Relatório Técnico Etapa 1 do trabalho de Sistemas distribuídos

1. Arquitetura do Sistema

O sistema foi desenvolvido seguindo a arquitetura **MVC distribuída**:

- **Model:**
  BankAccount.java e Database.java – responsáveis por armazenar, persistir e manipular os dados das contas bancárias.
- **View:**
  Menu.java – interface de linha de comando (CLI) que permite interação do usuário com o sistema.
- **Controller:**
  Group.java – integra a lógica de replicação e comunicação entre nós, utilizando o **JGroups** (por meio de RpcDispatcher e MethodCall).

O sistema opera de forma distribuída através do middleware **JGroups**, permitindo replicação horizontal: múltiplas instâncias do servidor participam de um mesmo canal, compartilhando o estado do sistema (lista de contas, saldos, histórico etc.).

2. Principais Decisões de Projeto

- **JGroups** foi escolhido para replicação e comunicação, aproveitando a simplicidade do modelo RPC (chamadas de métodos remotos) junto com a transferência de estado.
- Toda a lógica de persistência fica centralizada em Database.java, que lê e grava o arquivo banco_estado.dat, permitindo recuperar o estado após reinicializações.
- Cada cliente é identificado pelo **CPF**, garantindo unicidade.
- Métodos getState() e setState() do JGroups são usados para sincronizar automaticamente o estado global quando novos nós ingressam no cluster.
- Separação clara: Menu roda como cliente e chama métodos do Group, que por sua vez executa operações sincronizadas no banco distribuído.

3. Pontos Fortes da Solução

- **Replicação automática do estado**: novas instâncias recuperam o histórico e contas via getState()/setState().
- **Interface CLI** simples que facilita testes e demonstração.
- **Persistência local** robusta: mantém os dados mesmo após falhas ou desligamentos.
- **Evita inconsistência**: uso de synchronized nas operações críticas para garantir segurança em acessos concorrentes.

- **RPC funcional**: implementação com RpcDispatcher permite executar métodos remotamente entre os nós.

4. Pontos Fracos da Solução

- **Canal único:** não há divisão entre controle e dados; todos os nós estão no mesmo canal.
- **Faltam mecanismos de segurança:** como criptografia e autenticação robusta.
- **Sem testes de desempenho ou carga:** não há análise quantitativa de throughput ou latência.
- **Balanceamento limitado:** o coordenador do cluster concentra requisições.
- **Interface textual:** falta interface gráfica ou web para melhor experiência do usuário.

5. Justificativa da Pilha de Protocolos (JGroups)

A pilha foi definida no config.xml e inclui:

- UDP (Multicast): para descoberta automática e comunicação eficiente entre nós.
- GMS (Group Membership Service): gerencia a entrada e saída de membros e detecta falhas.
- NAKACK2: garante entrega confiável e ordenada das mensagens.
- STATE_TRANSFER: permite enviar o estado completo de um nó para outro ao ingressar no cluster.

Essa configuração busca **equilibrar simplicidade e robustez**, sendo fácil de alterar conforme necessidades futuras.

6. Estado do Sistema Compartilhado

Entre os nós participantes do cluster, é compartilhado:

- O **mapa de contas bancárias** (ConcurrentHashMap: CPF → BankAccount).
- O histórico de operações por conta.
- O saldo total do banco (calculado dinamicamente).

A sincronização ocorre via getState()/setState() quando novos servidores entram, garantindo consistência.

📎 Anexos

- Código-fonte completo (*.java)
- Bytecodes (*.class) na pasta out/
- Arquivo de configuração: config.xml
- Scripts de compilação e execução:
    - compile_e_executa_server.bat e compile_e_executa_server.sh
    - compile_e_executa_cliente.bat e compile_e_executa_cliente.sh
- Arquivo de estado: banco_estado.dat
- Biblioteca: lib/jgroups-3.6.4.Final.jar

Para compilar e executar utilize os **scripts prontos**:

- Servidor: compile_e_executa_server.*
- Cliente: compile_e_executa_cliente.*

Esses scripts geram os .class em /out e iniciam automaticamente as classes principais:

- Servidor: remote.BankServer
- Cliente: view.Menu