

# Banco Multithread

## Relatório de implementação do trabalho 1

### INE5645 - Programação Paralela e Distribuída

Giovane Pimentel de Sousa - 22202685  
Guilherme Henriques do Carmo - 22201630  
Isabela Vill de Aquino - 22201632

November 3, 2024

## Seção 1

### Introdução

O trabalho solicitado consiste na implementação de um servidor multithreaded para realizar operações bancárias usando um pool de threads e o modelo de produtor/consumidor.

### Decisões de Implementação

A implementação foi realizada em **Golang** 1.23, utilizando apenas módulos da biblioteca padrão, em principal "*sync*" para toda a estrutura não sequencial, portanto, não é necessária a instalação de nenhum pacote adicional. Pensamos em utilizar semáforos para controle de workers livres, porém em Go a forma oficial é com channels, que implementa dentro de si o controle de concorrência, o que era uma limitação estabelecida na definição do trabalho, portanto, decidimos por utilizar mutex e condition em toda estrutura de sincronização.

Foram escolhidas quatro parametrizações principais: **número de threads trabalhadoras**, **número de clientes**, **número máximo de requisições** que cada cliente efetuará e por fim o **tempo de serviço** (tempo de espera de cada operação).

Os clientes geram requisições em intervalos aleatórios e arbitrários de 0 a 5 segundos.

A variável *numClients* foi definida para ser tanto relacionada à quantidade de clientes quanto de contas existentes no banco.

Foi escolhido pelo grupo um ponto de parada: **X** clientes atingirem **N** (maxRequests) requisições definida por parâmetro. Portanto, quando o número de requisições atendidas pelos workers chegarem a  $X * N$ , o programa finaliza.

### Como executar

A sintaxe básica para execução é:

---

```
$ go run . [numWorkers] [numClients] [maxRequests] [serviceTimeInMiliseconds]
# Então por exemplo:
$ go run . 1 3 1 1000
$ go run . 2 5 2 300
```

---

Exemplos de execução:

```

> go run . 3 5 3 1000
Sacando na conta 3: 712.000000
Depositando na conta 1: 97.000000
Sacando na conta 4: 955.000000
Depositando na conta 3: 629.000000
Depositando na conta 3: 740.000000
Depositando na conta 1: 292.000000
Transferência: 5 → 4 : 315.000000
Sacando na conta 2: 811.000000
Sacando na conta 4: 611.000000
Depositando na conta 4: 540.000000
Balanco geral:
Account ID: 1   Account Balance: 389.00
Account ID: 2   Account Balance: -811.00
Account ID: 3   Account Balance: 657.00
Account ID: 4   Account Balance: -711.00
Account ID: 5   Account Balance: -315.00
Depositando na conta 3: 181.000000
Transferência: 3 → 1 : 85.000000
Transferência: 5 → 4 : 755.000000
Sacando na conta 2: 89.000000
Depositando na conta 5: 636.000000

```

```

> go run . 5 3 5 200
Depositando na conta 1: 597.000000
Transferência: 2 → 3 : 169.000000
Sacando na conta 3: 601.000000
Transferência: 3 → 1 : 683.000000
Sacando na conta 2: 266.000000
Sacando na conta 1: 201.000000
Transferência: 2 → 3 : 880.000000
Transferência: 3 → 1 : 621.000000
Transferência: 2 → 1 : 294.000000
Balanco geral:
Account ID: 1   Account Balance: 1994.00
Account ID: 2   Account Balance: -1609.00
Account ID: 3   Account Balance: -856.00
Depositando na conta 1: 695.000000
Transferência: 3 → 1 : 434.000000
Transferência: 2 → 1 : 812.000000
Transferência: 3 → 2 : 636.000000
Depositando na conta 2: 679.000000
Transferência: 1 → 3 : 243.000000

```

## Conclusões

Ao variar os parâmetros do sistema, observou-se que o número de threads no pool influencia diretamente a capacidade de processamento simultâneo do servidor. Um pool maior tende a reduzir o tempo de resposta das requisições sob cargas mais altas, mas aumenta o uso de recursos e a complexidade do gerenciamento de sincronização. Esses experimentos permitiram entender a importância de uma configuração equilibrada para otimizar o desempenho do servidor multithreaded conforme os requisitos de cada cenário.