Ciências ULisboa | Informática

2020/21

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 [4], sendo que a realização de cada um deles é necessária para a realização do projeto seguinte. Por essa razão, **é muito importante que consigam ir cumprindo os objetivos de cada projeto, de forma a não hipotecar os projetos seguintes**.

O objetivo geral do projeto será concretizar um serviço de armazenamento de pares chave-valor (nos moldes da interface *java.util.Map* da API Java) similar ao utilizado pela *Amazon* para dar suporte aos seus serviços Web [1]. Neste sentido, a estruturas 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 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 thread. Isto envolve:
    - Guardar no servidor dois contadores de operações de escrita, *last_assigned* e *op_count*. *last_assigned* simboliza a última operação recebida enquanto *op_count* simboliza a última operação executada.
        - Sempre que um cliente envia uma nova operação de escrita (*put* ou *delete*), o servidor responde com o valor atual de *last_assigned* e de seguida incrementa-o por uma unidade. Atenção que pedidos de leitura (*get*, *size* e *getkeys*) ficam inalterados.
        - Quando um thread executa uma tarefa, o valor de *op_count* é incrementado por uma unidade.
    - Implementar uma operação *verify*, que leva como argumento o identificador de uma operação e verifica se esta já foi executada.
    - Implementar uma **Fila de Tarefas (Produtor/Consumidor)** onde os pedidos de escrita são guardados até serem executados;
    - Adaptar o servidor para ter dois threads:

- o thread principal do programa, que fica responsável por fazer multiplexagem de novas ligações e pedidos de clientes, por responder a pedidos de leitura, e por inserir pedidos de escrita na Fila de Tarefas; e
- um thread secundário, lançado pela thread principal, que retira as operações a executar da Fila de Tarefas e executa-as.

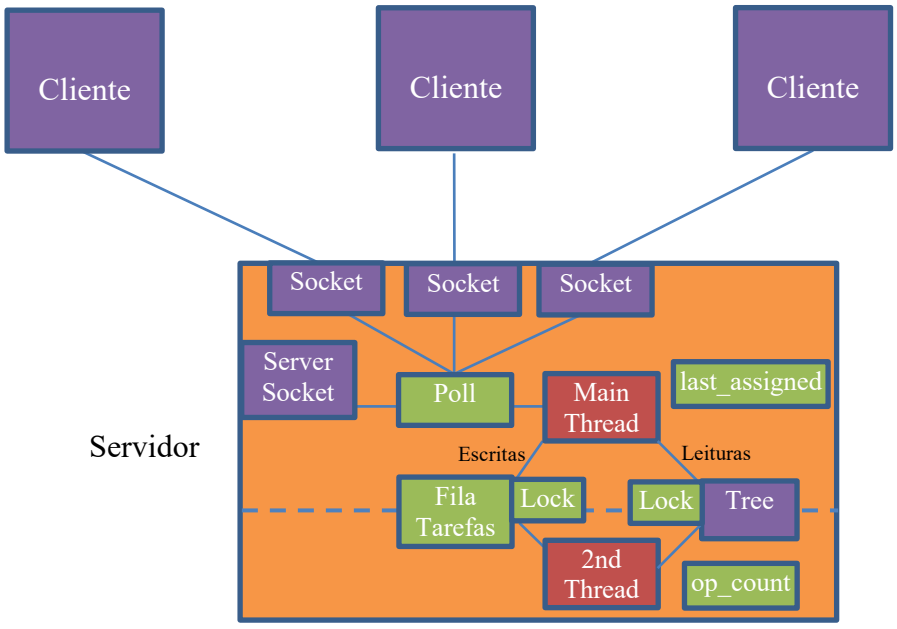
- Garantir a sincronização de threads no acesso à árvore e à Fila de Tarefas através do uso de Locks e Variáveis Condicionais.

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 a função `poll()` para multiplexagem de pedidos de clientes, a biblioteca de threads `pthreads` para separar o atendimento de pedidos de clientes da execução dos mesmos, e técnicas de *Mutual Exclusion* como *locks* para garantir a sincronização de threads no acesso a variáveis de memória partilhada. Devem também implementar uma estrutura de dados simples do tipo fila FIFO de forma a partilhar dados entre as várias threads.

A figura abaixo ilustra o novo modelo de comunicações que será usado no projecto 3. Atenção que este modelo abstrai os detalhes de comunicação implementados no projecto 2, ou seja, não mostra módulos stub, skeleton e network apenas por simplificação na apresentação. A cor roxa representa o que já foi feito dos projectos 1 e 2, verde e vermelho representa o que é preciso fazer neste projecto 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

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 inventar outro algoritmo).

2.2. Servidor com Pedidos de Escrita Assíncronos

Apesar do servidor fazer multiplexagem de pedidos dos clientes, ainda assim estes podem ficar pendurados à espera da resposta de um pedido se a computação necessária no lado do servidor for computacionalmente exigente. De forma a os pedidos de escrita passarem a ser assíncronos (pedidos de leitura continuam a ser síncronos por simplificação e uma vez que precisam de uma resposta), é necessário separar o tratamento de I/O do processamento de dados.

2.2.1 Resposta a pedidos assíncronos

O servidor, mais concretamente o *tree_skel*, passa a guardar internamente dois contadores de operações de escrita, *last_assigned* e *op_count*. Os contadores são números inteiros inicializados a zero. Sempre que é recebido um novo pedido de escrita, o servidor responde com o estado atual de *last_assigned* e de seguida incrementa-o por uma unidade. Sempre que uma thread executa uma operação, o valor de *op_count* é incrementado por um. Estas variáveis ajudam na separação da lógica de processamento de pedidos da lógica de processamento de dados (que pode ser mais lenta).

Adicionalmente, os alunos devem implementar um novo método *verify* que leva como argumento o número de uma operação 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         60
...
```

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 Servidor com Fila de Tarefas

A fila de tarefas permitirá que várias threads comuniquem entre si, guardando temporariamente as operações de escrita a serem executadas. Para tal é necessário definir uma struct *task*, que guarda a informação necessária para executar um pedido de escrita e um apontador para a próxima tarefa a executar. Segue parte da implementação da struct *task*, que deve ser completada pelos alunos.

```
struct task_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 (*task_t *queue_head*). Adicionalmente, devem usar um lock e uma variável condicional para produzir/consumir concorrentemente desta fila sem *race conditions*, como explicado na próxima secção.

2.2.3 Servidor com Threads, Locks e Variáveis Condicionais

Para processar os pedidos guardados na fila de tarefas, o servidor (nomeadamente, o *tree_skel* na função *tree_skel_init*) deve lançar uma nova thread, através da API de threads do UNIX (pthreads). Este thread deve executar uma nova função *process_task* que irá tentar consumir da fila de tarefas. Segue assinatura da função *process_task*. Atenção que *params* pode potencialmente ser NULL:

```
#ifndef _TREE_SKEL_H
#define _TREE_SKEL_H

...

/* Função do thread secundário que vai processar pedidos de escrita.
*/
void * process_task (void *params);

...
```

A thread principal do servidor, depois de lançar a thread anterior, continua a fazer multiplexagem dos pedidos dos clientes, a responder a pedidos de leitura dos clientes, e a escrever pedidos de escrita na fila (isto é, vai produzir para a fila).

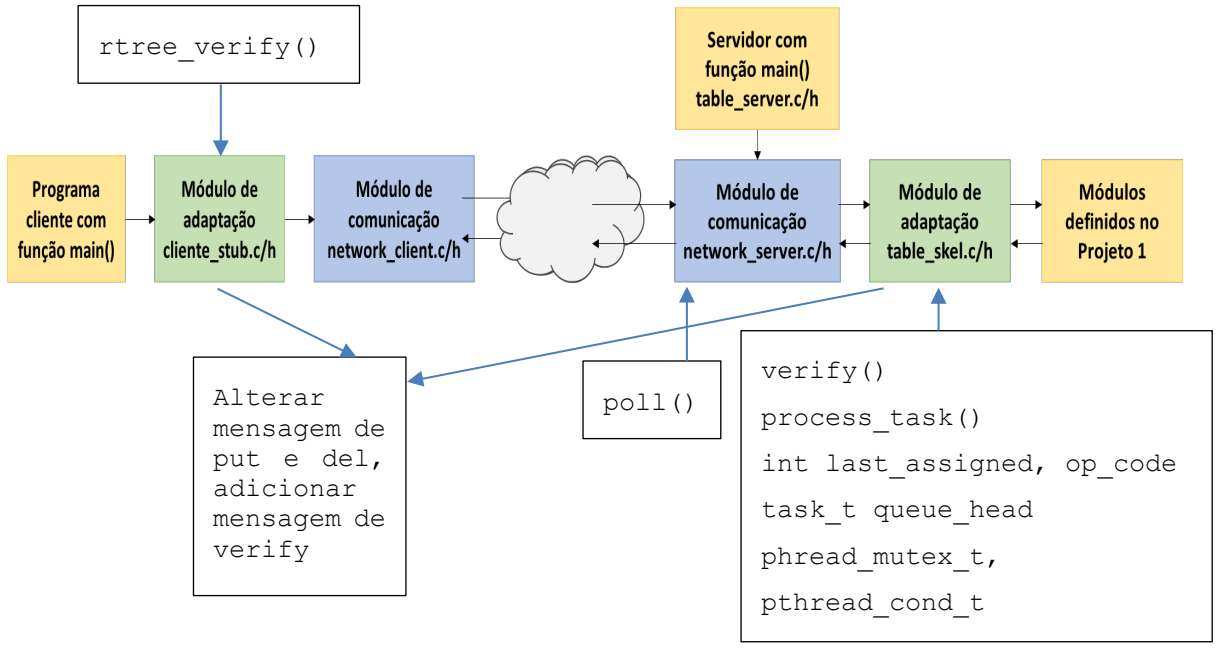
Isto significa que há duas estruturas que poderão ser modificadas concorrentemente pelas duas threads, a Fila de Tarefas e a árvore do Projecto 1. Como tal, é necessário usar um lock para a árvore, e um lock mais uma variável condicional para a fila, todos a nível do *tree_skel*:

pthread_mutex_t queue_lock, tree_lock;

pthread_cond_t queue_not_empty;

3. Sumário de Alterações

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



4. Makefile

Os alunos deverão manter o `Makefile` usado no Projecto 2, atualizando-o para compilar 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 Secção 7. Não devem ser incluídos no ficheiro ZIP os ficheiros objeto (.o) ou executáveis. Quaisquer outros ficheiros (por exemplo, de teste) também não deverão ser incluídos no ficheiro ZIP.

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

- **Se não for incluído um `Makefile`, se o mesmo não satisfizer os requisitos indicados, ou se houver erros de compilação (isto é, se não forem criados os ficheiros objeto e executáveis), o trabalho é considerado nulo**. Na página da disciplina, no Moodle, podem encontrar vídeos e documentos do utilitário `make` e dos ficheiros `Makefile` (cortesia da disciplina de Sistemas Operativos).
- Todos os ficheiros entregues devem começar com um cabeçalho com três ou quatro linhas de comentários a dizer o número do grupo e o nome e número dos seus elementos.
- Os programas são testados no ambiente dos laboratórios de aulas, pelo que se recomenda que os alunos testem os seus programas nesse ambiente.

O prazo de entrega é dia 22/11/2020 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.