

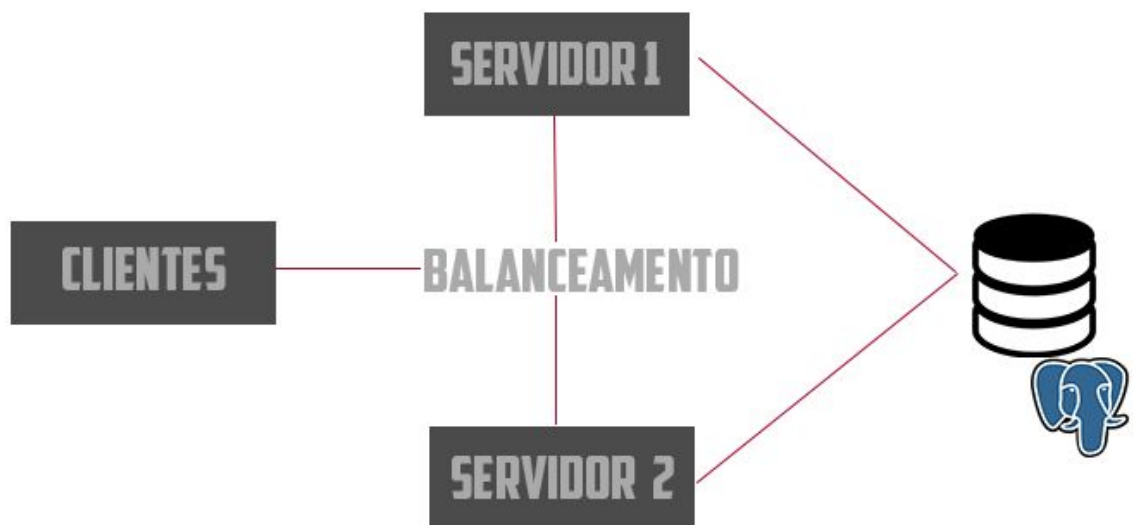
Márcio Rodrigues Filho
Lorenzo Antônio Leite
Lucas Wallace Nascimento Lima

Git do projeto : <https://github.com/mrodriguesfilho/banco-distribuido-sd1>

Sistema Bancário Distribuído

1. Organização lógica dos componentes do software

- 1.1. A visualização abaixo, permite ver como se comportam os componentes necessários para manter a aplicação do Sistema Bancário Distribuído executando;



- 1.2. Visto isso, é necessário definir o que são e para que servem cada componente da organização acima:

- 1.2.1. **Clientes:** o projeto deve conter pelo menos um cliente, para que este possa fazer as requisições no sistema bancário. Tendo em vista as necessidades de um cliente ideal em um banco, este possui um login e uma senha, além de um número único de conta e agência. Após entrar no sistema, (verificado através das pesquisas no banco de dados da tabela Contas) o cliente deverá especificar qual operação

quer realizar: consultar o saldo, consultar o extrato, transferência de valor, saque e depósito, além de poder escolher dar um 'log-off' no acesso;

1.2.2. Balanceamento: este componente foi definido de forma a realizar o balanceamento físico dos dados do Sistema Bancário Distribuído. Assim sendo, ele é um mediador entre o(s) cliente(s) e o servidor do banco. O balanceamento foi feito por meio de um operador ternário que direciona conexões, marcadas por um contador, se pares, vão para uma máquina e se ímpares para outra máquina;

1.2.3. Servidores: componentes do projeto que, rodando cada servidor em máquinas diferentes, mas com o mesmo banco de dados, irão receber as requisições dos clientes e, a partir do que foi requisitado, invocar remotamente aquele método do objeto Banco. O banco foi implementado com a tecnologia de middleware Java RMI.

1.2.4. Banco de dados: como o Sistema Bancário deve garantir a persistência dos dados, este componente é indispensável. Assim sendo, o banco foi implementado utilizando uma linguagem de consulta estruturada, em um banco com tabelas relacionais, ou seja, que se relacionam entre si. São elas: agência, conta, cliente, senha, saldo, extrato e data. O banco utilizado foi o PostgreSQL.

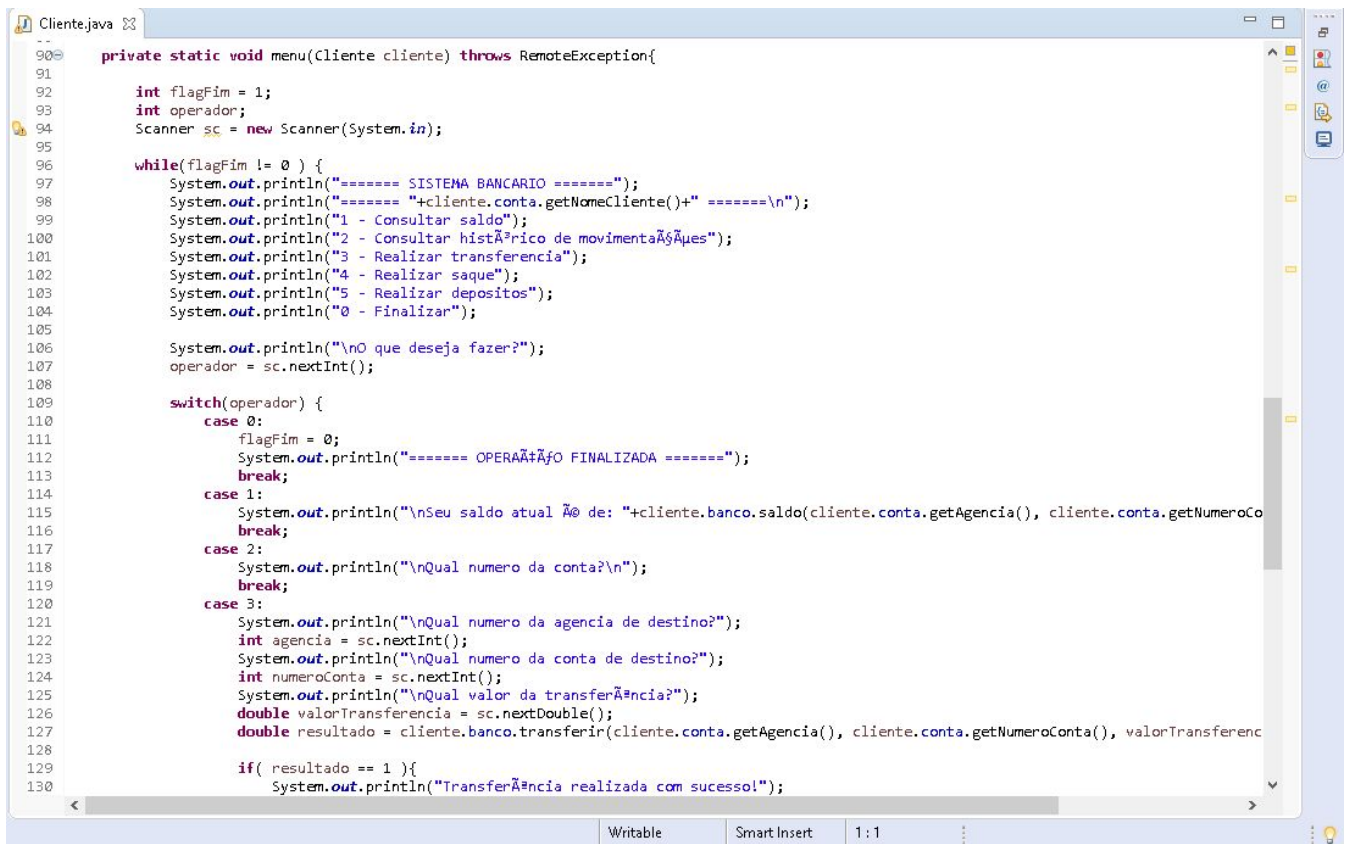
The screenshot shows the PGAdmin 4 interface. On the left, the 'Browser' pane displays the database structure, including a table named 'contas' with columns: numeroagencia, numeroconta, nomecliente, senha, and saldo. The 'Query Editor' pane shows a SQL query: `SELECT numeroagencia, numeroconta, nomecliente, senha, saldo FROM public.contas;`. The 'Data Output' pane displays the results of the query in a table format.

numeroagencia	numeroconta	nomecliente	senha	saldo
1	3	Lucas Wallace	123	10000
2	2	Enzo Leite	2424	1233
3	1	Márcio	123	101637
4	4	Diego	123	100000

A green notification box at the bottom right states: 'Successfully run. Total query runtime: 115 msec. 4 rows affected.'

2. Trechos do código fonte

2.1. Classe cliente



```
90 private static void menu(Cliente cliente) throws RemoteException{
91
92     int flagFim = 1;
93     int operador;
94     Scanner sc = new Scanner(System.in);
95
96     while(flagFim != 0 ) {
97         System.out.println("===== SISTEMA BANCARIO =====");
98         System.out.println("===== "+cliente.conta.getNomeCliente()+" =====\n");
99         System.out.println("1 - Consultar saldo");
100        System.out.println("2 - Consultar histórico de movimentações");
101        System.out.println("3 - Realizar transferência");
102        System.out.println("4 - Realizar saque");
103        System.out.println("5 - Realizar depósitos");
104        System.out.println("0 - Finalizar");
105
106        System.out.println("\nO que deseja fazer?");
107        operador = sc.nextInt();
108
109        switch(operador) {
110            case 0:
111                flagFim = 0;
112                System.out.println("===== OPERAÇÃO FINALIZADA =====");
113                break;
114            case 1:
115                System.out.println("\nSeu saldo atual é de: "+cliente.banco.saldo(cliente.conta.getAgencia(), cliente.conta.getNumeroCo
116                break;
117            case 2:
118                System.out.println("\nQual numero da conta?\n");
119                break;
120            case 3:
121                System.out.println("\nQual numero da agencia de destino?");
122                int agencia = sc.nextInt();
123                System.out.println("\nQual numero da conta de destino?");
124                int numeroConta = sc.nextInt();
125                System.out.println("\nQual valor da transferência?");
126                double valorTransferencia = sc.nextDouble();
127                double resultado = cliente.banco.transferir(cliente.conta.getAgencia(), cliente.conta.getNumeroConta(), valorTransferenc
128
129                if( resultado == 1 ){
130                    System.out.println("Transferência realizada com sucesso!");
```

Figura 1. Menu de operações.

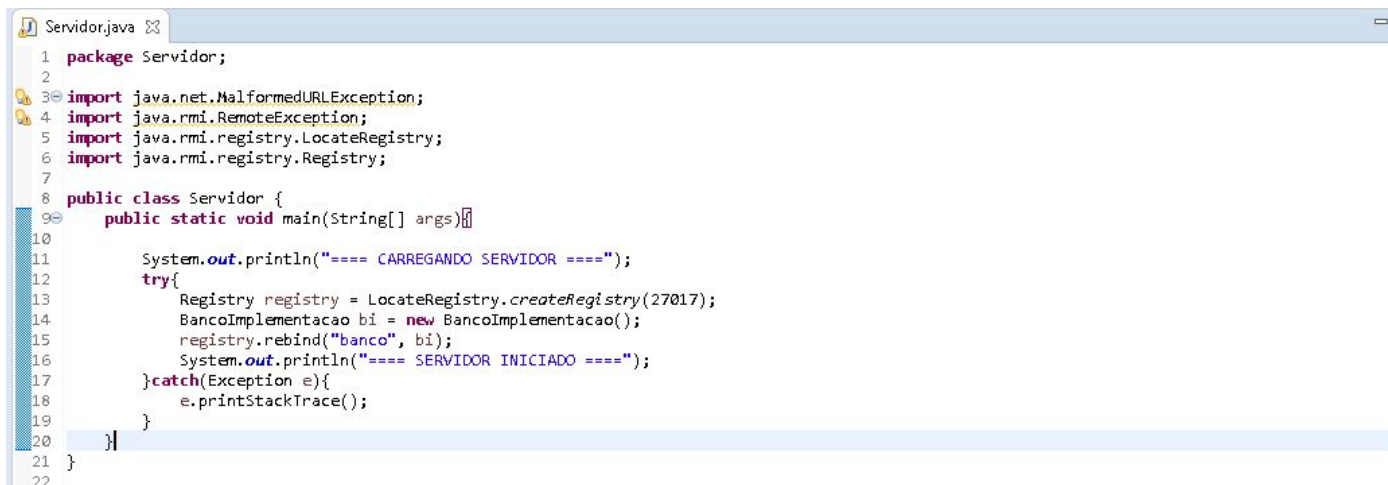
2.2. Classe ConnFactory



```
1 package Servidor;
2
3 import java.sql.*;
4
5 class ConnFactory {
6
7     public Connection getConnection() throws SQLException {
8         try {
9
10             Class.forName("org.postgresql.Driver");
11             Connection conn;
12             conn = DriverManager.getConnection("jdbc:postgresql://localhost/banco?user=postgres&password=admin");
13             return conn;
14
15         } catch (ClassNotFoundException e) {
16             throw new SQLException(e.getMessage());
17         }
18     }
19
20     public void runSQL() {
21
22     }
23 }
```

Figura 2. Conexão com o banco de dados, gerenciada pelo PostgreSQL.

2.3. Servidores



```
1 package Servidor;
2
3 import java.net.MalformedURLException;
4 import java.rmi.RemoteException;
5 import java.rmi.registry.LocateRegistry;
6 import java.rmi.registry.Registry;
7
8 public class Servidor {
9     public static void main(String[] args) {
10
11         System.out.println("==== CARREGANDO SERVIDOR ===");
12         try {
13             Registry registry = LocateRegistry.createRegistry(27017);
14             BancoImplementacao bi = new BancoImplementacao();
15             registry.rebind("banco", bi);
16             System.out.println("==== SERVIDOR INICIADO ===");
17         } catch (Exception e) {
18             e.printStackTrace();
19         }
20     }
21 }
22
```

Figura 3. Implementação para os servidores.

2.4. Balanceamento

```
protected BalanceamentoServidor() throws RemoteException, NotBoundException {
    super();
    connCounter++;
    Registry registry1, registry2;
    registry1 = LocateRegistry.getRegistry( host: "localhost", port: 27017);
    registry2 = LocateRegistry.getRegistry( host: "192.168.0.10", port: 27017);
    this.banco1 = (BancoInterface) registry1.lookup( name: "banco");
    this.banco2 = (BancoInterface) registry2.lookup( name: "banco");
}

public static void main(String[] args) {
    try {
        BalanceamentoServidor bsv = new BalanceamentoServidor();
        Registry registry = LocateRegistry.createRegistry( port: 27015);
        registry.rebind( name: "banco", bsv);
        System.out.println("=== BALANCEAMENTO INICIADO ===");
    } catch (Exception e) {
        System.out.println("Caixa erro: " + e.getMessage());
    }
}
```

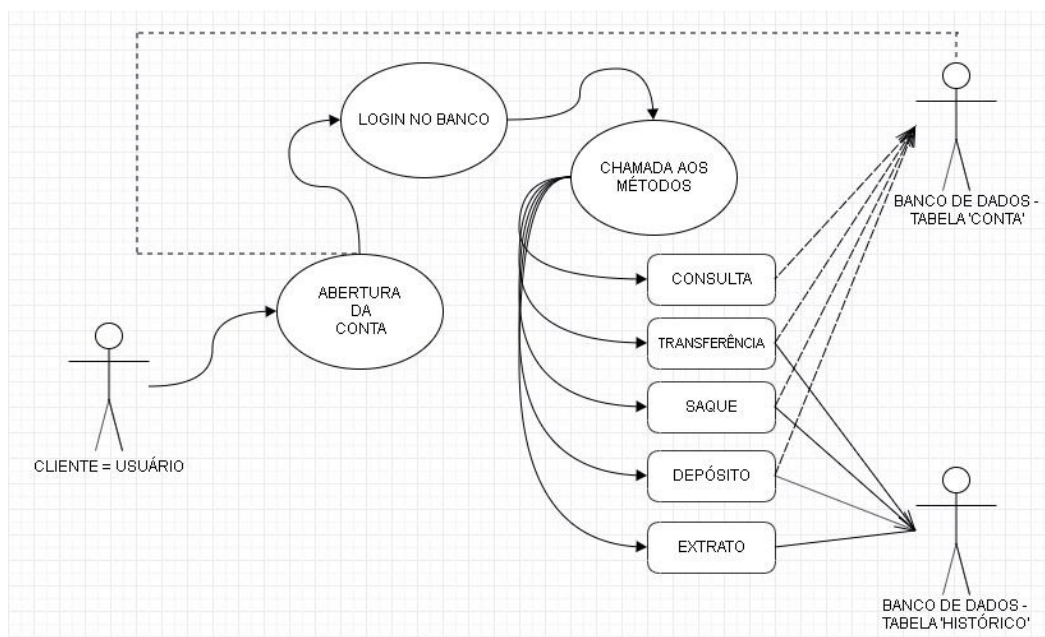
O balanceamento funciona da seguinte forma :

- Carregamos o servidor 1
- Carregamos o servidor 2
- Carregamos o balanceamento que conecta pelo RMI no servidor 1 e no servidor 2
- O cliente invoca os métodos do servidor de balanceamento
- As operações posteriormente são direcionadas por meio do contador para o servidor 1 ou servidor 2
- O banco de dados de cada servidor não se comunica com o outro e, portanto, o cliente não tem acesso sem a replicação.

3. Tecnologias Utilizadas

Para o projeto, foi utilizado o Java RMI para criar aplicações distribuídas em conjunto com a utilização do Java. Esta tecnologia faz uso de objetos remotos para seu funcionamento e permite que vários objetos clientes se conectem a um objeto servidor. Além disso, foi utilizado a API JDBC que providencia formas de executar comandos SQL dentro do código Java, diante disso, fez-se necessário o uso da API JPA para que pudesse definir um mapeamento objeto-relacional.

4. Casos de Uso



- 4.1. Login** - Ao rodarmos a aplicação do cliente o usuário é apresentado no console a opção de logar ou sair. Os dados para login são Agência, Número da Conta e a respectiva senha.

- 4.2. **Consulta de Saldo** - O cliente faz um uso simples da função saldo da interface remota que apenas faz a requisição no banco de dados e retorna o valor da conta
- 4.3. **Consulta de histórico de movimentações** - Nesta função o usuário ao requisitar o extrato interage com a interface por meio de um arraylist. Os dados de conta e agência são acessados pelo objeto que fez o login. A chamada do método remoto é feita sem a necessidade de input e um loop imprime os dados de uma tabela específica do banco de dados, histórico, que guarda todas as ações na conta.
- 4.4. **Transferência** - Neste método é necessário que usuário faça a entrada dos dados da conta destino. São necessários, todavia, apenas Agência e Conta. Dessa forma é feita uma query no banco para validar a existência da conta destino. Uma vez validada a existência, o valor da conta origem é checado para garantir que existe saldo suficiente. Por fim, satisfeitas validade da conta destino e saldo suficiente a transferência é realizada.
- 4.5. **Saque** - Neste método o usuário é requerido a entrada do valor a ser sacado de sua própria conta. Por meio de uma query é checado se esse valor está disponível e, se sim, o valor é subtraído no banco de dados.
- 4.6. **Depósito** - Neste método o depósito é realizado inspirado na forma física, de modo que é simplesmente feito uma query de update e o dinheiro é “gerado” no banco.
- 4.7. **Finalizar** - Fecha a sessão do banco e volta para o menu inicial de login.