Projeto DSID - 2019/01

Este projeto é resultado parcial da disciplina de Desenvolvimento de Sistemas de Informação Distribuídos da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo (EACH-USP).

O projeto foi co-autorado por:

- Danilo Dainez da Rocha - 9875458
- Lucas Leal Caparelli - 9844957

O objetivo do projeto é a criação de um sistema distribuído de arquitetura cliente-servidor que faça uso de uma implementação de *threads* POSIX (IEEE Std 1003.1c, 1995).

Para tal escolheu-se usar C/C++ importando a biblioteca pthread.h, nativa da linguagem. Escolheu-se também trabalhar entre as camadas de rede de transporte e aplicação, fazendo uso de uma biblioteca para encapsulamento das operações oferecidas pela API de *sockets* nativa do C.

Objetivo

Criação de uma aplicação que gerencie contas bancárias com os seguintes atributos:

- Nome completo do titular
- Endereço do titular
- Saldo do titular

O cliente deve ser, ao final do projeto, capaz de:

- Criar
- Atualizar
- Obter
- Deletar

contas no servidor.

Resultados parciais

Atualmente estão implementados no cliente e no servidor as operações de:

- Criação
- Obtenção

Além disso toda *stack* de comunicação através dos *sockets* está implementada, bem como o *parsing* das requisições do lado do servidor, criação da requisição JSON do lado do cliente e a implementação de paralelismo com base num pool de *threads* POSIX que permanecem bloqueadas enquanto não há requisições a serem processadas.

Próximos passos

Implementação das funções de:

- Atualização
- Deleção

Recebimento do destino da conexão (endereço do servidor) como parâmetro do cliente.

Implementação

Observação:

Tanto no Client como no Server foi utilizado uma biblioteca que provê uma abstração para as operações com *socket*, além de uma biblioteca para serialização e deserialização de JSON.

Cliente

A entrada é primeiramente recebida pelo cliente através da entrada padrão, recebendo um parâmetro por linha. A primeira linha contém a operação a ser realizada, sendo os possíveis valores no estado atual:

- POST
- GET

A informação que vem a seguir depende da operação. Para o POST, as três seguintes linhas são compostas por:

- Nome
- Endereço
- Saldo

Caso a operação seja GET é inserida apenas mais uma única linha, informando o nome do titular.

O cliente então é responsável por transformar essa entrada em um JSON seguindo os formatos a seguir:

> ### POST
>
> {
> "name": "nome",
> "address": "endereço",
> "balance": 0,
> "operation": "POST"
> }
>
> ### GET
>
> {
> "name": "nome",
> "operation": "GET"
> }

A requisição em formato JSON é escrita no *socket* aberto com destino ao servidor na porta 8080. Na implementação atual o destino é 127.0.0.1, a interface de *loopback*, então o servidor deve estar sendo executado

previamente no mesmo *host* para que obtenha-se sucesso na criação do socket.

Feita a escrita o cliente é bloqueado aguardando resposta do servidor. Quando recebida esta é escrita na saída padrão e o programa termina.

Servidor

A thread principal é composta pela criação do *pool* de *working threads*, que irão processar as requisições, além do *welcoming socket*, responsável pela recepção de novas requisições.

Um dos principais obstáculos encontrados durante a implementação foi o gerenciamento do *pool*. A implementação inicial do projeto iniciava uma nova *thread* para cada requisição que "morria" ao finalizar sua rotina. A fim de reduzir o overhead e não permitir um *overcommitment* dos recursos do servidor escolheu-se a criação de um número fixo de *threads* no início do programa que seriam reutilizadas ao decorrer de sua execução.

Para implementação desta estratégia era necessário um meio de fornecer à *thread* o socket que seria utilizado para comunicação. Este não poderia ser passado como argumento na inicialização da *thread*, pois seriam utilizados diversos *sockets* ao decorrer da execução. Seria necessário então uma estrutura global. Inicialmente cogitou-se a criação de um vetor global de *sockets* indexados pelos IDs das *threads*. Entretanto, como saber qual thread seria acordada em seguida pela rotina pthread_cond_signal()? O acoplamento referencial dessa estratégia fez com que fosse abandonada por uma fila global de *sockets*. O fluxo é o seguinte:

1. Receber uma nova conexão
2. Inserir o *socket* gerado pelo accept() numa fila FIFO global
3. Sinalizar uma das threads bloqueadas que há trabalho a ser feito
4. Repetir

Quando sinalizada, a *thread* acorda, consome o primeiro *socket* da fila e o utiliza para receber a requisição. Ela é então processada e a estrutura de contas em memória é alterada ou consultada. A resposta é então elaborada e escrita no *socket*. Feito isso a thread se bloqueia novamente até ser sinalizada outra vez.

Mais detalhes da implementação podem ser encontrados nos comentários no código-fonte.

Compilação e execução

Na pasta raiz do servidor, execute:

```
$ g++ -pthread -o EPDSID ServerEndpoint.cpp socket/Socket.cpp
socket/Socket.h socket/ServerSocket.cpp socket/ServerSocket.h
model/AccountModel.cpp service/AccountService.cpp service/
AccountService.h ServerEndpoint.h socket/SocketException.h
```

O produto binário é o arquivo "EPDSID". O execute para iniciar o servidor. Este estará ouvindo na interface de *loopback* (127.0.0.1) na porta 8080/tcp.

```
$ ./EPDSID
```

Na pasta raiz do cliente, execute:

```
$ g++ -o client main.cpp socket/ClientSocket.cpp socket/Socket.cpp
```

O produto binário é o arquivo "client". O execute para iniciar o cliente. Este irá tentar conexão em 127.0.0.1:8080. Caso não estabeleça conexão, a execução é abortada.

```
$ ./client
```