Uma infraestrutura para desenvolvimento de aplicações distribuídas baseada em minitransações

Leandro Ferro Luzia

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciências da Computação

Orientador: Prof. Dr. Francisco C. R. Reverbel

São Paulo, Junho de 2012

Uma infraestrutura para desenvolvimento de aplicações distribuídas baseada em minitransações

Esta é a versão original da dissertação elaborada pelo candidato Leandro Ferro Luzia, tal como submetida à Comissão Julgadora.

Resumo

LUZIA, L. F. **Uma infraestrutura para desenvolvimento de aplicações distribuídas baseada em minitransações**. 2012. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

A proposta deste trabalho é implementar uma infraestrutura para sistemas distribuídos que ofereça uma abstração de estado compartilhado entre as máquinas na forma de uma repositório de dados utilizando minitransações para garantir a atomicidade na execução de grupos de operações sobre esse repositório. As minitransações são uma modificação do protocolo de efetivação em duas fases em que todas as operações que compõem a transação são enviadas de uma só vez, diminuindo o custo de comunicação entre as máquinas do sistema. Com o uso da primitiva de minitransação os desenvolvedores podem projetar sistemas distribuídos baseando o compartilhamento de estado entre as máquinas em estruturas de dados, e não na troca explícita de mensagens. As máquinas terão à disposição um repositório de dados que pode crescer de forma a acomodar grandes quantidades de dados e que permite que aplicações tenham sempre acesso a dados consistentes. Assim, esperamos que o desenvolvimento da aplicação distribuída seja mais simples e ajude o desenvolvedor a focar nas necessidades reais da aplicação.

Palavras-chave: minitransação, transação, banco de dado, sistemas distribuídos.

Abstract

LUZIA, L. F. **An infrastructure for developing distributed applications based in minitransactions**. 2010. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2012.

We propose in this work a infrastructure for distributed systems that allows state sharing abstraction among machines as a data repository using minitransactions to ensure atomicity when executing groups of operations over this repository. Minitransactions are a modification in two phase commit protocol such that all transaction operations are sent in one network round trip, reducing the overhead of communication. By using the minitransaction primitive developers can design distributed systems in which the state sharing is based in the usage of data structures and not explicit message exchange. The machines will have access to a data repository that can grow to serve large amounts of data and allow applications to access consistent data. This way, we expect that the distributed application development will become simpler and help developers to focus in the real needs of the applications.

Keywords: minitransaction, transaction, database, distributed systems.

Sumário

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Há diversos motivos para construirmos uma aplicação de forma distribuída, como por exemplo o compartilhamento de recursos, tolerância a falhas e escalabilidade. Embora de grande importância, desenvolver um sistema distribuído com essas características pode ser difícil. Múltiplos fatores fazem com que o desenvolvimento de aplicações distribuídas exija um esforço adicional em relação ao esforço necessário para construir sistemas convencionais. Dentre esses fatores podemos mencionar diferentes plataformas de hardware, diferentes sistemas operacionais, comunicação não síncrona entre as máquinas, e falhas e conhecimento parcial do sistema.

Um sistema distribuído é uma coleção de dispositivos computacionais individuais que podem se comunicar uns com os outros [TvS06, AW04]. Essa definição engloba uma gama de sistemas computacionais atuais, desde placas de circutos integrados contendo diversos processadores até a *Internet*. Os sistemas distribuídos a que este trabalho se refere são compostos por diversos computadores interligados por uma rede de comunicação. Cada computador nesses sistemas possui acesso somente ao seu próprio sistema de armazenamento e a única forma de compartilharem informação é por meio da troca de dados pela rede de comunicação.

Implementar o compartilhamento de estado da aplicação utilizando a troca de dados na rede não é trivial, em especial quando os dados possuem restrições semânticas que precisam ser mantidas e validadas. Se considerarmos um sistema bancário em que as contas dos usuários estão distribuídas entre diversas máquinas e que temos uma solicitação de transferência entre contas que estão em duas máquinas diferentes, é esperado que essa transferência subtraia uma certa quantia da conta de origem e adicione essa mesma quantia na conta de destino, sem alterar o valor total das contas envolvidas. Se a máquina da conta de destino falhar, a quantia subtraída da conta de origem deve ser reposta.

O problema descrito acima exige que as operações efetuadas em cada máquina sejam consideradas como uma única operação lógica, ou uma transação. Para satisfazer essa exigência pode ser utilizado o protocolo de efetivação em duas fases (*two-phase commit* ou *2PC*). O *2PC* coordena a efetivação de operações executadas em diversas máquinas, garantido que essa efetivação só ocorra se houver um consenso entre as máquinas participantes da transação. Com a utilização do *2PC* são necessárias duas rodadas adicionais de comunicação entre as máquinas, aumentando o tempo e a complexidade de uma transação.

Neste trabalho propomos a utilização de minitransações para construir uma infraestrutura para o desenvolvimento de aplicações distribuídas. Uma minitransação é uma primitiva que aglutina as operações de uma transação na primeira fase do *2PC*, permitindo que as operações sejam executadas e efetivadas com apenas duas rodadas de comunicação entre as máquinas. Essa abordagem diminui o escopo em que as minitransações podem ser utilizadas, mas oferece ao desenvolvedor uma alternativa menos custosa para a execução de transações em um ambiente distribuído.

1.1 Objetivo

A proposta deste trabalho é implementar uma infraestrutura para sistemas distribuídos que ofereça uma abstração de estado compartilhado entre as máquinas na forma de um repositório de dados utilizando minitransações para garantir a atomicidade na execução de grupos de operações sobre esse repositório. Ao invés de trocarem mensagens explicitamente, as máquinas verão um repositório que pode crescer de forma a acomodar grandes quantidades de dados e que permite que todas as máquinas tenham sempre acesso a dados consistentes. Assim, esperamos que o desenvolvimento das aplicações distribuídas seja mais simples e ajude os desenvolvedores a focar nas necessidades reais das aplicações.

Essa infraestrutura será composta por máquinas que formam o repositório de dados e implementam o protocolo de minitransações. A infraestrutura oferecerá uma interface de acesso no formato chave-valor, e a comunicação entre ela e as aplicações será baseada em um protocolo simples da camada de aplicação da pilha de protocolos da *Internet* (*Transmission Control Protocol/Internet Protocol* ou *TCP/IP*).

1.2 Organização do texto

O capítulo 2 apresenta uma motivação para o uso de transações, sua utilização em um ambiente distribuído e a necessidade de um protocolo de efetivação nesse contexto, detalhando o *2PC* e o conceito original de minitransação. O capítulo 3 descreve como pretendemos implementar a infraestrutura e detalha a arquitetura da infraestrutura e os algoritmos e protocolos de execução das minitransações. O capítulo 4 apresenta o andamento do trabalho e as expectativas em relação a datas para o término do trabalho.

Capítulo 2

Conceitos

Este capítulo apresenta a motivação para o uso de transações em aplicações (2.1), ilustrando os exemplos com alguns algoritmos simples. Naturalmente estendemos o conceito de transação para envolver mais de uma máquina, o que nos leva às transações distribuídas e o problema de efetivar uma transação desse tipo (2.2). É descrito o protocolo de efetivação em duas fases, de ampla utilização, e por fim é apresentado o conceito de minitransação (2.3), uma extensão do protocolo de efetivação em duas fases que oferece melhor performance e escalabilidade ao mesmo tempo que garante atomicidade de operações em transações distribuídas.

2.1 Transações

Aplicações executam operações, de variados tipos e para diversas finalidades, como somar dois números, ler uma tecla digitada do teclado ou enviar um *email* através da rede. Vamos considerar por exemplo o Algoritmo 2.1, que implementa a transferência de valores entre duas contas, origem e destino. Digamos que as funções *Ler* e *Escrever* implementam as operações de leitura e escrita em um gerenciador de recursos que armazene os dados das contas. Essas operações são executadas imediatamente e abortam a execução do programa caso algum erro ocorra.

O algoritmo irá obter uma referência às contas, verificar se o saldo da conta de origem é suficiente, subtrair o valor da conta de origem, somar esse mesmo valor na conta de destino e escrever os novos valores nos recursos correspondentes. Se um erro ocorrer ao executar a escrita do novo valor na conta de destino os dados ficarão inconsistentes, pois o valor terá sido retirado da conta de origem ($Escrever(O, V_O - V)$ já aconteceu), mas não terá sido adicionado à conta de destino ($Escrever(D, V_D + V)$ falhou).

Algoritmo 2.1: Transferência de valores

```
1  início
2      O ← Recurso referente à conta de origem
3      D ← Recurso referente à conta de destino
4      V ← Valor a ser transferido
5      V_O ← Ler(O)
6      se V_O >= V então
7          V_D ← Ler(D)
8          Escrever(O, V_O − V)
9          Escrever(D, V_D + V)
10     senão
11         Imprimir(Saldo insulficiente)
12     fim
13 fim
```

Como alternativa poderíamos fazer com que as funções *Ler* e *Escrever* não abortem o programa caso algum erro ocorra, e que tivéssemos acesso a uma função, $HouveErro()$, que pode ser usada para checar se a última operação *Ler* ou *Escrever* falhou. Assim, poderíamos implementar uma nova versão do algoritmo para transferência (Algoritmo 2.2), efetuando novas operações para desfazer alterações no estado do sistema.

Algoritmo 2.2: Transferência de valores - tratamento de erros

```
1  início
2      O ← Recurso referente à conta de origem
3      D ← Recurso referente à conta de destino
4      V ← Valor a ser transferido
5      V_O ← Ler(O)
6      se V_O >= V então
7          V_D ← Ler(D)
8          Escrever(O, V_O − V)
9          se HouveErro() então
10             Imprimir(ERRO - não foi possível debitar valor)
11         senão
12             Escrever(D, V_D + V)
13             se HouveErro() então
14                 Escrever(O, V_O + V)
15                 se HouveErro() então
16                     Imprimir(ERRO - dados ficarão inconsistentes!)
17                 fim
18             fim
19         fim
20     senão
21         Imprimir(Saldo insulficiente)
22     fim
23 fim
```

O Algoritmo 2.2 demonstra duas coisas: mesmo que todos os erros sejam tratados, ainda é possível que os dados fiquem em um estado inconsistente (linha 16); e que as operações executadas pela aplicação não são isoladas, mas fazem parte de uma operação lógica mais abrangente, a transferência de valores, que só pode ocorrer por completo caso todas as operações que a constituem sejam finalizadas corretamente. Essa operação lógica constituída por um conjunto de operações sobre os recursos do sistema é chamada de **transação**. O uso mais clássico e difundido de transações é feito na área de banco de dados, em que uma transação é a unidade de execução de operações, composta por uma sequência de comandos de leitura e escrita de dados [GMUW08, RG00].

O uso de transações para o desenvolvimento de aplicativos facilita a maneira como o aplicativo pode ser implementado. Por exemplo, digamos que a transferência de valores do Algoritmo 2.1 possa ser agora implementada utilizando um gerenciador de recursos que suporte o agrupamento de operações em uma transação, como no Algoritmo 2.3.

Nesse algoritmo introduzimos três novas funções: $IniciarTransacao$, para criar uma nova transação, retornando um identificador para a transação criada; $Efetivar(T)$, que sinaliza que as alterações efetuadas pela transação T podem ser realmente executadas; e $Abortar(T)$, que indica que a transação foi cancelada e que alterações por ela efetuadas não surtirão efeito. As operações *Ler* e *Escrever* foram alteradas para referenciar a transação da qual fazem parte.

O uso da transação permitiu que o formato o Algoritmo 2.3 ficasse muito parecido com o do Algoritmo 2.1. As únicas diferenças são relacionadas à criação da transação, para demarcar o início das operações que devem ser executadas de forma atômica, e os momentos da efetivação ou cancelamento. No caso de efetivação, nenhum erro ocorreu e o gerenciador de recursos irá efetivar

Algoritmo 2.3: Transferência de valores - uso de transações

```
1  início
2      O ← Recurso referente à conta de origem
3      D ← Recurso referente à conta de destino
4      V ← Valor a ser transferido
5      T ← IniciarTransacao()
6      V_O ← Ler(T, O)
7      se V_O >= V então
8          V_D ← Ler(T, D)
9          Escrever(T, O, V_O − V)
10         Escrever(T, D, V_D + V)
11         Efetivar(T)
12     senão
13         Imprimir(Saldo insulficiente)
14         Abortar(T)
15     fim
16 fim
```

as alterações efetuadas pelas operações da transação. Caso a conta de origem não possua saldo suficiente, a transação será cancelada.

Nesse algoritmo o comportamento das funções *Ler* e *Escrever* é parecido com o comportamento apresentado no Algoritmo 2.1: se ocorrer um erro, o programa é terminado. Agora, porém, antes de terminar o programa, as funções irão abortar a transação à qual estão relacionadas, mantendo os dados inalterados.

2.2 Transações distribuídas e o protocolo de efetivação em duas fases

Uma transação distribuída é uma transação que engloba operações que executam em diversos gerenciadores de recursos na forma de subtransações, subordinadas à transação, e que é finalizada por uma requisição para efetivar ou abortar a transação [GL06]. Por exemplo, vamos considerar novamente o problema da transferência entre contas, como descrito na Seção 2.1, porém agora as contas estão em dois gerenciadores de recursos transacionais distintos (G_O e G_D), como na Figura 2.1.

A efetivação das subtransações geradas em G_O e G_D precisa ser efetuada atomicamente e precisa levar em conta que mais coisas podem dar errado em relação ao que podia acontecer no caso não distribuído, como por exemplo: as conexões de rede podem falhar, fazendo com que a aplicação consiga se comunicar somente com um dos gerenciadores; mensagens na rede podem ser duplicadas ou perdidas, exigindo que haja um tratamento especial para esses casos.

Para lidar com esses problemas e garantir que a efetivação seja atômica, é necessário o uso de um protocolo que irá coordenar a efetivação da transação distribuída e garantir que essa só será considerada efetivada se todas as subtransações forem efetivadas. O mais conhecido desses protocolos é o protocolo de efetivação em duas fases (*two-phase commit protocol* ou *2PC*).

A idéia desse protocolo é simples e utilizada a bastante tempo [Gra78]: verificar se todos os gerenciadores de recurso envolvidos em uma transação estão aptos a efetivar suas respectivas subtransações. Se estiverem, a transação será efetivada. Se algum gerenciador não puder efetivar, por qualquer motivo, a transação será abortada. O *2PC* protocolo necessita de um gerenciador que atue como o **coordenador** da transação, responsável por gerenciar a execução do protocolo e tomar a decisão sobre efetivar ou abortar a transação comunicando-se com os outros **participantes**, que executam as subtransações associadas à transação. O coordenador pode ser um participante da

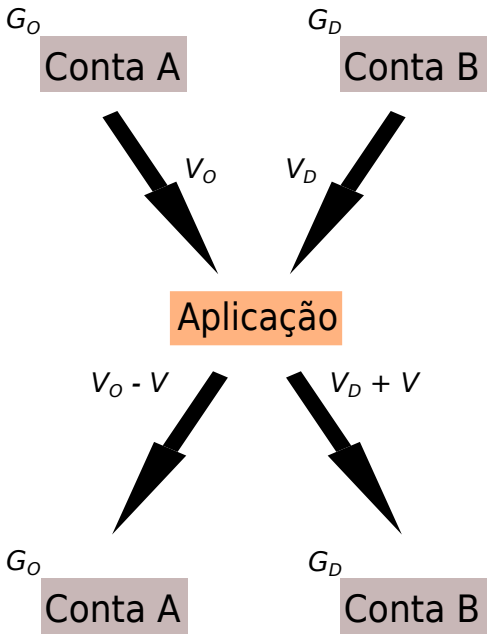

Figura 2.1: *Esquematização de uma transação distribuída*

transação também.

O protocolo é dividido em duas fases. Na primeira fase o coordenador solicita que os participantes enviem seus votos, indicando se estão aptos a efetivar a subtransação executada. Os votos são coletados e o coordenador decide se a transação pode ser efetivada (caso todos os participantes tenham votado de acordo) ou se deve ser abortada (caso algum participante tenha votado para não efetivar a transação). Decidido o resultado da votação, o coordenador efetua a segunda fase, em que os participantes são notificados do resultado da votação.

De forma mais detalhada, o *2PC* pode ser descrito pelos algoritmos 2.4, 2.5 e 2.6. O Algoritmo 2.4 descreve as ações do coordenador ao ser notificado que o protocolo deve iniciar. Os algoritmos 2.5 e 2.6 descrevem as ações executadas pelos participantes ao receberem uma solicitação de votação (primeira fase) e o resultado da votação (segunda fase), respectivamente.

Uma estrutura de dados essencial na execução do *2PC* é o registro de operações, ou *log*. Ele é responsável por registrar as decisões do coordenador e dos participantes, e é gravado localmente em cada gerenciador de recursos envolvido na transação para evitar que essas informações sejam perdidas caso ocorra alguma falha. No Algoritmo 2.4 log_c representa o *log* do coordenador e nos algoritmos 2.5 e 2.6 log_i é o *log* do participante p_i executando o algoritmo em questão.

A função *Adicionar* representa a operação de adicionar um elemento ao final do *log*. A função $Enviar(d, m)$ representa o envio de uma mensagem m para um destinatário d pela rede de comunicação, e a função $Receber(r)$ representa o recebimento de uma mensagem de um remetente r. Uma visualização da execução do algoritmo pode ser vista nas figuras 2.2 e 2.3.

A premissa que norteia o protocolo é que qualquer gerenciador envolvido na transação pode decidir abortá-la de forma unilateral, exigindo assim unanimidade na decisão pela efetivação da transação. As mensagens enviadas durante a execução do protocolo indicam uma decisão do remetente, e para garantir que essa decisão sobreviva a falhas no gerenciador que enviou a mensagem, os dados do *log* são gravados em um meio de armazenamento estável, como um disco rígido, antes da mensagem ser enviada. A transação T é considerada oficialmente efetivada (ou abortada) no momento que o registro de $(EFETIVAR, T)$ (ou $(ABORTAR, T)$) do *log* do coordenador for escrito para a área de armazenamento estável da máquina. Falhas posteriores não podem mudar a decisão do coordenador registrada em seu *log* e gravada em disco.

É importante notar que esse protocolo não define como as operações da transação devem ser

Algoritmo 2.4: Coordenador 2PC

```
1  início
2      Adicionar(log_c, (PREPARAR, T))
3      para todo p_i ∈ Participantes faça
4          Enviar(p_i, (PREPARAR, T))
5      fim
6      d ← EFETIVAR
7      para todo p_i ∈ Participantes faça
8          v ← Receber(p_i)
9          se v = ABORTAR então
10             d ← ABORTAR
11         fim
12     fim
13     Adicionar(log_c, (d, T))
14     para todo p_i ∈ Participantes faça
15         Enviar(p_i, (d, T))
16     fim
17 fim
```

Algoritmo 2.5: Votação 2PC - p_i recebe $(PREPARAR, T)$ de c

```
1 início
2     v ← Decidir(T)
3     Adicionar(log_i, (v, T))
4     Enviar(c, v)
5 fim
```

Algoritmo 2.6: Notificação 2PC - p_i recebe (d, T) de c

```
1 início
2     se d = EFETIVAR então
3         Efetivar(T)
4         Adicionar(log_i, (EFETIVAR, T))
5     senão
6         Abortar(T)
7         Adicionar(log_i, (ABORTAR, T))
8     fim
9 fim
```

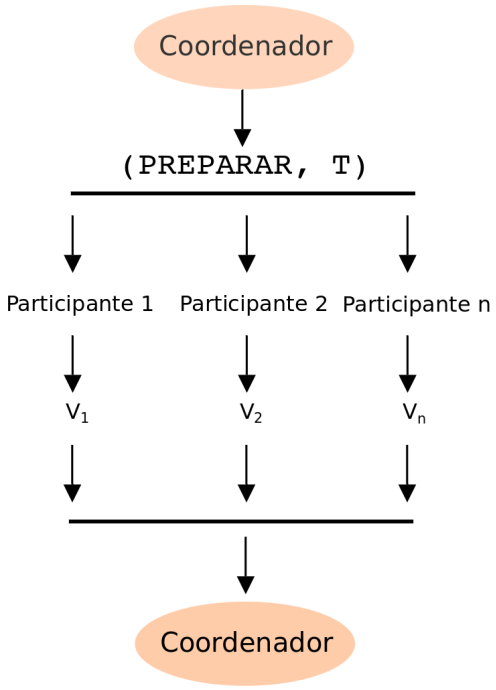

Figura 2.2: *Primeira fase 2PC - O coordenador inicia a votação e os participantes respondem com seus votos V_i*

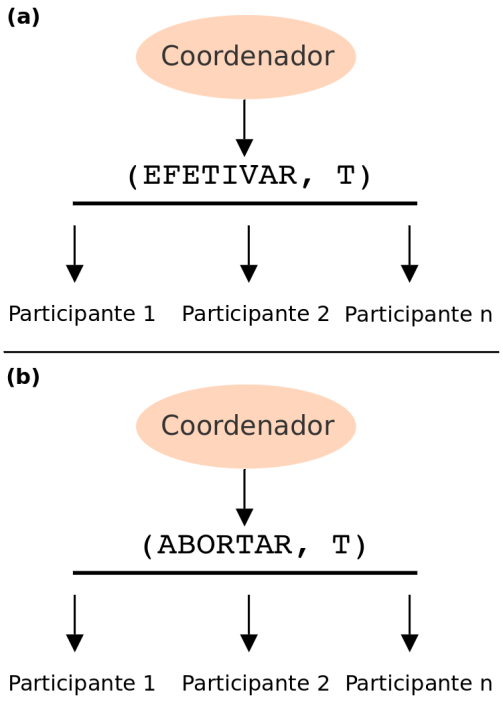

Figura 2.3: *Segunda fase 2PC- O coordenador apura os votos e notifica os participantes. Em (a) os participantes são notificados de uma efetivação. Em (b), a transação foi abortada.*

executadas nas máquinas participantes, mas sim como a efetivação da transação deve proceder. As subtransações ocorrem em cada participante antes que o protocolo de efetivação inicie. Portanto, a utilização desse protocolo aumenta o número de mensagens e consequentemente o tempo e o esforço necessários para que uma transação seja executada. Algumas otimizações podem ocorrer, como no caso de transações que efetuam somente operações de leitura ou de transações que envolvam dados em somente uma máquina do sistema mas, de forma geral, a execução do *2PC* é custoso [GL06].

2.3 Minitransações

Minitransação é uma primitiva que permite que as operações de uma transação sejam executadas durante o protocolo de efetivação. Esse protocolo de efetivação é uma modificação do *2PC*, e oferece um mecanismo simples para ler e alterar dados de forma condicional em um ambiente distribuído garantindo atomicidade na execução das operações [AMS+07]. Dessa forma, o número de mensagens e o tempo de execução da transação são reduzidos.

Porém, o uso das minitransações impõe certas restrições em relação ao que pode ser feito, diminuindo sua aplicabilidade. Por exemplo, o condicionamento das operações de leitura e escrita é baseado somente em comparações de igualdade, e a única forma da aplicação cliente abortar a minitransação é se essa comparação falhar, como será detalhado. Com as minitransações originais não é possível efetuar a transferência entre contas distribuídas como no Algoritmo 2.3, pois a comparação de maior ou igual (>=) não pode ser feita. A nossa infraestrutura irá permitir que essa e outras comparações sejam feitas.

Em 2.3.1 é apresentado como o protocolo *2PC* pode ser usado como ponto de partida para otimizações e para a obtenção do protocolo de minitransações e em 2.3.2 definimos formalmente o conceito de minitransação.

2.3.1 Otimização do *2PC*

A decisão pela efetivação ou cancelamento de uma transação distribuída depende tanto de aspectos operacionais, relacionados ao ambiente de execução, quanto de aspectos semânticos, específicos do domínio da aplicação. A falha na execução de uma subtransação em alguma máquina do ambiente inviabiliza a efetivação da transação como um todo, e por isso o coordenador é forçado a cancelar a transação. Esse tipo de falha operacional não está ligada ao domínio da aplicação, mas sim ao ambiente em que essa aplicação está rodando e está, portanto, fora do controle do coordenador, que pode somente cancelar a transação e, opcionalmente, tentar executá-la novamente. Por outro lado, o aspecto semântico envolvido na decisão pela efetivação ou cancelamento da transação é específico de cada aplicação e depende, direta ou indiretamente, dos dados do sistema.

Considerando novamente o sistema bancário e a operação de transferência de uma determinada quantia entre uma conta de origem e de destino, a transferência só pode ocorrer se o saldo na conta de origem da transferência for maior ou igual à quantia a ser transferida. Essa checagem deve ser feita pela aplicação após a leitura da informação da máquina que armazena os dados da conta de origem, e a decisão pelo cancelamento ou não da transação fica subordinada à semântica dada aos dados do sistema. Isso exige que uma requisição de leitura seja feita e uma resposta seja enviada, para só então a aplicação decidir se vai efetivar e então, após a execução do restante da transação, iniciar a primeira fase do protocolo *2PC* (votação).

Podemos ver que, do ponto de vista semântico, as operações que influenciam na decisão pela possível efetivação ou pelo cancelamento da transação são operações de leitura. As operações de escrita não influenciam nessa decisão, a não ser pelo ponto de vista operacional, ou seja, se ocorrer realmente um erro na operação de escrita. Assim, se tivermos uma transação cuja última ação não afete a decisão do coordenador sobre efetivar a transação, podemos embutir essa última ação na mensagem de votação da primeira fase do protocolo de efetivação.

O aspecto semântico da transação em relação aos dados pode ser tratado nos participantes também, e não somente no coordenador, caso o participante saiba como o coordenador irá utilizar

Figura 2.4: *Estrutura de uma minitransação*

o dado para fazer sua decisão sobre efetivar ou cancelar a transação. Se isso for possível, podemos então embutir também operações de leitura que influenciam a decisão do coordenador no protocolo de efetivação e fazer o participante adequar seu voto à maneira como o coordenador faria ao analisar o dado retornado.

As minitransações surgem no contexto em que todas as operações de uma subtransação podem ser embutidas dentro do protocolo de efetivação, utilizando somente as trocas de mensagens que ocorreriam no protocolo de efetivação, após a execução dos comandos. Para que isso possa ocorrer, o protocolo *2PC* precisa ser alterado.

2.3.2 Definição

Uma minitransação é composta por três conjuntos: itens de comparação, itens de leitura e itens de escrita, como pode ser visto na figura 2.4. Todos os itens possuem uma referência a qual dado deve ser utilizado ($ID_{A...D}$), e os itens de comparação e escrita incluem também os dados ($DADO_{1...4}$) que serão comparados com ou substituirão os dados armazenados [AMS+07].

Formalmente, uma minitransação pode ser vista como uma tupla na forma (I, C, L, E). I é o identificador da minitransação, gerado pelo criador da minitransação. C é o conjunto de itens de comparação, L é o conjunto de itens para leitura e E é o conjunto de itens para escrita. Os elementos de L são identificadores de dados, e o domínio de seus valores é o conjunto de identificadores armazenados na máquina participante. C e E possuem elementos que podem ser representados como tuplas no formato $(Id, Dado)$, em que Id é o identificador do dado e $Dado$ é o valor para se comparar com o valor identificado por Id ou para substituí-lo.

Sendo uma extensão do *2PC*, o protocolo de minitransações possui também um coordenador responsável por iniciar e gerenciar a execução do protocolo entre os participantes da transação. O protocolo de minitransações é composto também por duas fases, mas agora a primeira fase passa a ser uma fase de execução, em que as minitransações são executadas em cada participante e seus votos são enviados para o coordenador. O coordenador coleta os votos de todos os participantes e, como no protocolo original, irá decidir por efetivar a transação somente se os votos forem unânimes. Na segunda fase os participantes são notificados da decisão do coordenador e devem atuar de acordo, efetivando as operações da minitransação ou desfazendo as seus ações.

Um operador especial, $Id[X]$, representa o conjunto formado pelo elemento Id de todas as tuplas no formato $(Id, Dado)$ do conjunto X. Esse operador permite obter o conjunto D de todos os identificadores utilizados pela minitransação: $D \leftarrow L \cup Id[C] \cup Id[E]$. Cada elemento de D fica sob responsabilidade de um participante p específico da minitransação, e $\forall d \in D$ o participante responsável pelo identificador d é definido por $Participante(d)$, uma função que indica que p é a possível localização de d entre os participantes, uma vez que o identificador d pode ainda não ter sido inserido em p.

O conjunto $Participantes$ é formado por todos os participantes da minitransação. Cada participante $P_j \in Participantes$ possui um conjunto K_j de identificadores sob sua responsabilidade, e para cada participante o coordenador constrói uma nova minitransação $M_j = (I_t, C_j, L_j, E_j)$ tal que:

- $I_t \leftarrow IdentificadorUnico(I)$
- $\forall i_l \in L_j, i_l \in K_j$;
- $\forall i_c \in Id[C_j], i_c \in K_j$; e
- $Id[E_j] \supseteq K_j$

O identificador I_t é unicamente definido e associado ao identificador I por meio da função $IdentificadorUnico$. Como o identificador I é fornecido pela aplicação cliente, nada garante que ele seja único, e por isso utilizamos $IdentificadorUnico$ para criar um novo identificador globalmente único associado à I. Os identificadores de E_j são tratados como um superconjunto de K_j pois a operação de escrita pode inserir novos dados no sistema, e não somente alterar dados que já existem.

Cada M_j é enviada ao respectivo P_j pelo coordenador, que irá esperar pela execução e resposta de cada participante. Ao receber uma minitransação, cada participante irá tentar ler os dados identificados por $Id[C_j]$ e compará-los (comparação de igualdade) com os respectivos $Dado[C_j]$. Caso a comparação não seja bem sucedida, esse participante nem tentará ler ou modificar os dados em L_j e E_j, e responderá para o coordenador com um voto $ABORTAR$. A Figura 2.5 ilustra essa primeira fase, que combina a primeira fase do *2PC* com a execução das operações da transação.

Se a comparação for bem sucedida para todos os elementos de $Id[C_j]$, então o participante irá ler os dados identificados por L_J e agrupá-los no conjunto de resposta R_j. Os dados $Dado[E_j]$ identificados por $Id[E_j]$ serão inseridos no conjunto de dados ou irão alterar algum dado já existente. Na verdade, a operação de inserir ou alterar um dado é registrada no *log* do participante. O participante envia então um voto $EFETIVAR$ para o coordenador, junto com o conjunto R_j.

Ao coletar todas as respostas, o coordenador irá apurar os votos de cada participante. Para cada resposta $EFETIVAR$ o coordenador agrega os respectivos R_j em um conjunto de respostas R. Se uma das respostas for $ABORTAR$, a transação precisa ser abortada, notificando os participantes do cancelamento, e a aplicação cliente precisa ser notificada desse erro, como podemos ver na Figura 2.6 **b**. Se não houve nenhum voto $ABORTAR$, o coordenador decide então por efetivar a transação, enviando o conjunto R para a aplicação cliente e notificando os participantes da efetivação, como ilustrado na Figura 2.6 **a**.

Quando os participantes são notificados do cancelamento da minitransação, as operações de escrita da minitransação presentes no *log* do participante não serão efetuadas, deixando os dados inalterados, e será registrado no *log* que a minitransação foi abortada. Se a notificação for de efetivação, as operações de escrita do *log* serão efetuadas, alterando dados existentes ou inserindo novos dados no participante, e será registrado no *log* que a minitransação foi efetivada. O *log* será gravado em disco e a minitransação será considerada oficialmente efetivada ou abortada no momento que esse registro do *log* estiver gravado em disco.

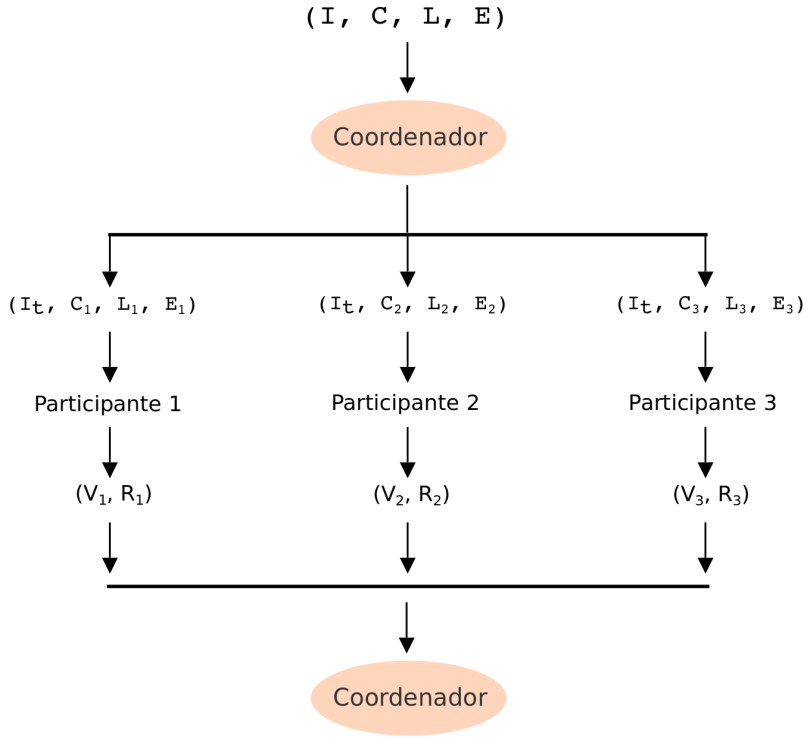

Figura 2.5: *Fase de execução de uma minitransação*

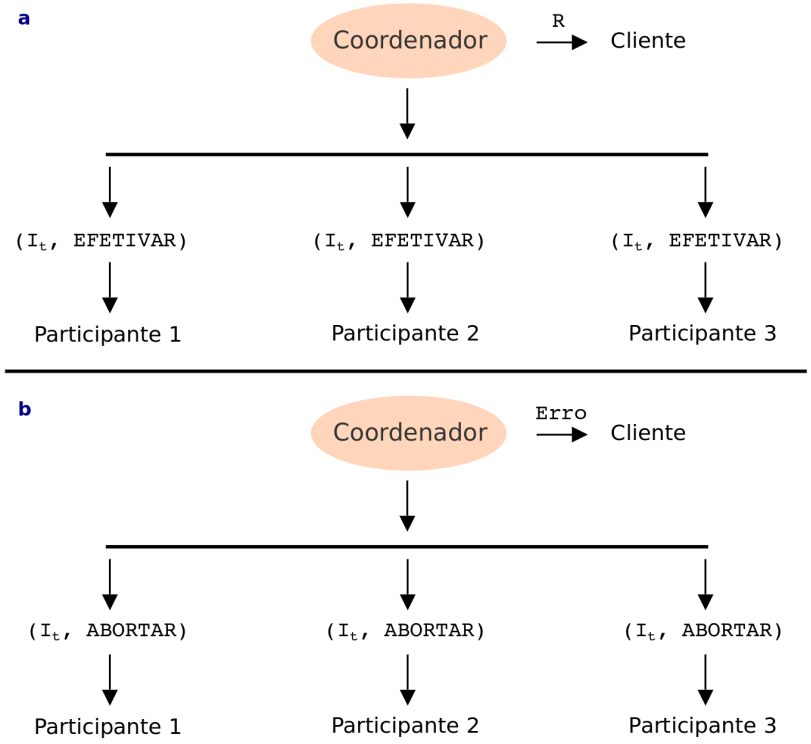

Figura 2.6: *Fase de notificação de uma minitransação*

Capítulo 3

A infraestrutura

Neste capítulo apresentamos uma visão detalhada da infraestrutura desenvolvida. A arquitetura e a interface de acesso serão discutidas na Seção 3.1. A Seção 3.2 descreve as estruturas de dados e algoritmos utilizados para implementar a execução das minitransações. Na Seção 3.3 são apresentados os trabalhos que compartilham características de nossa infraestrutura, assim como suas diferenças em relação à nossa infraestrutura.

No trabalho descrito em [AMS+07] as minitransações podem ser compostas por apenas três tipos de operações: Leitura, Escrita e Comparação por igualdade. Essa estrutura permite que uma grande quantidade de transações possam ser descritas como uma minitransação, mas exclui uma série de outros tipos de transação. Por esse motivo, a nossa infraestrutura estenderá o conceito original de minitransação e permitirá que operações sejam definidas e utilizadas pelos usuários, de maneira semelhante à utilização de um procedimento armazenado (*stored procedure*) de um banco de dados convencional. Esses comandos serão chamados daqui em diante de Comandos de Extensão ou Extensões.

Assim, esperamos flexibilizar a utilização e aumentar o número de cenários em que a infraestrutura pode ser utilizada. Portanto, a estrutura das minitransações descrita neste capítulo difere um pouco da estrutura descrita em 2.3.2. As operações de leitura e escrita originais serão mantidas pois possuem uma semântica diferenciada. A operação de comparação original (comparação por igualdade) será implementada como uma extensão em nossa infraestrutura.

3.1 Arquitetura e interface de acesso

A infraestrutura desenvolvida neste trabalho é um sistema composto por dois tipos de componentes de maior importância: execução e acesso. O componente de execução é responsável por prover acesso aos dados através da execução das operações especificadas nas minitransações, e é chamado de *nó de memória* ou somente *nó*. O componente de acesso é o ponto de comunicação entre a aplicação e a infraestrutura, atuando como o coordenador das transações e isolando a aplicação cliente dos detalhes sobre a distribuição dos dados entre os nós de memória e será chamado *coordenador*. O terceiro componente é um componente de gerenciamento, utilizado para operar o sistema manualmente, chamado *gerenciador*.

Os nós de memória formam o núcleo da infraestrutura e serão implementados de tal forma que não haverá nenhum compartilhamento de informação entre eles, constituindo uma arquitetura não compartilhada (*shared nothing architecture* [Sto86]) e permitindo que a infraestrutura possa escalar para acomodar uma grande quantidade de dados e atender um número crescente de usuários.

A Figura 3.1 ilustra a relação entre os componentes do sistema. As aplicações cliente acessam somente os coordenadores. Nesse exemplo há quatro nós de memória (*Nó 1* até *Nó 4*) que armazenam os dados e executam as minitransações. O gerenciador é um componente secundário, que permite a execução de procedimentos de gerenciamento sobre os nós de memória, como recuperação de falhas ou a criação dos comandos de extensão.

A interface de acesso oferecida aos clientes da infraestrutura é a de uma tabela associativa

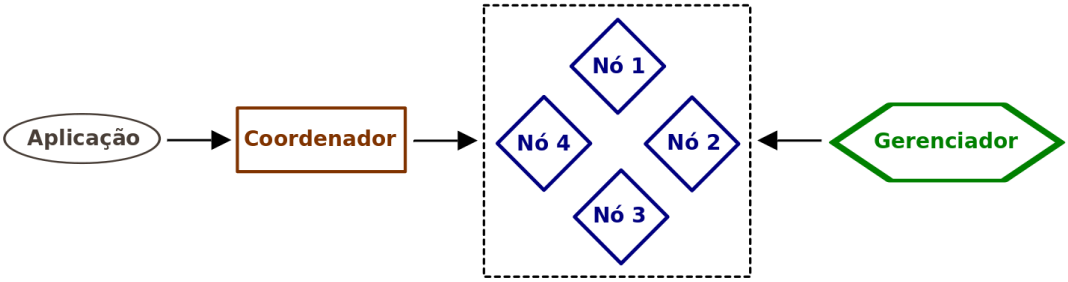

Figura 3.1: *Visão geral da arquitetura da infraestrutura*

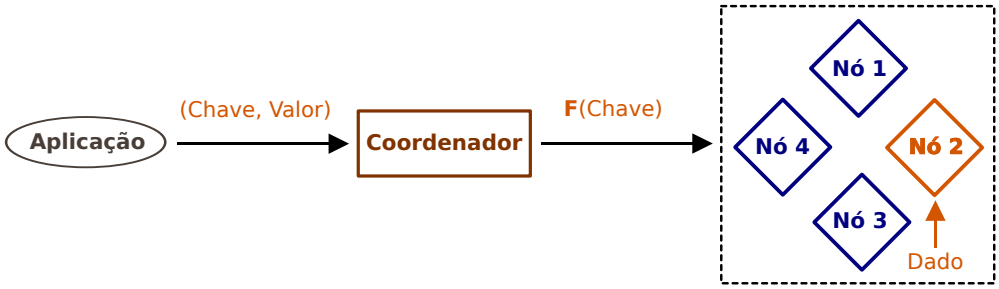

Figura 3.2: *Mapeamento entre a chave e o nó de memória responsável por seu armazenamento*

chave-valor: os nós de memória armazenam uma sequência de bytes (valor) nomeados por uma outra sequência de bytes (chave). Cada chave é única e o número de bytes utilizados para armazenar o valor é, teoricamente, ilimitado. Os coordenadores são responsáveis por efetuar o mapeamento entre as chaves e os nós de memória que armazenam esses dados, utilizando uma função de espalhamento consistente (*consistent hashing* [KLL+97]) aplicada à chave acessada. Como pode ser visto na Figura 3.2, o coordenador irá aplicar uma função de espalhamento consistente F à $Chave$, resultando em um número que indica que $Valor$ deve ser armazenado no nó de memória 2.

As aplicações se comunicam com os coordenadores por meio de um protocolo simples da camada de aplicação da pilha TCP/IP. Esse protocolo é textual, como o protocolo HTTP (*Hypertext Transmission Protocol*), utilizando o protocolo TCP para o transporte de dados.

Os comandos do protocolo são separados em linhas e são compostos por uma letra, que indica a ação desse comando, e um conjunto de parâmetros, chamados campos, que fornecem informações para a execução da ação. Os comandos que iniciam uma minitransação e que contém a resposta da execução dessa minitransação podem ser compostos por subcomandos, e por isso são utilizadas chaves ({}) para delimitar o seu escopo. Os campos de cada comando são separados por um caracter de espaço em branco. Os comandos utilizados na comunicação entre coordenador e aplicação estão listados a seguir:

- **M** *TamanhoIdentificador Identificador*: indica a criação ou execução de uma minitransação com identificador *Identificador*. *TamanhoIdentificador* é o número de bytes do campo *Identificador*. É composto por subcomandos, como leitura ou escrita, e o seu escopo é delimitado por chaves ({ e }).
- **L** *TamanhoChave Chave*: indica a leitura dos bytes associados à chave *Chave*. *TamanhoChave* indica o número de bytes de *Chave*.
- **E** *TamanhoChave Chave TamanhoDado Dado*: indica a escrita de *Dado*, cujo número de bytes é *TamanhoDado*, no valor associado à *Chave*.
- **C** *IdentificadorComando TamanhoChave Chave TamanhoParâmetro1 Parâmetro1 TamanhoParâmetro2 Parâmetro2*, ...: indica a execução de um comando de extensão identificado por

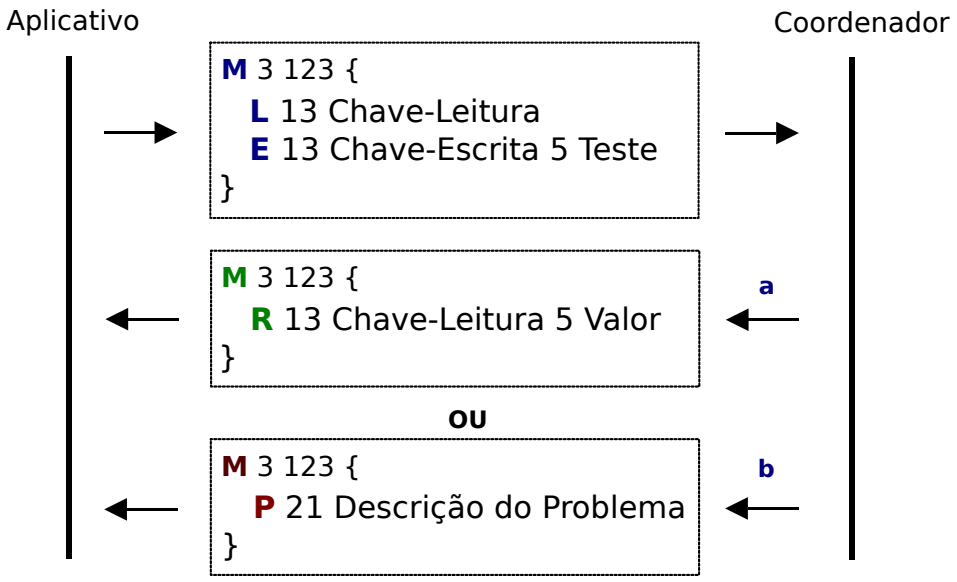

Figura 3.3: *Exemplo de execução de uma minitransação por meio do protocolo textual utilizado entre as aplicações e o coordenador com os dois possíveis resultados: (a) A transação é bem sucedida e foi efetivada e (b) A transação fo cancelada*

IdentificadorComando sobre a chave *Chave*. Os parâmetros adicionais são repassados ao comando, que vai interpretá-los apropriadamente. Como *IdentificadorComando* é composto por 4 bytes, a infraestrutura pode conter até 2^{32} comandos de extensão. Estes comandos não tem a permissão de escrita em nenhum dado do sistema.

- **R** *TamanhoChave Chave TamanhoDado Dado*: indica a resposta de um comando de leitura dos bytes associados à *Chave*.
- **P** *TamanhoDescrição Descrição*: indica que ocorreu um erro e que a minitransação não pode ser executada. *Descrição* contém informações sobre o erro ocorrido.

A Figura 3.3 ilustra a interação entre a aplicação e o coordenador para a execução de uma minitransação cujo identificador é *123* (**M** *3 123*)). Essa minitransação possui dois subcomandos: a leitura da chave "Chave-Leitura" (**L** *13 Chave-Leitura*) e a escrita de "Teste" na chave "Chave-Escrita" (**E** *13 Chave-Escrita 5 Teste*). Como as chaves são formadas por uma sequência de bytes arbitrária, é necessário que seja especificado quantos bytes fazem parte da chave. Pelo mesmo motivo é necessário também informar o número de bytes do valor do campo "Dado".

No exemplo podemos ver os dois cenários de resposta. Em **a**, a execução da minitransação é bem sucedida e o participante retorna um subcomando de resposta de leitura (**R** *13 Chave-Leitura 5 Valor*). No cenário **b**, a execução não foi bem sucedida e o coordenador notifica a aplicação através do comando **P**, descrevendo o erro ocorrido.

O protocolo de execução de minitransações usado internamente entre os coordenadores e os nós de memória é composto por um superconjunto dos comandos do protocolo descrito anteriormente. Os comandos adicionais são:

- **S**: Esse comando sem parâmetros indica que a minitransação foi executada com sucesso no nó de memória, e essa minitransação está pronta para ser efetivada.
- **N** *TamanhoDescrição Descrição*: Esse comando indica que a execução da minitransação não foi bem sucedida e a transação precisa ser cancelada. O parâmetro *Descrição* contém informações relevantes sobre o problema encontrado ao executar a minitransação.
- **F**: É o comando enviado pelo coordenador para os participantes para informar que a transação foi bem sucedida em todos os nós e que deve ser efetivada.

- **A**: Indica que a transação foi abortada e que os nós de memória devem descartar as possíveis alterações executadas pela minitransação.

Os dois primeiros comandos, **S** e **N**, são as possíveis respostas dos nós de memória à fase de execução do protocolo indicando, respectivamente, que a transação pode ou não pode ser efetivada. No caso negativo, o nó de memória irá informar no parâmetro *Descrição* o motivo. Os comandos **F** e **A** são enviados do coordenador para os nós de memória para indicar que a transação foi ou não efetivada, ficando a cargo dos nós de memória os procedimentos para efetivar ou cancelar as alterações efetuadas pelas minitransações.

3.2 Algoritmos e estruturas de dados

Iremos considerar que os dados da infraestrutura estão distribuídos em n nós de memória $P_1, P_2, \ldots, P_n$. Cada P_i armazena um conjunto de informações D_i. O conjunto de dados $D = \bigcup_{i=1}^{n} D_i$ representa então o estado do sistema, e os elementos de cada D_i são disjuntos. Existem duas estruturas de dados principais utilizadas pelos nós de memória e mais duas estruturas adicionais para o controle de concorrência e de falhas:

- **Tabela de dados**: Cada conjunto D_i é representado por uma tabela associativa, que pode ser vista como uma função que associa um identificador a um valor, representados respectivamente por $Id(X)$ e $Dado(X)$, onde $X \in D_i$. Essa tabela suporta as seguintes operações:

  1. $Ler(Id(X))$, que retorna $Dado(X)$ ou um valor especial ($NULO$) caso $X \notin D_i$
  2. $Escrever(Id(X), V)$, que atribui a $Dado(X)$ o valor V caso $X \in D_i$ ou que insere X em D_i ($D_i \leftarrow D_i \cup \{X\}$) caso contrário.

- **Registro de operações** (*log*): É uma lista ordenada de operações de escrita sobre os dados. Os elementos dessa lista estão no formato ($Id(X)$, $Dado(X)$, $Dado'(X)$) onde $Id(X)$ e $Dado(X)$ são o identificador e o valor do elemento X armazenado na tabela de dados, e $Dado'(X)$ é o novo valor que deve ser associado a $Dado(X)$ caso a minitransação I_t seja efetivada. Essa estrutura permite que as operações de transações efetivadas sejam aplicadas corretamente e permite que o sistema se recupere de falhas. As seguintes operações são suportadas pelo *log*:

  1. $Adicionar(l, t, o)$, que adiciona uma operação o da transação t ao final da lista de operações identificada por l.
  2. $Selecionar(l, t)$, retorna uma lista de operações registradas em l referentes à minitransação t.
  3. $Gravar(l)$, que força que o registro de operações identificado por l seja gravado em alguma forma de armazenamento persistente, como um disco rígido.

- **Tabela de minitransações**: É um conjunto que registra a situação de uma determinada minitransação identificada por I_t. As possíveis situações de uma minitransação podem ser $AGUARDANDO$, $EFETIVADA$ ou $CANCELADA$.

- **Tabela de travas**: Associa a uma chave c dois possíveis valores, chamados travas: $LEITURA$ e $ESCRITA$. A função $Travar(I_t, L, E)$ tenta associar aos elementos de L a trava $LEITURA$ e aos elementos de E a trava $ESCRITA$, retornando $TRAVOU$ caso seja possível, ou $FALHOU$ caso contrário. A função $Liberar(I_t)$ retira todas as travas associadas à transação I_t.

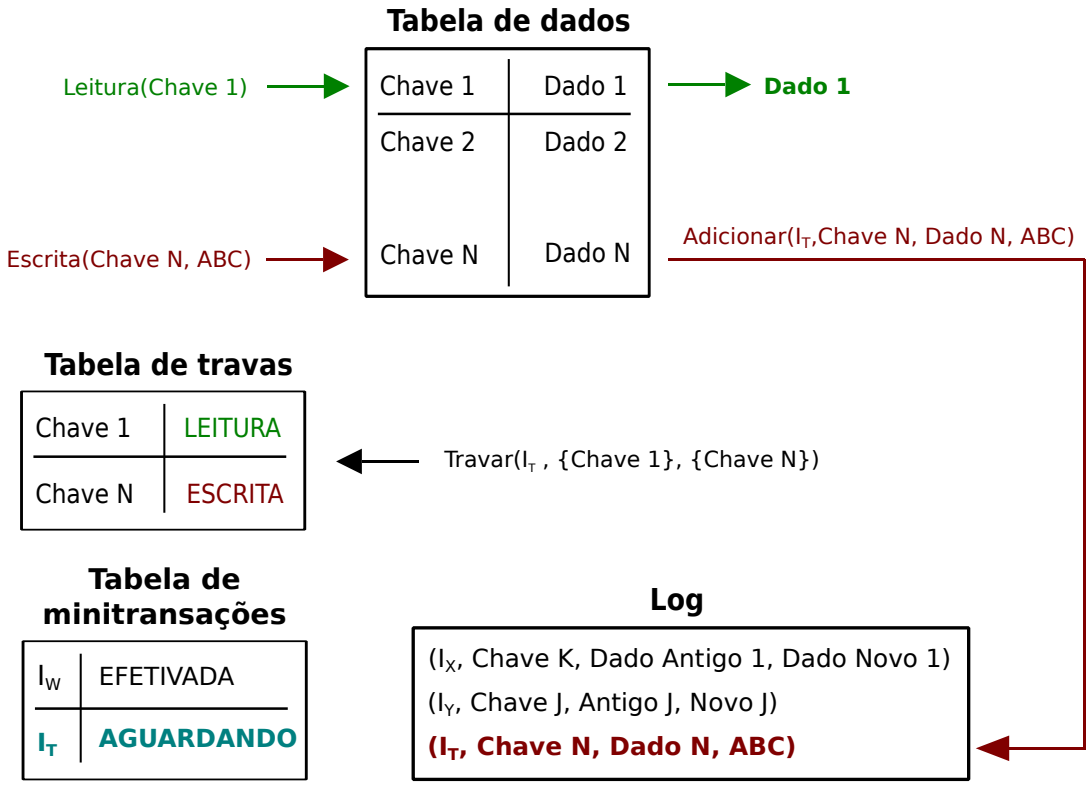

Figura 3.4: *Operações sobre as estruturas e a relação entre elas*

Um exemplo das operações sobre as estruturas e a relação entre elas pode ser visto na figura 3.4. Nessa figura, são executadas duas operações: uma leitura de "Chave 1", resultando em "Dado 1", e uma escrita em "Chave N", resultando na alteração do valor associado à essa chave para "ABC". Antes que essas duas operações possam ocorrer, são obtidas duas travas—uma de leitura para a chave "Chave 1" e uma de escrita para a chave "Chave N". O registro $(I_T, Chave\ N, Dado\ N, ABC)$ é adicionado ao *log* de operações, e o elemento $\{I_t, AGUARDANDO\}$ é incluído na tabela de minitransações.

O comportamento do coordenador é descrito pelo Algoritmo 3.1. I_t é o identificador da transação, e é unicamente determinado pelo coordenador, relacionado com o identificador I enviado pelo cliente. C é o conjunto de comandos de extensão — esses comandos indicarão ao nó de memória se a minitransação pode ser efetivada ou não, de forma análoga à operação de comparaçao das minitransações originais. L é o conjunto de itens a serem lidos e E é o conjunto de dados a serem atualizados ou inseridos. $L_j, Id[C_j]$ e $Id[E_j]$ são os conjuntos de identificadores da transação que estão armazenados em P_j. $Op(c)$ indica a operação associada ao comando $c \in C_j$ e $Parametros(c)$ representa a lista de parâmetros de c. Cada participante p_j mantém um *log* de operações log_{p_j}, uma tabela de minitransações mt_{p_j} e uma tabela de travas $travas_{p_j}$.

Ao receber a minitransação, cada nó de memória segue o procedimento descrito no Algoritmo 3.2, primeiro adicionando a nova minitransação no conjunto de minitransações. Depois o nó de memória tenta obter todas as travas para acessar os dados especificados na minitransação—essa estratégia é usada para evitar problemas de esperas cíclicas (*deadlocks*) entre diferentes minitransações. Uma resposta especial, $FALHOU\_AO\_TRAVAR$ é enviada caso não tenha sido possível obter alguma trava. Se conseguiu obter todas as travas, o nó de memória vai executar os comandos de extensão, e se algum deles retornar $ERRO$, a minitransação será abortada.

Após a execução, os participantes respondem à primeira fase do protocolo para o coordenador, que analisa cada resposta para descobrir o resultado da transação. Caso uma das respostas seja $FALHOU\_AO\_TRAVAR$, o coordenador deve abortar a transação em todos os participantes e iniciar novamente a execução. Uma resposta $ABORTAR$ indica que algum comando de extensão

Algoritmo 3.1: Coordenador - recebe uma transação no formato (I, C, L, E)

```
1  início
2      repetir ← false
3      repita
4          I_t ← GerarIdentificador(I)
5          para todo p_j ∈ Participantes(C, L, E) faça
6              Enviar(p_j, (I_t, C_j, L_j, E_j))
7          fim
8          d ← EFETIVAR
9          R ← {}
10         para todo p_j ∈ Participantes faça
11             (v_j, r_j) ← Receber(p_j)
12             se v_j = FALHOU_AO_TRAVAR então
13                 repetir ← true
14                 d ← ABORTAR
15             senão se v_j = ABORTAR então
16                 repetir ← false
17                 d ← ABORTAR
18                 P ← Descrição do Problema
19             senão
20                 repetir ← false
21                 R ← r_j ∪ R
22             fim
23         fim
24         para todo p_j ∈ Participantes faça
25             Enviar(p_j, (d, I_t))
26         fim
27     até repetir = false
28     se d ≠ ABORTAR então
29         Enviar(Cliente, (I, R))
30     senão
31         Enviar(Cliente, (I, P))
32     fim
33 fim
```

Algoritmo 3.2: Execução - p_j recebe (I_t, C_j, L_j, E_j) de c

```
1  início
2      se Travar(C_j ∪ L_j, E_j) = FALHOU então
3          Enviar(c, (FALHOU_AO_TRAVAR))
4      senão se (I_t, CANCELADA) ∈ mt_{p_j} então
5          Enviar(c, (ABORTAR,"Cancelado pelo gerenciador"))
6      senão
7          mt_{p_j} ← (I_t, AGUARDANDO) ∪ mt_{p_j}
8          d ← EFETIVAR
9          para todo c ∈ C_j faça
10             se Executar(Op(c), Id(c), Parametros(c)) = Erro então
11                 d ← ABORTAR
12             fim
13         fim
14         R_l ← {}
15         se d = EFETIVAR então
16             para todo l ∈ L_j faça
17                 R_l ← Ler(Id(l)) ∪ R_l
18             fim
19             para todo e ∈ E_j faça
20                 Adicionar(log_{p_j}, I_t, (Id(e), Ler(Id(e)), Dado(e)))
21             fim
22             Gravar(log_{p_j})
23         fim
24         Liberar(I_t)
25         Enviar(c, (d, R_l))
26     fim
27 fim
```

Algoritmo 3.3: Confirmação - p_j recebe (d, I_t) de c

```
1  início
2      se d = EFETIVAR então
3          para todo (Id(X), Dado(X), Dado'(X)) ∈ Selecionar(log_{p_j}, I_t) faça
4              Escrever(Id(X), Dado'(X))
5          fim
6          mt_{p_j} ← (I_t, EFETIVADA) ∪ mt_{p_j}
7      senão
8          mt_{p_j} ← (I_t, CANCELADA) ∪ mt_{p_j}
9      fim
10     Adicionar(log_{p_j}, I_t, d)
11     Gravar(log_{p_j})
12     Liberar(I_t)
13 fim
```

não foi bem sucedido, a transação é cancelada em todos os nós e o protocolo termina. Se todos os nós responderem com $EFETIVAR$, as respostas aos comandos de leitura são agregadas no conjunto R e retornadas para a aplicação cliente.

A decisão sobre a efetivação ou cancelamento da transação é comunicada a cada nó de memória, que segue o procedimento descrito no Algoritmo 3.3. Caso a decisão tenha sido para efetivar a minitransação, o participante irá vasculhar o seu *log* por operações relacionadas à transação I_t e irá executar essas transações, alterando a tabela de dados caso operações de escrita tenham sido especificadas na minitransação. A decisão sobre a efetivação ou cancelamento da minitransação é registrada no *log* de cada participante e esse *log* é gravado em disco.

Diferentemente do que ocorre no *2PC*, o coordenador nunca escreve em um *log*. Como os componentes de acesso estarão em geral nas mesmas máquinas que as aplicações cliente, e essas máquinas são consideradas menos confiáveis que as máquinas em que os nós de memória estarão, essa abordagem permite que a execução de uma minitransação possa continuar caso o coordenador fique indisponível. O procedimento para recuperação em casos de falhas nesse cenário é mais complexo do que o caso centralizado do 2PC, em que a recuperação de falhas é feita a partir do *log* do coordenador.

A recuperação de falhas na nossa infraestrutura será efetuada pelo nó de gerenciamento. Esse procedimento pode ser ativado tanto manualmente quanto de forma automática. O nó de gerenciamento consulta os nós de memória sobre minitransações que estejam na situação $AGUARDANDO$ por mais que um determinado período de tempo e, caso haja alguma, o nó irá solicitar que essa minitransação seja abortada. Nesse caso, o participante irá mudar a situação da minitransação para $CANCELADA$, e o algoritmo 3.2 irá validar essa situação (linha 4). Nos algoritmos descritos é assumido que a operação de união de conjuntos considerará a unicidade do elemento baseado no identificador da transação. Por exemplo, se o conjunto mt_{p_j} já possuir um elemento (k, S), sendo k um identificador e S uma situação, se tentarmos efetuar a operação $mt_{p_j} \cup (k, S_1)$, essa dupla irá substituir a dupla (k, S) em mt_{p_j}.

O controle de concorrência efetuado pelos nós de memória visa oferecer uma forma de sequenciamento entre as operações de minitransações concorrentes que, de forma similar às transações sequencíaveis de bancos de dados convencionais [RG00], tem o efeito de garantir que as operações de uma minitransação não interfiram nas operações de outras minitransações. Para efetuar a leitura da chave c, o nó de memória deve primeiro tentar associar uma trava $LEITURA$ à chave c na tabela de travas. Essa associação só pode ocorrer se não houver nenhuma trava do tipo $ESCRITA$ associada a essa chave. A associação de uma trava $ESCRITA$ só pode ocorrer se não houver nenhuma trava $LEITURA$ ou $ESCRITA$ associada à chave c. Esse tipo de trava é chamada de trava leitor-escritor (*read-write lock* [Ste99]), e permite que várias transações possam ler um determinado dado ao mesmo tempo mas somente uma pode escrever informações nesse dado, e somente se nenhuma outra transação estiver lendo ou escrevendo.

3.3 Trabalhos relacionados

O conceito de minitransações é introduzido como base para a construção de *Sinfonia* [AMS+07], um sistema cujo foco é prover a base para o desenvolvimento de sistemas distribuídos de baixo nível, como sistemas de arquivos distribuídos, gerenciadores de travas ou serviços de comunicação de grupos de computadores, enquanto que o objetivo deste trabalho é utilizar as minitransações como base para a construção de uma infraestrutura que facilite o desenvolvimento de aplicações distribuídas de alto nível, como sistemas de comércio eletrônico ou redes sociais.

Sinfonia é composto também por nós de memória que armazenam os dados, mas a interface exposta por esses componentes é diferente: cada nó de memória mantém uma sequência de bytes puros, organizados em um espaço de endereçamento linear, sem nenhuma estrutura. Cada nó de memória possui um espaço de endereçamento separado de forma que os dados são referenciados por um par que especifica o nó de memória e o endereço deste dado. Em nossa infraestrutura, o acesso é feito através de uma única chave, que é um conjunto de bytes que faça sentido para a aplicação.

Ao invés de utilizar um coordenador separado como em nossa infraestrutura, *Sinfonia* disponibiliza uma biblioteca que é compilada junto ao aplicativo cliente, e nesse caso o aplicativo cliente é o coordenador. Em nossa infraestrutura, o coordenador permite isolar a aplicação de detalhes da distribuição dos dados entre os nós de memória, e pode ser acessado pela rede. Acreditamos que esta abordagem seja mais efetiva que a abordagem que utiliza uma biblioteca, pois qualquer linguagem que permita a utilização da pilha TCP/IP poderá utilizar nossa infraestrutura, ao passo que se disponibilizarmos uma biblioteca, somente as linguagens para as quais uma biblioteca for escrita poderão usufruir da infraestrutra.

[PP11] apresenta um sistema de armazenamento baseado em minitransações tolerante a falhas bizantinas. Em sistemas que toleram somente componentes com falhas simples (*fail-stop components*), é assumido que um componente pode estar em dois estados: ativo e inativo. Se estiver em um estado ativo, o componente se comportará de acordo com a especificação do sistema. Se estiver inativo, o componente simplesmente para de interagir com o sistema. Essa é uma maneira simples e um tanto simplificada de lidar com falhas no sistema, mas é a forma utilizada por diversos sistemas, entre eles *Sinfonia* e a nossa infraestrutura. A maneira mais geral de lidar com falhas é através de falhas bizantinas [LSP82]. Em sistemas que lidam com esse tipo de falha, um componente ativo pode se comportar de forma incorreta, enviando mensagens com conteúdo aleatório (correto ou incorreto), ou não enviando mensagem nenhuma. O tratamento de falhas bizantinas não será discutido neste trabalho.

Como nosso sistema, gerenciadores de bancos de dados relacionais também permitem a execução de transações, como Oracle e MySQL. O tipo de transação oferecida por esses sistemas é normalmente chamada de ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [RG00]. Essas transações podem ser usadas em cenários nos quais não conseguimos utilizar as minitransações, sendo portanto muito mais gerais. Porém, devido às propriedades que devem garantir, o uso dessas transações não permite escalar o banco de dados para um grande número de máquinas, algo que as minitransações permitem.

Existem diversos sistemas que visam o armazenamento escalável de informações e a disponibilização de serviços para facilitar o desenvolvimento de sistemas distribuídos, visando em geral a utilização em aplicações de internet de larga escala. Entre eles, podemos citar Bigtable [CDG+08], Dynamo [DHJ+07], ZooKeeper [HKJR10], PNUTS [CRS+08], entre outros.

Bigtable é o sistema de armazenamento distribuído do Google que oferece uma abstração de um mapa ordenado, multi-dimensional, esparço e distribuído. Dynamo, da Amazon, é um sistema de armazenamento chave-valor que visa oferecer alta disponibilidade à aplicações. Esses dois sistemas permitem que dados sejam particionados e replicados para obter melhor performance e disponibilidade, mas permitem que diferentes máquinas possam ter versões diferentes de uma mesma informação.

ZooKeeper é um sistema que provê serviços de coordenação e sincronização para a construção de sistemas distribuídos que utiliza o algoritmo Paxos [Lam98] para garantir consistência entre as operações. Como o objetivo do ZooKeeper é permitir a coordenação entre componentes de um sistema distribuído, sua capacidade de armazenamento é limitada (em um *megabyte*), e portanto não é utilizável como um repositório de dados.

PNUTS é o serviço de armazenamento de dados do Yahoo! que garante que todas as réplicas de um determinado dado armazenado executam as mesmas alterações, na mesma ordem. PNUTS permite a existência de várias versões de um dado, e oferece uma primitiva *test-and-set-write*, que efetua uma escrita de dados somente se uma determinada versão do dado for encontrada, semelhante ao mecanismo de comparação das minitransações.

Por último, Hazelcast [Haz] e Akka [Akk] são ferramentas para o desenvolvimento de sistemas distribuídos que visam eliminar a necessidade de comunicação explícita entre os participantes do sistema, oferecendo abstrações como estruturas de dados distribuídas ou memória transacional. Hazelcast oferece a programas rodando na *JVM Java* implementações distribuídas das coleções da biblioteca padrão (*Collection, Set, List e Map*). Akka permite a utilização de memória transacional por *software* (*software transactional memory* ou *STM* [ST97]), uma abordagem que emprega o

conceito de transação discutido neste trabalho em operações de leitura e escrita na memória principal do computador.

Capítulo 4

Cronograma

Os próximos passos no trabalho serão:

- Desenvolvimento da infraestrutura: deverão ser implementados os nós de memória, coordenador e gerenciador. Período: Julho/2012 - Janeiro/2013.
- Desenvolvimento das aplicações de teste: aplicações simples que permitam explorar e analisar a utilização e a performance da infraestrutura. Período: Novembro/2012 - Março/2013.
- Testes das aplicações: execução dos testes, coleta e análise dos dados obtidos. Período: Março/2013 - Julho/2013.

Referências Bibliográficas

[Akk] Akka. Akka. *http://akka.io/*. 21

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch e Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. Em *In SOSP*, páginas 159–174. ACM, 2007. 9, 10, 13, 20

[AW04] Hagit Attiya e Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004. 1

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes e Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, Junho 2008. 21

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver e Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Agosto 2008. 21

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall e Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Outubro 2007. 21

[GL06] Jim Gray e Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Março 2006. 5, 9

[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman e Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edição, 2008. 4

[Gra78] Jim Gray. Notes on data base operating systems. Em R Bayer, R Graham e G Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, páginas 393–481. Springer Berlin / Heidelberg, 1978. 5

[Haz] Hazelcast. Hazelcast. *http://www.hazelcast.com/*. 21

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira e Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. Em *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, páginas 11–11, Berkeley, CA, USA, 2010. USENIX Association. 21

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine e Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. Em *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, páginas 654–663, New York, NY, USA, 1997. ACM. 14

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, Maio 1998. 21

[LSP82] Leslie Lamport, Robert Shostak e Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, Julho 1982. 21

[PP11] Ricardo Padilha e Fernando Pedone. Scalable byzantine fault-tolerant storage. Em *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, páginas 171–175, Washington, DC, USA, 2011. IEEE Computer Society. 21

[RG00] Raghu Ramakrishnan e Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edição, 2000. 4, 20, 21

[ST97] Nir Shavit e Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. 10.1007/s004460050028. 21

[Ste99] W.R. Stevens. *UNIX Network Programming: Interprocess communications*. The Unix Networking Reference Series , Vol 2. Prentice Hall PTR, 1999. 20

[Sto86] Michael Stonebraker. The case for shared nothing. *Data Engineering 9 1*, 9(1):4–9, 1986. 13

[TvS06] Andrew S. Tanenbaum e Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 1