Ciências ULisboa | Informática

Projeto 3

Bernardo Ferreira / Alan Oliveira /
António Casimiro

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

Fonte da imagem: http://www.dsn.jhu.edu

Projeto3

- **Objetivo:**

Criar um sistema concorrente que aceita e processa pedidos de múltiplos clientes em simultâneo através do uso de multiplexagem de I/O e *threads*.

Projeto3

- Mais concretamente, é preciso adaptar o servidor para:
  - Dar suporte a pedidos de múltiplos clientes ligados em simultâneo através de multiplexagem de I/O (`poll()`).
  - Dar respostas assíncronas aos pedidos de escrita dos clientes:
    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*.

Projeto3

- Isto envolve:
  - Guardar no servidor um contador de operações de escrita `last_assigned`. 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: pedidos de leitura (get, size, height, getkeys e getvalues) continuam a ser executados de forma síncrona pela thread principal.

  - Guardar no servidor uma `struct op_proc` tendo:
    - `int max_proc` → regista o maior identificador das operações de escrita já concluídas; e
    - *array* de inteiros `in_progress` → regista o identificador das operações de escrita que estão a ser atendidas por um conjunto de *threads* dedicadas às escritas.

Projeto3

- Isto envolve:

  - Implementar no cliente e no servidor uma operação *verify*, que leva como argumento o identificador de uma operação de escrita e verifica 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.

Projeto3

- Isto envolve:

  - Adaptar o servidor para ter dois tipos de *threads*:
    - » ***Main Thread***: 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

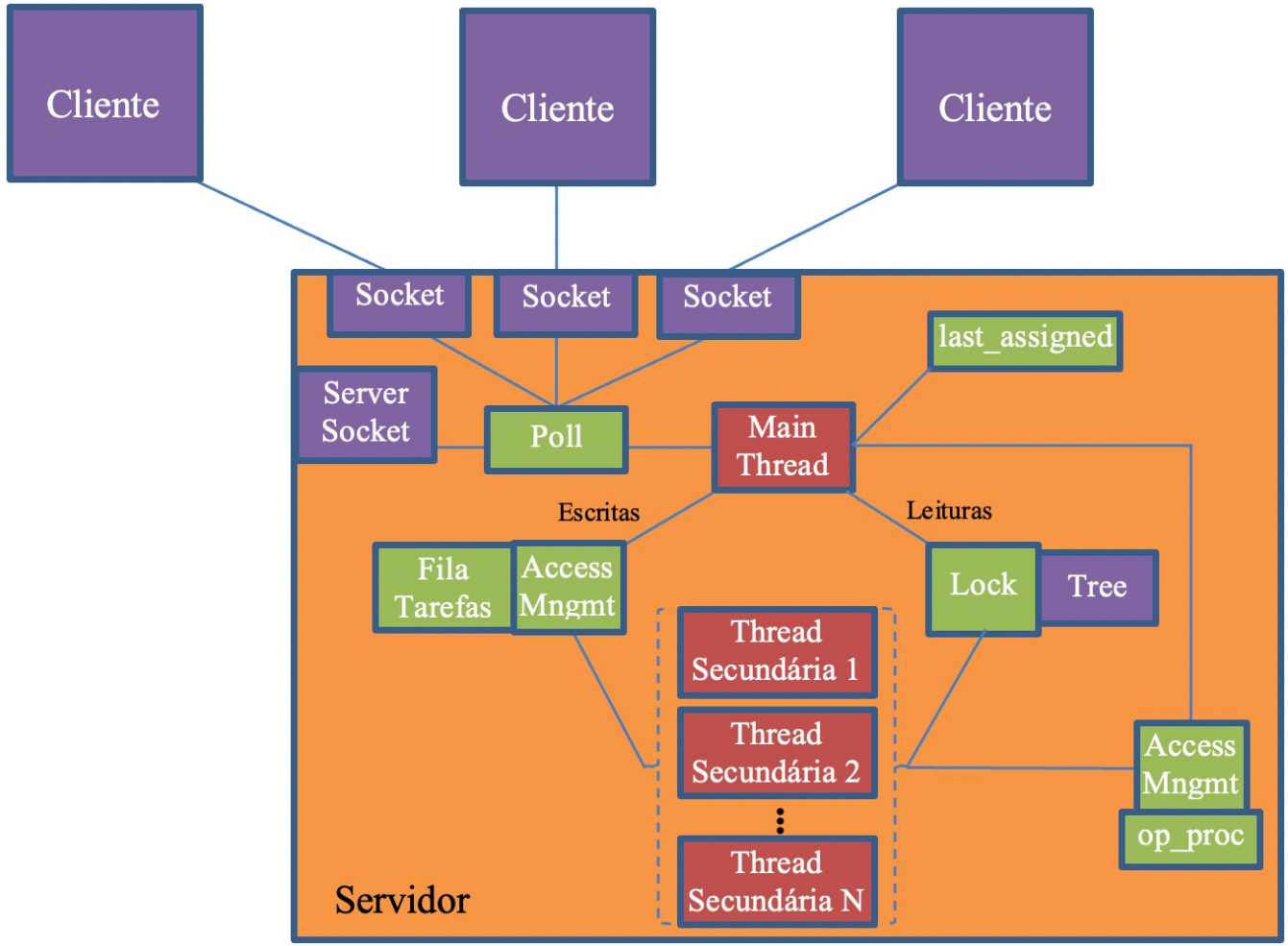
    - » Conjunto de N ***Threads* Secundárias** lançadas pela *main thread:* retiram as operações da Fila de Pedidos e as executam. N é indicado na partida do servidor:

      ```
      tree-server <port> <N>
      ```

      Nota: array `in_progress` deve ter N posições.

    - » Garantir a sincronização das *threads* no acesso à árvore, à Fila de Pedidos, e à estrutura op_proc (gestão da concorrência).

Projeto 3 - *overview*



Servidor com multiplexagem de I/O

```
/* 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
        }
    }
}
```


Resposta assíncrona a pedidos

- Pedido *verify* e novo formato de mensagens:

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>

- Novo *opcode*:

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

- Adicionar o método *stats* ao client_stub.c/h

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

- Incluir o tratamento da operação *verify* ao tree_skel.c/h

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


Servidor com Fila de Pedidos

- Definir uma struct request_t, que guarda a informação necessária para executar um pedido de escrita e um apontador para o próximo pedido a executar:

```
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
}
```

- Para processar os pedidos guardados na fila de pedidos, o servidor (o tree_skel, na função tree_skel_init) deve lançar N novas threads secundárias:

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

Gestão da concorrência

- Estruturas acedidas concorrentemente pela *main thread* e *threads secundárias*:

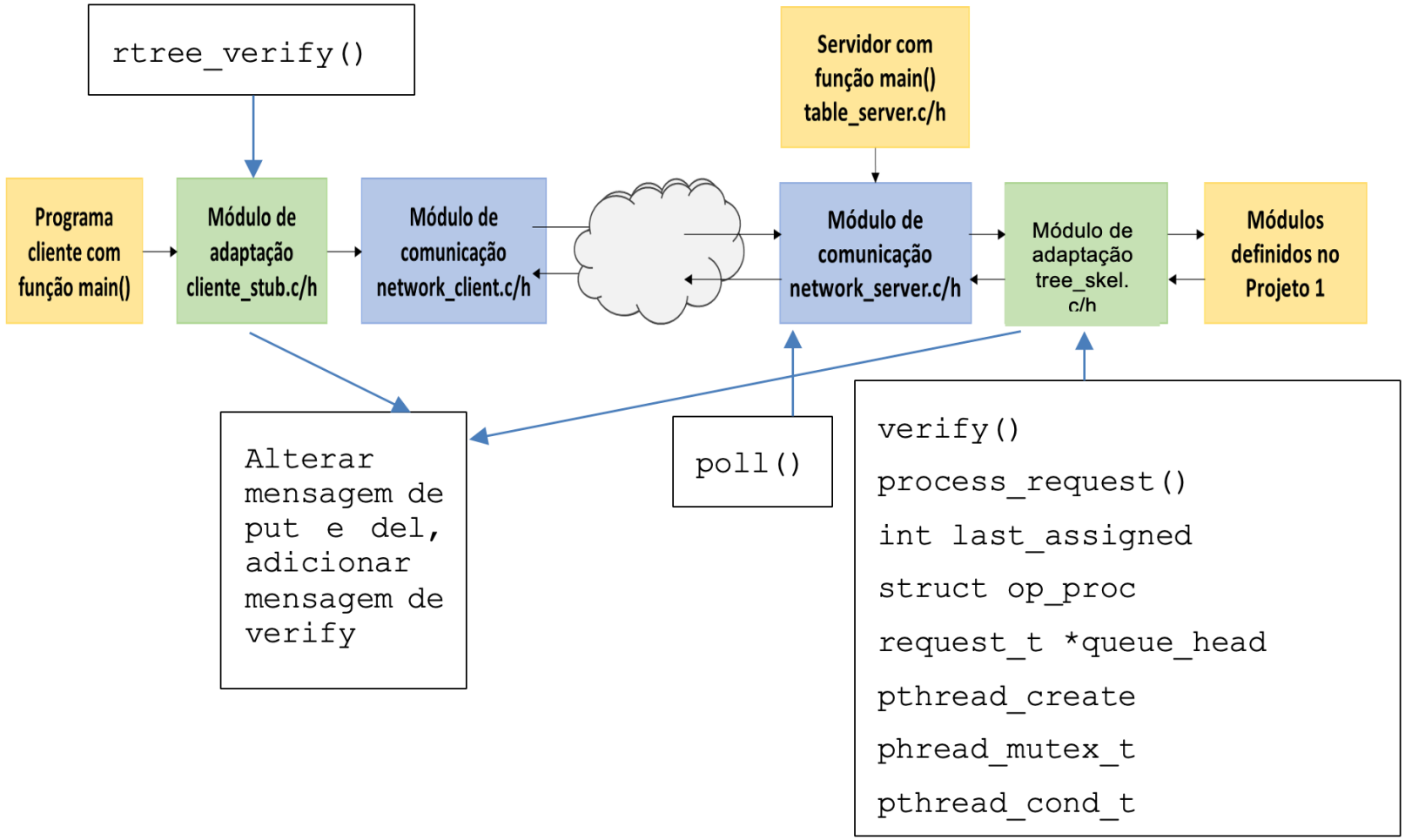
  1) **Fila de Pedidos**:
     - Acedida pela *main thread* e pelas *threads* secundárias. Deve-se permitir que a *main thread* produza para a fila e as *threads* secundárias consumam da fila sem comprometer a correção da mesma.
     - Se não existir nenhum pedido na Fila de Pedidos, as *threads* secundárias deverão ficar bloqueadas até que seja inserido um novo pedido pela *main thread*.

  2) **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.

  3) **Árvore**: deve-se usar 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.

- Mecanismo de gestão da concorrência:
  - *locks* (`pthread_mutex_t`)
  - variáveis condicionais (`pthread_cond_t`)

Projeto 3 - *overview*



Referências

- 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.
- Wikipedia . Binary Search Tree. https://en.wikipedia.org/wiki/Binary_search_tree
- B. W. Kernighan, D. M. Ritchie, C Programming Language, 2nd Ed, Prentice-Hall, 1988.
- W. Richard Stevens, Bill Fenner. Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition), 2003.