Instituto Federal de Educação, Ciência e Tecnologia da Bahia

Campus: Santo Antônio de Jesus **Data**: 27/08/2025

Curso: Analise e Desenvolvimento de Sistemas

Disciplina: Sistemas Distribuídos **Docente**: Felipe Silva

Discentes: Anderson Guilherme Santana Moraes e Carlos de Jesus Conceição Junior

Relatório Atividade 7 - Controle Colaborativo com Exclusão Mútua e Recuperação de Falhas

Sumário

1. Introdução

1.1. Contexto e Motivação

Este relatório técnico detalha o projeto e a implementação de um sistema distribuído de controle colaborativo. O objetivo central é simular um ambiente onde múltiplos processos concorrem pelo acesso a um recurso compartilhado, neste caso, um documento de texto. O projeto materializa conceitos teóricos fundamentais, como exclusão mútua, replicação de dados, controle de concorrência com relógios lógicos e, crucialmente, mecanismos avançados de tolerância a falhas. A solução desenvolvida em Java não apenas cumpre os requisitos básicos, mas estende a arquitetura para suportar a falha do coordenador central através de um processo de eleição de líder autônomo, garantindo alta disponibilidade ao sistema.

1.2. Objetivos do Projeto

O objetivo geral é construir um sistema consistente que demonstre o funcionamento integrado de múltiplos algoritmos clássicos de sistemas distribuídos. Os objetivos específicos são:

- Implementar uma arquitetura híbrida com um coordenador dinâmico e quatro nós replicados, utilizando Sockets TCP para comunicação.
- Garantir a exclusão mútua no acesso ao recurso compartilhado por meio do algoritmo Centralizado.
- Manter a consistência eventual dos dados através de uma estratégia de replicação passiva.
- Assegurar a ordenação causal das requisições utilizando Relógios de Lamport.
- Desenvolver um mecanismo completo de tolerância a falhas, combinando checkpoints periódicos, eleição de líder (Algoritmo Bully) para recuperação de falha do coordenador (failover) e rollback de operações críticas interrompidas.

2. Arquitetura do Sistema Distribuído

2.1. O Modelo Híbrido: Coordenador Dinâmico e Nós Replicados

A arquitetura do sistema é híbrida, composta por um conjunto de processos pares (nós) onde um deles assume dinamicamente o papel de coordenador. Inicialmente, o nó com o maior ID é definido como o coordenador padrão. No entanto, diferentemente de um modelo estático, a falha deste coordenador não paralisa o sistema. Em vez disso, um novo coordenador é eleito entre os nós sobreviventes, tornando a arquitetura resiliente e eliminando o ponto único de falha.

Cada um dos quatro nós do sistema mantém uma réplica local do documento compartilhado, atuando como cliente ao solicitar acesso e como servidor ao participar do processo de eleição. O coordenador, por sua vez, é responsável por gerenciar a fila de requisições, conceder permissões de acesso e difundir as atualizações do documento.

2.2. Comunicação Baseada em Sockets TCP

Toda a comunicação no sistema, seja entre um nó e o coordenador ou entre nós durante uma eleição, é realizada através de Sockets TCP. A escolha do TCP se justifica pela sua confiabilidade. A garantia de entrega e a natureza orientada à conexão do TCP são cruciais para:

1. **Comunicação com o Coordenador:** Assegurar que as solicitações e permissões não sejam perdidas.
2. **Detecção de Falhas:** Uma falha ao estabelecer ou manter uma conexão TCP com o coordenador é um indicador inequívoco de que o mesmo está inoperante, servindo como o principal gatilho para o processo de recuperação.

3. Exclusão Mútua e Controle de Concorrência

3.1. Justificativa do Algoritmo Centralizado

Para garantir a exclusão mútua, foi implementado o Algoritmo Centralizado. A escolha foi motivada pela sua simplicidade, eficiência e adequação à nossa arquitetura híbrida. Neste modelo, um nó que deseja acessar o recurso crítico envia uma única mensagem de REQUISICAO_SC ao coordenador. O coordenador gerencia uma fila de pedidos e responde com uma mensagem de PERMISSAO_SC quando o recurso está livre. Ao final da operação, o nó envia uma mensagem de LIBERACAO_SC.

A principal vantagem é a eficiência: cada operação de acesso ao recurso requer apenas três mensagens, independentemente do número de nós no sistema. A desvantagem, o ponto único de falha do coordenador, foi mitigada pela nossa implementação de eleição de líder, tornando a escolha deste algoritmo a mais lógica e performática para o nosso cenário.

3.2. Ordenação Causal com Relógios de Lamport

Para garantir a justiça e a ordem causal das solicitações, o sistema utiliza Relógios de Lamport. Cada nó mantém um contador lógico (AtomicInteger) que é incrementado a cada evento interno ou envio de mensagem. Ao receber uma mensagem, o nó ajusta seu relógio para max(relogio_local, relogio_mensagem) + 1.

Quando um nó envia uma REQUISICAO_SC, ele anexa o valor atual do seu relógio. O coordenador utiliza esse timestamp (junto com o ID do nó, para desempate) para ordenar os pedidos em uma PriorityQueue. Isso garante que as solicitações sejam atendidas na ordem em que "aconteceram antes", respeitando a causalidade, mesmo que cheguem fora de ordem devido a latências de rede.

4. Replicação de Dados e Consistência Eventual

4.1. Modelo de Replicação Passiva

O sistema implementa uma forma de replicação passiva (ou primário-backup). O coordenador detém a cópia primária (mestre) do documento. Quando um nó obtém acesso à seção crítica, ele modifica sua cópia local. Ao liberar o recurso, o nó envia a versão atualizada do documento junto com a mensagem de LIBERACAO_SC. O coordenador então atualiza sua cópia mestre e, de forma assíncrona, propaga esta nova versão para todos os outros nós conectados.

4.2. Análise da Consistência Eventual

Este modelo de replicação resulta em consistência eventual. Existe uma janela de tempo entre o momento em que o coordenador atualiza o documento mestre e o momento em que todas as réplicas recebem a atualização. Durante esse período, diferentes nós podem ter visões ligeiramente diferentes do documento. Para uma aplicação colaborativa não-crítica como a edição de um log, este modelo é aceitável, pois o sistema eventualmente converge para um estado consistente. A principal vantagem é a performance, pois o nó que realizou a alteração não precisa esperar a confirmação de todas as réplicas.

5. Estratégia de Tolerância a Falhas

Em vez de apenas implementar um mecanismo de rollback simples, foi desenvolvida uma solução de failover completa e autônoma. Esta estratégia multifacetada torna o sistema resiliente à falha do seu componente mais crítico, o coordenador, garantindo alta disponibilidade e a integridade dos dados. A seguir, detalhamos as quatro camadas que compõem esta arquitetura resiliente.

5.1. Detecção de Falha do Coordenador

A detecção de falhas no sistema adota um modelo implícito e reativo, que se destaca pela sua eficiência e simplicidade. Em vez de empregar um mecanismo de *heartbeat* ativo (que consumiria largura de banda com mensagens periódicas de "estou vivo"), a detecção é uma consequência natural da comunicação orientada à conexão do TCP.

Quando um nó tenta se comunicar com o coordenador, seja para enviar uma REQUISICAO_SC ou uma LIBERACAO_SC, a camada de transporte TCP garante a tentativa de entrega. Se o processo do coordenador falhou, o sistema operacional fechará seu socket, fazendo com que qualquer tentativa de escrita subsequente por parte do nó cliente resulte em uma java.io.IOException. Este evento é um indicador inequívoco e imediato de que o coordenador está inalcançável. Em vez de ser tratada como um simples erro, essa exceção é o gatilho deliberado que inicia o protocolo de recuperação, acionando o processo de eleição de um novo líder.

5.2. Eleição de Líder com o Algoritmo Bully

Uma vez detectada a falha, o sistema precisa eleger um novo coordenador para restaurar sua funcionalidade. Para isso, foi implementado o Algoritmo Bully, escolhido por sua lógica descentralizada e seu resultado previsível, que garante a eleição do nó ativo de maior ID. O processo, encapsulado na classe No.java, ocorre da seguinte forma:

1. **Início da Eleição:** O nó que detectou a falha assume o papel de coordenador da eleição e envia uma mensagem ELECTION para todos os outros nós com um ID superior ao seu. Este ato é uma sondagem para descobrir se existe algum nó "mais forte" (com ID maior) na rede.

2. **Resposta (O "Bullying"):** Se um nó com ID maior recebe a mensagem ELECTION, ele efetivamente "intimida" o nó de ID menor. Ele responde com uma mensagem OK, sinalizando que tomará para si a responsabilidade de continuar o processo eleitoral. Ao receber este OK, o nó de ID menor abandona suas ambições de liderança e assume uma postura passiva, aguardando o anúncio do vencedor.

3. **Condição de Vitória:** Um nó se considera o vencedor da eleição se, após enviar suas mensagens ELECTION para todos os nós de ID superior, um período de *timeout* se esgota sem que ele receba nenhuma mensagem OK em resposta. Isso significa que ou não existem nós com ID maior, ou todos eles falharam. Neste momento, ele é, por definição, o mais forte da rede.

4. **Anúncio e Transição:** O vencedor assume o papel de coordenador, instancia e inicia seu próprio ServicoCoordenador, e então anuncia sua liderança a todos os outros nós do sistema através de uma mensagem VICTORY. Ao receberem esta mensagem, os demais nós atualizam sua variável coordinatorId, encerram qualquer processo de eleição em andamento e iniciam o processo de reconexão ao novo líder, restaurando a operação normal do sistema.

5.3. Checkpoints e Recuperação de Estado

Para garantir a durabilidade dos dados e prevenir a perda de estado em caso de falha do coordenador, o sistema implementa um mecanismo de checkpoints periódicos. A cada 30 segundos, o ServicoCoordenador ativo serializa o objeto documentoMestre e o salva no arquivo checkpoint.dat.

A importância deste mecanismo se manifesta no momento da recuperação: quando um novo coordenador é eleito e inicia seu serviço, sua primeira ação é verificar a existência deste arquivo. Se encontrado, ele desserializa o conteúdo e o carrega como o estado inicial do documentoMestre. Esta estratégia garante que todas as alterações confirmadas sob a liderança do coordenador anterior sejam preservadas, permitindo que o sistema continue a partir do último estado consistente conhecido, em vez de reiniciar com um documento vazio. O intervalo de 30 segundos representa um balanço

entre o risco de perda de dados (no máximo 30 segundos de trabalho) e o custo de I/O de disco.

5.4. Rollback de Operações para Integridade do Recurso

Além da falha do coordenador, o sistema também é resiliente à falha de um nó cliente enquanto este detém o acesso exclusivo ao recurso crítico. Se a conexão TCP com este nó é perdida, o método removerNo do coordenador é acionado.

Este método contém uma lógica crucial de rollback: ele verifica se o ID do nó desconectado corresponde ao idNoEmSecaoCritica. Se a verificação for positiva, significa que o nó falhou no meio de uma operação crítica. O coordenador então:

1. Libera imediatamente o bloqueio sobre o recurso (recursoOcupado = false).
2. Descarta qualquer alteração que o nó falho pudesse ter enviado, pois a operação nunca foi formalmente concluída com uma mensagem de LIBERACAO_SC.
3. Concede a permissão ao próximo nó na fila de requisições.

Este mecanismo é vital para a saúde do sistema, pois previne cenários de *deadlock* onde o recurso ficaria permanentemente bloqueado devido à falha de um nó cliente.

6. Avaliação dos Cenários de Teste

Os testes foram conduzidos através da classe Simulador.java, que orquestra a inicialização e a simulação de falhas.

6.1. Teste 1: Operação em Estado Normal

- **Cenário:** O sistema é iniciado, P4 se torna o coordenador, e os outros nós (P1, P2, P3) se conectam a ele. Os nós solicitam acesso ao recurso em intervalos de tempo aleatórios.
- **Resultado Observado:** Sucesso. Os logs de execução mostram claramente que o coordenador enfileira os pedidos corretamente com base em seus relógios de Lamport e concede a permissão a um nó de cada vez. As atualizações do documento são propagadas para todos após cada liberação.

6.2. Teste 2: Falha e Recuperação do Coordenador

- **Cenário:** Após 45 segundos de operação normal, o Simulador interrompe a thread do coordenador P4, simulando uma falha abrupta.
- **Resultado Observado:** Sucesso. O primeiro nó que tenta se comunicar com P4 após a falha recebe uma IOException e imediatamente inicia uma eleição. Os logs mostram as mensagens de ELECTION e OK sendo trocadas. P3, sendo o nó com o maior ID restante, não recebe respostas OK e se declara o novo coordenador. Em seguida, P1 e P2 recebem a mensagem VICTORY e se reconectam com sucesso a P3. O sistema retoma a operação normal, agora sob

a nova liderança. A implementação de retentativas de conexão se mostrou crucial para evitar "tempestades de eleições".

7. Análise Crítica e Quantitativa

7.1. Complexidade de Mensagens

- **Operação Normal (Algoritmo Centralizado):** A complexidade é constante, O(1), requerendo sempre 3 mensagens por acesso à seção crítica (Requisição, Permissão, Liberação).
- **Recuperação de Falha (Algoritmo Bully):** No pior caso, onde o nó com o menor ID inicia a eleição, a complexidade é da ordem de $O(n^2)$, onde 'n' é o número de processos. Embora mais custoso, este processo só ocorre em situações excepcionais de falha.

7.2. Vantagens da Solução Implementada

- **Alta Disponibilidade:** A implementação do algoritmo de eleição de líder elimina o ponto único de falha, tornando o sistema significativamente mais robusto.

- **Modularidade e Clareza:** A refatoração do código para usar um ServicoCoordenador que pode ser instanciado por qualquer nó, junto com uma classe Logger dedicada, tornou o código limpo, organizado e fácil de depurar.

- **Simulação Realista:** O Simulador provou ser uma ferramenta eficaz para orquestrar e validar um cenário de falha complexo de forma controlada e repetível.

7.3. Limitações e Melhorias Futuras

- **Condições de Corrida na Eleição:** Embora o Algoritmo Bully tenha funcionado, ele é suscetível a condições de corrida se as mensagens de eleição ou os timeouts não forem cuidadosamente gerenciados. Uma melhoria futura seria substituir o Bully por um algoritmo de consenso mais robusto, como o Raft ou o Paxos, que fornecem garantias matemáticas de que apenas um líder será eleito.

- **Ambiente de Teste:** A simulação em uma única máquina não reflete latências de rede reais. A melhoria mais impactante seria containerizar cada nó usando Docker, permitindo simular falhas e atrasos de rede de forma muito mais realista.

8. Conclusão

Este projeto demonstrou com sucesso a concepção e implementação de um sistema distribuído de controle colaborativo que não apenas gerencia o acesso concorrente a um recurso, mas também é capaz de se recuperar autonomamente de falhas críticas. A combinação do algoritmo Centralizado para exclusão mútua com Relógios de Lamport para ordenação provou ser eficiente, enquanto a integração do Algoritmo Bully para eleição de líder e a persistência com checkpoints garantiram a alta disponibilidade e a durabilidade dos dados.

9. Apêndice A: Estrutura de Arquivos do Projeto

- **controlecolaborativo/**
  - **Simulador.java:** Classe principal que inicia a simulação e simula a falha do coordenador.
  - **comum/:** Pacote com classes compartilhadas.
    - **Documento.java:** Representa o recurso crítico compartilhado.
    - **Mensagem.java:** Define a estrutura de comunicação entre os nós.
    - **PedidoAcesso.java:** Objeto usado na fila de prioridade do coordenador.
    - **Logger.java:** Classe utilitária para formatação e coloração dos logs.
  - **no/:** Pacote contendo a lógica principal dos processos.
    - **No.java:** Classe que representa um nó, contendo a lógica de cliente, de servidor de eleição e a capacidade de se tornar coordenador.
  - **coordenador/:** Pacote com a lógica do serviço do coordenador.
    - **ServicoCoordenador.java:** Classe Runnable que encapsula toda a lógica do coordenador.
    - **TratadorNo.java:** Thread que lida com a conexão de um nó específico ao coordenador.