U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Sistemas Operativos

Home Banking

MIEIC - Turma 4 - Grupo 1

Nuno Cardoso up201706162@fe.up.pt
João Campos up20174982@fe.up.pt
Filipa Senra up201704077@fe.up.pt

Porto, 19 de maio de 2019.

Estruturas de dados usadas na comunicação

Para a troca de pedidos e respostas entre utilizadores e servidor utilizamos as estruturas de dados providenciadas no código de apoio ao projeto, no ficheiro *types.h*.

Os pedidos enviados pelos utilizadores são armazenados numa estrutura do tipo struct ***tlv_request_t****,* com três parâmetros:

1. ***enum op_type type***: atribui valores numéricos às 4 operações possíveis;
2. ***uint32_t length***: corresponde ao tamanho do pedido a enviar;
3. ***req_value_t value*****:** struct com os valores de ***header*** comuns a todas as operações (*pid*, *account_id*, *password* e *op_delay_ms*) e com uma *union* de *structs* ***req_create_account_t create*** (membros-dado: *account_id, balance* e *password*) e ***req_transfer_t transfer*** (membros-dado: *account_id* e *amount*), correspondentes aos valores específicos das operações de criação de conta e transferência, respetivamente.

As respostas enviadas pelo servidor são armazenadas numa estrutura do tipo *struct* ***tlv_reply_t*****,** com parâmetros em tudo semelhantes à estrutura anterior à excepção do parâmetro ***rep_value_t value***:

1. ***rep_header_t header*****;**
2. *union* de *structs* ***rep_balance_t balance*** (*uint32_t balance*), ***rep_transfer_t transfer*** (*uint32_t balance*) e ***rep_shutdown_t shutdown*** (*uint32_t active_offices*).

Comunicação entre o utilizador e o servidor

A comunicação entre utilizador e servidor é feita através de dois momentos:

Primeiramente, o utilizador envia o pedido (apenas uma chamada *write*, com o tamanho completo da mensagem):

➔ *“write(fdr, user_request, sizeof(op_type_t) + sizeof(uint32_t) + user_request->length;”*

De notar que *user_request* é um apontador para uma instância da estrutura ***tlv_request_t*****.**

De seguida, o servidor recebe o pedido (três chamadas read, de modo a receber toda a informação da mensagem no menor tamanho):

➔ *“read(fd_srv, &(user_request->type), sizeof(enum op_type));*
➔ *read(fd_srv, &(user_request->length), sizeof(uint32_t));*
➔ *read(fd_srv, &(user_request->value), user_request->length);”*

De notar que *user_request* é um apontador para uma instância da estrutura ***tlv_request_t*****.**

Posteriormente, o processo repete-se para o envio da resposta por parte do servidor e receção por parte do utilizador, com a estrutura ***tlv_reply_t*****.**

Mecanismos de Sincronização

No nosso programa, temos uma fila de pedidos e um array de contas. Para impedir a leitura e escrita errada nestas, fizémos uso de *mutexes*. Para além disso, utilizámos semáforos de acordo com o problema do produtor-consumidor.

O *mutex* ***q_mutex*** permite acesso à lista de pedidos e é bloqueada sempre que o servidor ou um box office tenta aceder a esta. A array de *mutexes* ***db_mutex*** tem um *mutex* correspondente a cada conta. Deste modo, quando se pretende ler ou alterar os atributos de uma dada conta é bloqueado o *mutex* de respetivo índice.

Inicialmente, sentimos algumas dificuldades no acesso a base de dados das contas por termos apenas um *mutex* a bloquear o acesso a todas as contas. Esta estratégia, impossibilitava a requisição fluida de vários pedidos em instantes próximos. Com esta nova abordagem ao problema, ao evitar bloquear toda a base de dados para o acesso a uma conta (ou duas, no caso de uma transferência), assegura-se uma melhor sincronização, multiplicidade e eficiência da plataforma.

Além de *mutexes* descritos supra, utilizamos dois semáforos, de valores iniciais: ***n_req*** com valor inicial 0 e ***b_off*** com valor inicial igual ao número de threads utilizados.

O semáforo ***n_req*** representa o número de requests que estão à espera de ser processados na queue. Por sua vez, o semáforo ***b_off*** permite saber quantos *box offices* estão livres num determinado momento.

Os valores destes semáforos tem um papel importante no encerramento do servidor como demonstraremos mais à frente.

Encerramento do servidor

O encerramento do servidor surgiu como um dos maiores desafio no nosso trabalho.

Inicialmente, utilizamos uma variável que sinalizava quando um pedido de *shutdown* era processado. Infelizmente, na esmagadora maioria dos casos, esta variável só era alterada quando o servidor estava à espera de receber um novo pedido e só no processamento de um pedido posterior ao do *shutdown* é que era processado o encerramento.

Posto isto, desenvolvemos uma estratégia que passa por 2 passos: quando recebe um pedido de *shutdown*, este sai do ciclo principal de processamento de pedidos e espera que todos os balcões virtuais estejam livres antes de prosseguir à sequência de encerramento.

Foi discutido uma possível implementação de um sinal que desbloqueasse o servidor e prosseguisse à sequência de encerramento. No entanto, optamos por uma estratégia mais simples de implementar que passa pelo server conferir que se trata de um pedido válido de encerramento e prosseguir aos passos descritos supra.

A sequência de encerramento consiste em primeiramente fechar o FIFO para apenas leitura. Seguidamente esperamos que todos os pedidos na queue tenham sido processados. Finalmente, fechamos todos os *box offices* e FIFO's e destruímos todos os semáforos e *mutexes*.