Relatório do Sistema de Operações Bancárias com RPC em Go

Este relatório descreve um sistema de operações bancárias implementado em Go, utilizando RPC (Remote Procedure Call) para facilitar a comunicação entre cliente e servidor. O sistema foi projetado para oferecer funcionalidades básicas de um banco, garantindo segurança e eficiência nas transações.

Funcionalidades do Sistema

- Abertura de contas: Permite a criação de novas contas bancárias com saldo inicial zero.
- Fechamento de contas: Possibilita o encerramento de contas existentes, com devolução do saldo.
- Depósitos: Permite a adição de fundos a uma conta específica.
- Saques: Permite a retirada de fundos de uma conta, desde que haja saldo suficiente.
- Consulta de saldo: Fornece informações sobre o saldo atual de uma conta.

Arquitetura do Sistema

O sistema é baseado em uma arquitetura cliente-servidor, onde:

- Servidor: Gerencia as contas bancárias, processa as solicitações dos clientes e mantém a consistência dos dados.
- Cliente: Envia solicitações ao servidor para realizar operações bancárias e recebe as respostas.
- A comunicação entre cliente e servidor é realizada através de RPC, permitindo que os clientes chamem procedimentos no servidor como se fossem funções locais.

Detalhamento dos Arquivos

Arquivo server.go

Este arquivo implementa a lógica do servidor, incluindo:

- Definição da estrutura de dados para as contas bancárias.
- Implementação dos métodos RPC (ObtemSaldo, AbrirConta, FecharConta, Deposito, Saque).
- Uso de mutex para evitar condições de corrida em operações concorrentes.
- Inicialização de um conjunto de contas predefinidas para teste.
- Configuração do servidor RPC e gerenciamento de conexões de clientes.

Arquivos client.go e caixa.go

Estes arquivos implementam a lógica do cliente:

- client.go: Responsável por estabelecer a conexão com o servidor e enviar solicitações RPC.
- caixa.go: Implementa operações bancárias específicas, como depósitos e saques, utilizando goroutines para operações simultâneas.

Ambos os arquivos demonstram o uso de contas novas e antigas para testar diferentes cenários.

Exemplo de Execução

- Inicialização do servidor (executando server.go).
- Conexão do cliente ao servidor (através de client.go ou caixa.go).
- Realização de operações bancárias:
- Abertura de novas contas.
- Depósitos em contas existentes.
- Saques de contas com saldo suficiente.
- Consultas de saldo após as operações.
- O servidor processa as solicitações e envia respostas ao cliente.
- O cliente exibe os resultados das operações para o usuário.

Este sistema demonstra a aplicação prática de conceitos como RPC, concorrência em Go, e operações bancárias básicas em um ambiente distribuído.

Controle de idempotência:

A execução exactly-once visa garantir que cada operação seja executada exatamente uma vez, evitando duplicações e garantindo consistência, mesmo em situações de reenvio de mensagens. Implementar exactly-once requer algumas mudanças no código para rastrear operações já processadas, geralmente usando identificadores únicos (UUIDs) para cada requisição.

O controle de idempotência no código fornecido é implementado através dos seguintes mecanismos:

Geração de ID único: Função gerarIDOperacao() no cliente gera um UUID para cada operação.

Armazenamento de operações realizadas: Servidor utiliza um mapa operações Realizadas para armazenar resultados de operações processadas.

Verificação de idempotência: Antes de executar uma operação, o servidor verifica se o idOperacao já existe no mapa. Se existir, retorna o resultado armazenado sem reexecutar a operação.

Uso do ID de operação: Nas chamadas do cliente para o servidor, o idOperacao é incluído como argumento. Este mecanismo garante que: Operações enviadas múltiplas vezes são executadas apenas uma vez no servidor.Retornos de operações repetidas são consistentes.Duplicação de operações é evitada, mantendo a consistência dos dados.

O sistema implementa idempotência para todas as operações principais: Obtenção de saldo, abertura de conta, fechamento de conta, depósito, saque. Cada operação verifica o idOperacao antes de processar, garantindo que transações duplicadas não afetem o estado do sistema.

Teste

Para testagem do código foram feitos várias go routines gerenciadas por um waitGroup que adiciona contas e faz saques e depósitos em ordens diferentes.

Função Main do Servidor: ela realiza as seguintes tarefas: Inicializa um novo servidor na porta 1234. Registra o servidor para uso com RPC (Remote Procedure Call). Inicializa a lista de contas pré-definidas. Configura um listener TCP na porta especificada. Entra em um loop infinito para aceitar conexões de clientes. Para cada conexão aceita, inicia uma nova goroutine para lidar com as chamadas RPC. Esta função é responsável por manter o servidor em execução e pronto para receber solicitações dos clientes.

Funções Main dos Clientes: Existem duas versões da função main para o cliente no código fornecido. Ambas compartilham uma estrutura similar, mas com algumas diferenças nas operações realizadas.

Primeira versão da função main do Cliente realiza as seguintes operações: Verifica se o endereço da máquina do servidor foi fornecido como argumento. Estabelece uma conexão RPC com o servidor. Executa operações de depósito e saque em contas novas e antigas de forma concorrente. Verifica o saldo de todas as contas após as operações.

Segunda Versão da Função Main é similar à primeira, mas com uma diferença importante por se tratar da versão caixa automático naão consegue chamar funções de abertura e fechamento de contas.

Ambas as versões utilizam goroutines para executar operações concorrentemente, demonstrando o uso de programação paralela em Go. Elas também implementam um mecanismo de espera (WaitGroup) para garantir que todas as operações sejam concluídas antes do término do programa. As funções main dos clientes são essenciais para testar e demonstrar a funcionalidade do sistema bancário implementado, simulando múltiplas operações simultâneas em diferentes contas.

```
C:\Users\USER\Documents\GitHub\Progrmacao_paralela\T2>make
go build server.go
go build client.go
go build caixa.go
./server
Servidor aguardando conexões na porta 1234
Conta criada para  Bruno
Conta criada para  Izis
Conta criada para  Gabriel
Conta criada para  Carlos
Conta criada para  Sofia
Conta criada para  Enzo
Desposito Realizado com sucesso
Saque Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Desposito Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Desposito Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Desposito Realizado com sucesso
Saque Realizado com sucesso
Saque Realizado com sucesso
Saque Realizado com sucesso
Saque Realizado com sucesso
```

```
C:\Users\USER\Documents\GitHub\Progrmacao_paralela\T2>client.exe localhost
Resposta do servidor: Conta de Gabriel criada com sucesso!
Resposta do servidor: Conta de Izis criada com sucesso!
Resposta do servidor: Conta de Bruno criada com sucesso!
Resposta do servidor: Conta de Carlos criada com sucesso!
Resposta do servidor: Conta de Sofia criada com sucesso!
Resposta do servidor: Conta de Enzo criada com sucesso!
Resposta do servidor: Deposito de 200 feito, novo saldo de Bruno = 100
Resposta do servidor: Saque de 100 feito, novo saldo de Enzo = 100
Resposta do servidor: Saque de 100 feito, novo saldo de Joao = 756.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Izis = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Joao = 756.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Enzo = 100
Resposta do servidor: Saque de 100 feito, novo saldo de Gabriel = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Alexandre = 1000.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Carlos = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Maria = 1046
Resposta do servidor: Saque de 100 feito, novo saldo de Bruno = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Gabriel = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Paulo = 10646
Resposta do servidor: Saque de 100 feito, novo saldo de Maria = 1046
Resposta do servidor: Saque de 100 feito, novo saldo de Pedro = 7565
Resposta do servidor: Deposito de 200 feito, novo saldo de Barbara = 955.5
Resposta do servidor: Saque de 100 feito, novo saldo de Sofia = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Sofia = 100
Resposta do servidor: Saque de 100 feito, novo saldo de Carlos = 100
Resposta do servidor: Deposito de 200 feito, novo saldo de Pedro = 7565
Resposta do servidor: Saque de 100 feito, novo saldo de Izis = 100
Resposta do servidor: Saque de 100 feito, novo saldo de Paulo = 10646
Resposta do servidor: Saque de 100 feito, novo saldo de Barbara = 955.5
Resposta do servidor: Saque de 100 feito, novo saldo de Alexandre = 1000.5

C:\Users\USER\Documents\GitHub\Progrmacao_paralela\T2>
```

```
C:\Users\USER\Documents\GitHub\Progrmacao_paralela\T2>caixa.exe localhost
Resposta do servidor: Deposito de 200 feito, novo saldo de Sofia = 300
Resposta do servidor: Deposito de 200 feito, novo saldo de Pedro = 7665
Resposta do servidor: Deposito de 200 feito, novo saldo de Alexandre = 1100.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Enzo = 200
Resposta do servidor: Deposito de 200 feito, novo saldo de Izis = 200
Resposta do servidor: Deposito de 200 feito, novo saldo de Joao = 856.5
Resposta do servidor: Saque de 100 feito, novo saldo de Maria = 1146
Resposta do servidor: Deposito de 200 feito, novo saldo de Carlos = 200
Resposta do servidor: Deposito de 200 feito, novo saldo de Barbara = 1055.5
Resposta do servidor: Saque de 100 feito, novo saldo de Pedro = 7665
Resposta do servidor: Saque de 100 feito, novo saldo de Bruno = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Alexandre = 1100.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Maria = 1146
Resposta do servidor: Saque de 100 feito, novo saldo de Joao = 856.5
Resposta do servidor: Deposito de 200 feito, novo saldo de Gabriel = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Izis = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Enzo = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Sofia = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Barbara = 1055.5
Resposta do servidor: Saque de 100 feito, novo saldo de Carlos = 200
Resposta do servidor: Saque de 100 feito, novo saldo de Paulo = 10746
Resposta do servidor: Saque de 100 feito, novo saldo de Gabriel = 200
Resposta do servidor: Deposito de 200 feito, novo saldo de Paulo = 10746
Resposta do servidor: Deposito de 200 feito, novo saldo de Bruno = 200
Resposta do servidor: Conta de Sofia com R$ 200
Resposta do servidor: Conta de Gabriel com R$ 200
Resposta do servidor: Conta de Joao com R$ 856.5
Resposta do servidor: Conta de Pedro com R$ 7665
Resposta do servidor: Conta de Alexandre com R$ 1100.5
Resposta do servidor: Conta de Enzo com R$ 200
Resposta do servidor: Conta de Barbara com R$ 1055.5
Resposta do servidor: Conta de Carlos com R$ 200
Resposta do servidor: Conta de Paulo com R$ 10746
Resposta do servidor: Conta de Maria com R$ 1146
Resposta do servidor: Conta de Bruno com R$ 200
Resposta do servidor: Conta de Izis com R$ 200

C:\Users\USER\Documents\GitHub\Progrmacao_paralela\T2>
```

Inicio arquivo server.go:

```
package main

import (
	"fmt"
	"net"
	"net/rpc"
	"strconv"
	"sync"
)

var mutex sync.Mutex

// Estrutura para representar um aluno
type Conta struct {
	Nome  string
	Saldo float64
}

// Estrutura para o servidor
type Servidor struct {
	contas		[]Conta
	operacoesRealizadas map[string]string
}

// Método para inicializar a lista de alunos no servidor
func (s *Servidor) inicializar() {
	s.contas = []Conta{
		{"Alexandre", 900.5},
		{"Barbara", 855.5},
		{"Joao", 656.5},
		{"Maria", 946.0},
		{"Paulo", 10546.0},
		{"Pedro", 7465.0},
	}
}

// Método remoto que retorna o saldo de um cliente dado o seu nome
func (s *Servidor) ObtemSaldo(args []string, resposta *string) error {
	nome := args[0]
	idOperacao := args[1]

	if resultado, existe := s.operacoesRealizadas[idOperacao]; existe {
		*resposta = resultado
		return nil // Operação já foi processada
	}

	mutex.Lock()
	for _, conta := range s.contas {
		if conta.Nome == nome {
			fmt.Println("Saldo verificado para ", nome)
			*resposta = fmt.Sprintf("Conta de %s com R$ %g", nome, conta.Saldo)
			mutex.Unlock()
			return nil
		}
	}
	mutex.Unlock()
	return fmt.Errorf("Aluno %s não encontrado", nome)

}

func (s *Servidor) AbrirConta(args []string, resposta *string) error {

	nome := args[0]
	idOperacao := args[1]

	if resultado, existe := s.operacoesRealizadas[idOperacao]; existe {
```

```
		*resposta = resultado
		return nil // Operação já foi processada
	}

	// mutex.Lock()
	for _, conta := range s.contas {
		if conta.Nome == nome {
			fmt.Println("Conta existente para ", nome)
			*resposta = fmt.Sprintf("Conta com ", nome, "ja existe")
			// mutex.Unlock()
			return fmt.Errorf("conta com nome %s encontrada", nome)
		}
	}
	conta := Conta{
		Nome:  nome,
		Saldo: 0.0,
	}
	s.contas = append(s.contas, conta)
	fmt.Println("Conta criada para ", nome)
	*resposta = fmt.Sprintf("Conta de %s criada com sucesso!", nome)
	// mutex.Unlock()
	return nil
}

func (s *Servidor) FecharConta(args []string, resposta *string) error {
	nome := args[0]
	idOperacao := args[1]

	if resultado, existe := s.operacoesRealizadas[idOperacao]; existe {
		*resposta = resultado
		return nil // Operação já foi processada
	}
	mutex.Lock()
	for i, a := range s.contas {
		if a.Nome == nome {
			// Remove a conta da lista
			fmt.Println("Conta excluida de ", nome)
			*resposta = fmt.Sprintf("Conta removida com sucesso e saldo devolvido = %g", a.Saldo)
			s.contas = append(s.contas[:i], s.contas[i+1:]...)
			mutex.Unlock()
			return nil

		}
	}
	mutex.Unlock()
	*resposta = "Conta não encontrada."
	return fmt.Errorf("conta com nome %s não encontrada", nome)
}

func (s *Servidor) Deposito(args []string, resposta *string) error {
	nome := args[0]
	idOperacao := args[2]
	saldo, err := strconv.ParseFloat(args[1], 64)
	if err != nil {
		fmt.Println("Saldo inválido:", err)
		return fmt.Errorf("Valor invalido = ", nome)
	}
	if resultado, existe := s.operacoesRealizadas[idOperacao]; existe {
		*resposta = resultado
		return nil // Operação já foi processada
	}
	// mutex.Lock()
	for i, a := range s.contas {
		if a.Nome == nome {
			s.contas[i].Saldo += saldo
			fmt.Println("Desposito Realizado com sucesso ")
```

;

```
*resposta = fmt.Sprintf("Deposito de %g feito, novo saldo de %s = %g", saldo, nome, s.contas[i].Saldo)
                                // mutex.Unlock()
                                return nil
                        }
                }
                fmt.Println("Erro no deposito de %d ", nome)
                // mutex.Unlock()
                return fmt.Errorf("conta com nome %s não encontrada", nome)
}

func (s *Servidor) Saque(args []string, resposta *string) error {
        nome := args[0]
        idOperacao := args[1]
        saldo, err := strconv.ParseFloat(args[1], 64)
        if err != nil {
                fmt.Println("Saldo inválido:", err)
                return fmt.Errorf("Valor invalido = ", nome)
        }
        if resultado, existe := s.operacoesRealizadas[idOperacao]; existe {
                *resposta = resultado
                return nil // Operação já foi processada
        }
        // mutex.Lock()
        for i, a := range s.contas {
                if a.Nome == nome {
                        s.contas[i].Saldo -= saldo
                        fmt.Println("Saque Realizado com sucesso ")
                        *resposta = fmt.Sprintf("Saque de %g feito, novo saldo de %s = %g", saldo, nome,
s.contas[i].Saldo)
                        // mutex.Unlock()
                        return nil
                }
        }
        fmt.Println("Erro no deposito de %d ", nome)
        // mutex.Unlock()
        return fmt.Errorf("conta com nome %s não encontrada", nome)
}

func main() {
        porta := 1234
        servidor := new(Servidor)
        servidor.inicializar()
        rpc.Register(servidor)
        l, err := net.Listen("tcp", "0.0.0.0:1234")
        if err != nil {
                fmt.Println("Erro ao iniciar o servidor:", err)
                return
        }

        fmt.Println("Servidor aguardando conexões na porta", porta)
        for {
                conn, err := l.Accept()
                if err != nil {
                        fmt.Println("Erro ao aceitar conexão:", err)
                        continue
                }
                go rpc.ServeConn(conn)
        }
}
Fim server.go
```

Inicio Caixa.go:

```
package main
import (
	"fmt"
	"net/rpc"
	"os"
	"sync"
	"uuid-master"
)

func gerarIDOperacao() string {
	return uuid.New().String()
}

var contas_novas = []string{"Bruno", "Sofia", "Izis", "Enzo", "Carlos", "Gabriel"}
var contas_antigas = []string{"Maria", "Pedro", "Joao", "Alexandre", "Barbara", "Paulo"}

// Estrurura wait para Goroutines
var wg sync.WaitGroup

type Conta struct {
	Nome  string
	Saldo float64
}

// Metodo de verificação do saldo disponivel para um nome
func SALDO(nome string, cliente *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
	var err = cliente.Call("Servidor.ObtemSaldo", args, &resposta)
	if err != nil {
		fmt.Println("Erro ao ver saldo conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()
}

// Metodo de Abrir conta
func ABRIR(nome string, cliente *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
	var err = cliente.Call("Servidor.AbrirConta", args, &resposta)
	if err != nil {
```

```
		fmt.Println("Erro ao abrir conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de Fechar  conta
func FECHAR(nome string, client *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
	var err = client.Call("Servidor.FecharConta", args, &resposta)
	if err != nil {
		fmt.Println("Erro ao fechar conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de Depositar em conta
func DEPOSITO(nome string, client *rpc.Client) {
	var resposta string
	var val = "200.0"
	args := []string{nome, val, gerarIDOperacao()}
	var err = client.Call("Servidor.Deposito", args, &resposta)
	if err != nil {
		fmt.Println("Erro no deposito conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de saque em conta
func SAQUE(nome string, client *rpc.Client) {
	var resposta string
	var val = "100.0"
	args := []string{nome, val, gerarIDOperacao()}
	var err = client.Call("Servidor.Saque", args, &resposta)
	if err != nil {
		fmt.Println("Erro no saque conta:", err)
```

```
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Uso:", os.Args[0], "<maquina>")
		return
	}
	porta := 1234
	maquina := os.Args[1]
	client, err := rpc.Dial("tcp", fmt.Sprintf("%s:%d", maquina, porta))
	if err != nil {
		fmt.Println("Erro ao conectar ao servidor:", err)
		return
	}
	defer client.Close()

	for i := 0; i < len(contas_novas); i++ {
		wg.Add(1)
		go ABRIR(contas_novas[i], client)
	}
	wg.Wait()
	for i := 0; i < len(contas_antigas); i++ {
		wg.Add(1)
		go DEPOSITO(contas_novas[i], client)
		wg.Add(1)
		go DEPOSITO(contas_antigas[i], client)
	}
	for i := 0; i < len(contas_antigas); i++ {
		wg.Add(1)
		go SAQUE(contas_antigas[i], client)
		wg.Add(1)
		go SAQUE(contas_novas[i], client)
	}
	wg.Wait()
	// for i := 0; i < 3; i++ {
	//	FECHAR(contas_antigas[i], client)
	// }

}
```

Fim Cliente.go

Inicio Caixa.go

```
package main

import (
	"fmt"
	"net/rpc"
	"os"
	"sync"
	"uuid-master"
)

func gerarIDOperacao() string {
	return uuid.New().String()
}

var contas_novas = []string{"Bruno", "Sofia", "Izis", "Enzo", "Carlos", "Gabriel"}
var contas_antigas = []string{"Maria", "Pedro", "Joao", "Alexandre", "Barbara", "Paulo"}

// Estrurura wait para Goroutines
var wg sync.WaitGroup

type Conta struct {
	Nome  string
	Saldo float64
}

// Metodo de verificação do saldo disponivel para um nome
func SALDO(nome string, cliente *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
	var err = cliente.Call("Servidor.ObtemSaldo", args, &resposta)
	if err != nil {
		fmt.Println("Erro ao ver saldo conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()
}

// Metodo de Abrir conta
func ABRIR(nome string, cliente *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
```

```
	var err = cliente.Call("Servidor.AbrirConta", args, &resposta)
	if err != nil {
		fmt.Println("Erro ao abrir conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de Fechar  conta
func FECHAR(nome string, client *rpc.Client) {
	var resposta string
	args := []string{nome, gerarIDOperacao()}
	var err = client.Call("Servidor.FecharConta", args, &resposta)
	if err != nil {
		fmt.Println("Erro ao fechar conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de Depositar em conta
func DEPOSITO(nome string, client *rpc.Client) {
	var resposta string
	var val = "200.0"
	args := []string{nome, val, gerarIDOperacao()}
	var err = client.Call("Servidor.Deposito", args, &resposta)
	if err != nil {
		fmt.Println("Erro no deposito conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

// Metodo de saque em conta
func SAQUE(nome string, client *rpc.Client) {
	var resposta string
	var val = "100.0"
	args := []string{nome, val, gerarIDOperacao()}
	var err = client.Call("Servidor.Saque", args, &resposta)
```

```
	if err != nil {
		fmt.Println("Erro no saque conta:", err)
	} else {
		fmt.Println("Resposta do servidor:", resposta)
	}
	wg.Done()

}

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Uso:", os.Args[0], "<maquina>")
		return
	}
	porta := 1234
	maquina := os.Args[1]
	client, err := rpc.Dial("tcp", fmt.Sprintf("%s:%d", maquina, porta))
	if err != nil {
		fmt.Println("Erro ao conectar ao servidor:", err)
		return
	}
	defer client.Close()

	for i := 0; i < len(contas_antigas); i++ {
		wg.Add(1)
		go DEPOSITO(contas_novas[i], client)
		wg.Add(1)
		go DEPOSITO(contas_antigas[i], client)
	}
	for i := 0; i < len(contas_antigas); i++ {
		wg.Add(1)
		go SAQUE(contas_antigas[i], client)
		wg.Add(1)
		go SAQUE(contas_novas[i], client)
	}
	wg.Wait()
	for i := 0; i < len(contas_antigas); i++ {
		wg.Add(1)
		go SALDO(contas_antigas[i], client)
		wg.Add(1)
		go SALDO(contas_novas[i], client)
	}
	wg.Wait()

} Fim Caixa.go
```