUT CT_ENTRY <entry>	OP_PUT+1 CT_RESULT <op_n> OP_ERROR CT_NONE <none>
verify <op_n>	OP_VERIFY CT_RESULT <op_n>	OP_VERIFY+1 CT_RESULT <result> OP_ERROR CT_NONE <none>

Tabela 1: Novo pedido *verify* e novo formato de resposta para pedidos de escrita

Assim como o novo *opcode*:

```
/* Define os possíveis opcodes da mensagem */
...
OP_VERIFY         80
...
```

Não esquecer de adicionar o método *verify* ao *client_stub.c/h*:

```
#ifndef _CLIENT_STUB_H
#define _CLIENT_STUB_H

...

/* Verifica se a operação identificada por op_n foi executada.
 */
int rtree_verify(struct rtree_t *rtree, int op_n);

#endif
```

E ao *tree_skel.c/h*:

```
#ifndef _TREE_SKEL_H
#define _TREE_SKEL_H

...

/* Verifica se a operação identificada por op_n foi executada.
 */
int verify(int op_n);

...
```

2.2.2 Fila de Pedidos

A fila de pedidos permitirá à *thread* principal inserir novos pedidos de escrita (*put* e *del*) recebidos dos clientes, que as *threads* secundárias terão de remover para o atenderem. Assim, será necessário definir uma estrutura *request*, que guarda a informação necessária para executar um pedido de escrita, bem como um apontador para o próximo pedido a executar. Apresenta-se de seguida parte da implementação da estrutura *request*, que deve ser completada pelos alunos.

```
struct request_t {
        int op_n; //o número da operação
        int op; //a operação a executar. op=0 se for um delete, op=1 se for um put
        char* key; //a chave a remover ou adicionar
        char* data; // os dados a adicionar em caso de put, ou NULL em caso de delete
        //adicionar campo(s) necessário(s) para implementar fila do tipo produtor/consumidor
}
```

O *tree_skel* deve guardar a cabeça da fila (*request_t *queue_head*). Adicionalmente, devem usar mecanismos de gestão da concorrência para produzir/consumir concorrentemente desta fila sem *race conditions*, como explicado na próxima secção.

2.2.3 Gestão da Concorrência

Para processar os pedidos guardados na fila de pedidos, o servidor (nomeadamente, o *tree_skel* na função `tree_skel_init`) deve lançar **N** novas *threads* secundárias, através da API de threads do UNIX (pthreads).

A nova assinatura da função `tree_skel_init` é apresentada abaixo. Estas *threads* devem executar uma nova função `process_request` que irá tentar consumir da fila de pedidos. Indica-se também abaixo a assinatura da função `process_request`. Atenção que `params` pode potencialmente ser NULL:

```
#ifndef _TREE_SKEL_H
#define _TREE_SKEL_H

...

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

...

/* Função da thread secundária que vai processar pedidos de escrita.
*/
void * process_request (void *params);

...
```

A *main thread* servidor, depois de lançar as **N** *threads* secundárias, continua a fazer multiplexagem dos pedidos dos clientes, a responder a pedidos de leitura e *verify* dos clientes, e a inserir pedidos de escrita na fila de pedidos (isto é, vai produzir para a fila).

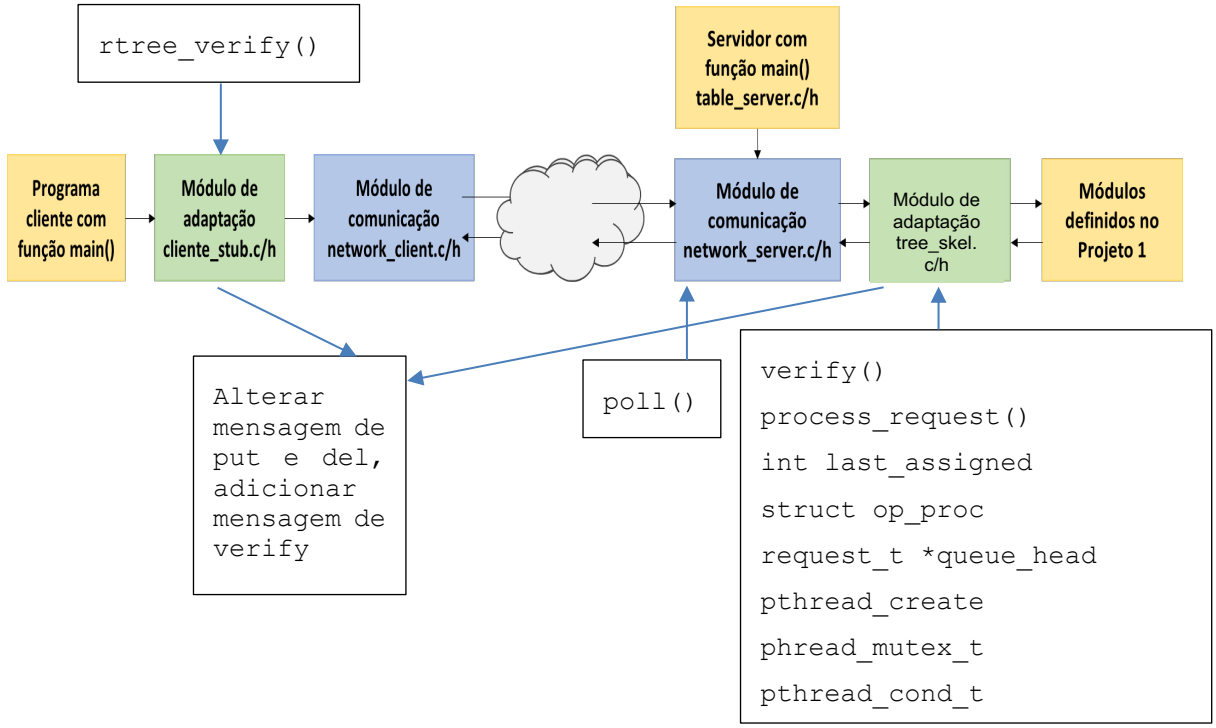
Isto significa que há três estruturas que poderão ser acedidas concorrentemente pela *main thread* e pelas *threads* secundárias: a Fila de Pedidos; a árvore do Projeto 1; e a estrutura *op_proc*. Assim, para garantir que o acesso a estas estruturas é feito de forma ordenada, que os resultados das operações sobre as mesmas são corretos, e que não existe o risco de as estruturas ficarem corrompidas, torna-se necessário utilizar mecanismos de gestão da concorrência a nível do *tree_skel.*

Existem dois mecanismos que podem ser usados para concretizar a gestão da concorrência: *locks* (`pthread_mutex_t`) e variáveis condicionais (`pthread_cond_t`). A forma como são utilizados é da responsabilidade dos alunos. Contudo, os alunos devem ter em conta o seguinte:

1. A Fila de Pedidos será acedida pela *main thread* e pelas *threads* secundárias. O mecanismo de gestão da concorrência deve permitir que a *main thread* produza para a fila e as *threads* secundárias consumam da fila sem comprometer a correção da mesma.
2. Se não existir nenhum pedido na Fila de Pedidos, as *threads* secundárias deverão ficar bloqueadas até que seja inserido um novo pedido pela *main thread*. Para tal, devem ser usadas variáveis condicionais para bloquear as *threads*, e as funções de sinalização (*pthread_cond_signal* ou *pthread_cond_broadcast*) para as acordar.
3. A estrutura *op_proc* será escrita pelas *threads* secundárias e lida pela *main thread*. Deve-se permitir a execução concorrente tanto quanto possível, evitando bloquear operações que possam ser executadas sem comprometer a correção da estrutura.
4. Relativamente à árvore, deve-se usar o mecanismo de *lock* para prevenir que operações concorrentes de escrita (realizadas pela *main thread*) e leitura (realizadas pela *thread* secundária) comprometam a correção da árvore.

3. Sumário de Alterações

Esta secção apresenta um sumário de onde deve ser feita cada alteração, dada a estrutura de ficheiros definida no projeto 2.



4. Makefile

Os alunos deverão manter o `Makefile` usado no Projeto 2, atualizando-o para compilar o novo código, se necessário.

5. Entrega

A entrega do projeto 3 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-projeto3.zip`** (XX é o número do grupo).
2. Submeter o ficheiro **`grupoXX-projeto3.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 Seção 4. 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 18/11/2020 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.

6. 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] W. Richard Stevens, Bill Fenner. Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition), 2003.