

# Projeto DSID - 2019/01

---

Este projeto é resultado 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.

# Implementação

---

## Observação:

---

No cliente e no servidor foi utilizada uma biblioteca que provê uma abstração para as operações com *socket*, além de uma biblioteca para serIALIZAÇÃO e desserialização de JSON no servidor.

## 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:

- POST
- GET
- PUT
- DELETE

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 ou DELETE é inserida apenas mais uma única linha, informando o id da conta desejada.

Para o PUT, informa-se:

- Id
- Nome
- Endereço
- Saldo

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

```
{
  "id": 1,
  "operation": "GET"
}
```

## PUT

```
{
  "id": 1
  "name": "nome novo",
  "address": "endereço novo",
  "balance": 100,
  "operation": "PUT"
}
```

## DELETE

```
{
  "id": 1,
  "operation": "DELETE"
}
```

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 no relatório final e nos comentários no código-fonte.

# Compilação e execução

---

Na pasta raiz do projeto, execute:

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

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

```
$ ./servidor
```

Na pasta raiz do projeto, execute:

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

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

```
$ ./cliente
```

## Exemplos de uso

---

Depois de iniciado o servidor:

```
$ ./servidor
```

Iniciar o cliente:

```
$ ./cliente
```

Insira na entrada padrão do processo do cliente:

```
POST  
Ana
```

```
Rua 3
1000
```

O output deve ser:

```
Account Created!
id: 1
name: Ana
address: Rua 3
balance: 1000
```

Então podemos utilizar esse output para consultar uma conta usando o ID que foi atribuído:

```
GET
1
```

Deve retornar:

```
Found account!
name: Ana
address: Rua 3
balance: 1000
```

Ao buscarmos uma conta não existente:

```
GET
10000
```

Temos:

```
Account doesn't exist.
```

Para modificar uma conta:

```
PUT
1
Ana Rosa
Rua 4
10020
```

A saída deve ser:

```
Account Updated!
id:1
```

```
name: Ana Rosa  
address: Rua 4  
balance: 10020
```

Ao tentar atualizar uma conta não existente:

```
POST  
2  
Ana Rosa  
Rua 4  
10020
```

Temos:

```
Account doesn't exist.
```

E para deletar uma conta, basta informar a operação e o ID da conta:

```
DELETE  
1
```

Temos:

```
Account Deleted.
```

E ao tentar remover novamente a mesma conta:

```
Account doesn't exist.
```

## Logging

---

O servidor possui um log em logs/log.txt. Este arquivo é criado automaticamente caso não exista. O log pode ser utilizado para realizar-se uma análise do comportamento das *threads* e nele são gravadas as seguintes informações:

- Criação de *thread*;
- Inserção de *thread* no *pool*;
- Remoção de *thread* no *pool*;
- Início do processamento de uma requisição;
- Fim do processamento de uma requisição;