LORENZO FRANCESCO GIOVANNI GINO MARIA RIDOLFI

PROPOSTA DE ARQUITETURA TRANSACIONAL DISTRIBUÍDA

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, 28 de setembro de 1995

Lorenzo Francesco Giovanni Gino Maria Ridolfi

PROPOSTA DE ARQUITETURA TRANSACIONAL DISTRIBUÍDA

Dissertação apresentada ao Departamento de Informática da PUC-RIO como parte dos requisitos para a obtenção do título de Mestre em Ciências em Informática.

Orientador: Marco Antônio Casanova

Co-orientador: Luiz Fernando Gomes Soares

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 28 de setembro de 1995 (Data da defesa)

À Paixão,

em todas as suas formas,

sentidos e interpretações

AGRADECIMENTOS

- ao Professor Marco Antônio Casanova, pelos quase cinco anos de ótima convivência, pela excelente orientação, confiança e perceverança na conclusão deste trabalho
- ao Professor Luiz Fernando Gomes Soares, pela co-orientação, onde colaborou com inúmeras idéias e discussões, enriquecendo o conteúdo deste trabalho
- aos meus pais, que simplesmente me deram tudo e todas as condições para que este trabalho existisse
- aos meus grandes amigos Sérgio Colcher, José Alberto Vasi Werner e Marcelo Pereira Chaves Duarte que muito me apoioaram e incentivaram durante todo o tempo deste trabalho, sempre dispostos a ajudar e me ouvir em todas as ocasiões que precisei
- à Paula Nóbrega Bentolila, por todo carinho, incentivo e compreensão, sobretudo nos momentos finais, quando por muitas vezes estive ausente e não pude dar toda a atenção e carinho que alguém tão especial merece
- à Luiz Tucherman, por toda sua amizade, confiança, oportunidades, inúmeros conselhos, críticas, idéias, e por sempre ter insistido que concluísse este trabalho
- aos Amigos do antológico chopp das quarta-feiras, responsáveis por incontáveis momentos agradáveis que jamais serão esquecidos
- à todas pessoas com quem convivi no Centro Científico Rio da IBM Brasil, onde este trabalho teve início, por todo o apoio e amizade em todo este tempo de convívio
- ao meu amigo Leonardo Goldfeld que, mesmo extremamente atarefado, encontrou tempo para me enviar prontamente um importante paper para este trabalho
- a C. Mohan e Christos Polyzois pela colaboração no envio de diversas publicações, inclusive inéditas, que muito colaboraram para a elucidação de diversas questões deste trabalho
- ao SENAI/DN, em particular ao CIET, que cedeu parte de minha jornada de trabalho à conclusão desta dissertação
- a CAPES pelo apoio financeiro

RESUMO

O presente trabalho propõe uma arquitetura para o processamento distribuído de transações com mecanismos adicionais, que garantem maior confiabilidade no acesso a informações armazenadas. Esta arquitetura é uma variação do modelo *shared nothing*, que introduz a divisão das funções de um servidor em *servidores de armazenamento* e *servidores de transação*, visando uma maior flexibilidade na alocação dos dados e uma melhor tolerância a falhas de servidores. A arquitetura conta ainda com a presença de uma estrutura central de comunicação de dados que evita a ocorrência de falhas de difícil contorno pela arquitetura distribuída, o que torna os mecanismos de segurança da arquitetura mais simples e eficientes. A confiabilidade desta arquitetura é mantida por um mecanismo de recuperação e replicação, baseado na abordagem por *cópia primária*, que prevê suporte diferenciado às aplicações que exijam maior segurança na manipulação de dados. Este mecanismo garante um alto grau de confiabilidade às transações que manipulam dados críticos, enquanto atua com economia de recursos em transações que não possuam requisitos especiais de segurança. Adicionalmente, o mecanismo de replicação oferece suporte a *pseudo-transações*, uma transação com requisitos mínimos de confiabilidade, que garantem um melhor aproveitamento dos dados replicados, usualmente pouco aproveitados na replicação por cópia primária, oferecendo uma alternativa de baixo custo de processamento para aplicações que toleram a presença de dados com ligeiras inconsistências.

ABSTRACT

This work presents an enhanced architecture for distributed transaction processing with additional mechanisms to assure reliability in information storage. It is based on the shared-nothing model, and divides server functions into two kinds of nodes, called *transaction servers* and *storage servers*, aiming flexibility in data allocation and fault-tolerance. It also uses a central data communication structure which avoids occurence of faults with complex fixing procedures. The reliability of this architecture is kept by a recovery and replication mechanism, using the primary-copy replication model, that supports transactions based on their reliability requirements. It offers higher confidence to critical data, while saving resources in ordinary transactions. Additionally, its replication and recovery mechanism introduces a concept of pseudo-transactions, a transaction with low reliability requirements used to increase the utility of the backup copies, usually poorly used in the primary-copy model. By doing this, it offers a low cost alternative for applications that tolerate a slight data inconsistency.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Hoje em dia é comum convivermos com serviços ininterruptos de terminais bancários, reserva de passagens aéreas, aprovação de despesas de cartão de crédito, entre muitos outros. Para fornecer estes serviços na qualidade e na escala exigida pela grande quantidade de usuários atuais, as empresas buscam cada vez mais apoio nos ambientes para o processamento *on-line* das transações de suas principais aplicações e de sistemas de apoio a decisão. Fatores adicionais, tais como a globalização de mercados e o aumento da concorrência, reforçam a necessidade de velocidade e continuidade no fornecimento de informação útil para clientes, fornecedores e para a tomada de decisões.

Tradicionalmente, os principais sistemas vitais de uma empresa são processados em computadores de grande porte, com custos elevados de manutenção e com opções restritas de crescimento ou expansão. De outra forma, os microcomputadores e estações de trabalho atuais possuem um grande poder de processamento e armazenamento, com custos por unidade de processamento ou por megabyte muito inferiores aos dos tradicionais computadores de grande porte. Com a disseminação da tecnologia de redes locais de alta velocidade e do modelo cliente/servidor, os microcomputadores e estações de trabalho tornaram-se uma excelente plataforma para os ambientes distribuídos voltados para o processamento *on-line* de transações e para suporte a tomada de decisões.

O modelo cliente/servidor é particularmente interessante para a construção de aplicações distribuídas para o processamento de transações. Esta abordagem consiste em dividir as funções de uma aplicação em dois tipos de componentes, *clientes* e *servidores*. Os clientes são responsáveis pela construção de requisições de serviço a serem atendidas pelos servidores, onde tais requisições se referem a operações implementadas no servidor. A função principal do servidor, desta forma, é processar os requisitos dos clientes, retornando-lhes posteriormente os resultados destas requisições.

Este modelo possui importantes propriedades para o processamento distribuído de transações:

Modularidade O modelo cliente/servidor facilita expansões na capacidade do sistema, que são realizadas através do simples acréscimo de novos servidores. Tal facilidade permite que o ambiente assuma uma formação inicial reduzida e cresça proporcionalmente a demanda por novos recursos ou serviços.

Desempenho Pela sua natureza distribuída, o modelo cliente/servidor permite que o processamento de transações seja feito em diversos processadores, através da soma de suas capacidades individuais de processamento e armazenamento. Adicionalmente, a divisão de funções do modelo permite que os servidores assumam configurações especializadas para a sua função, oferecendo melhor desempenho aos clientes.

Independência A separação das funções entre os clientes e servidores permite que os clientes se abstraiam da estrutura interna e da localização dos servidores. Na verdade, os clientes só precisam se preocupar com os nomes e os parâmetros dos serviços que necessitam, uma vez que os detalhes da implementação dos serviços é totalmente encapsulada nos servidores. A localização dos servidores, por sua vez, também não é relevante pois os clientes buscam os serviços através de identificadores abstratos e independentes, de forma independente da localização física do servidor que irá prestar o serviço.

No modelo cliente/servidor, o processamento de transações pode ser distribuído em diversos processadores, somando-se desta forma as capacidades individuais de processamento e de armazenamento de cada computador. Isto permite que os processadores operem em sinergia, oferecendo desempenho equivalente a um computador de maior porte, a um custo inferior ao da solução centralizada.

Existem diversas abordagens para o grupamento de processadores para o processamento distribuído de transações. Um método para o acoplamento de processadores é obtido através do compartilhamento de dados, onde todos os nós grupados possuem acesso direto aos dados armazenados em discos compartilhados por todos os nós. Esta abordagem é chamada de *arquitetura shared disks* [37, 24]. Outra técnica de grupamento é realizada através do particionamento dos dados entre os diversos processadores, onde nenhum disco ou memória é compartilhada. Esta forma de grupamento é conhecida como *arquitetura shared nothing* [54].

A abordagem por particionamento dos dados é considerada o consenso final entre as diversas opções para o grupamento de processadores para o processamento de transações [12]. A arquitetura *shared nothing* permite explorar massivamente o paralelismo das diversas opções de computadores comuns encontrados no mercado. A abordagem por compartilhamento de dados, por sua vez, é restrita aos mecanismos e equipamentos especializados necessários ao compartilhamento de discos por mais de um processador, sendo pouco difundida na prática.

Em um ambiente real, no qual existe a possibilidade de falha em processadores, a partição estrita de dados proposta pela arquitetura *shared nothing* pode impactar seriamente a disponibilidade do banco de dados em um ambiente massivamente paralelo. Como o banco de dados encontra-se particionado entre os diversos processadores que compõem o sistema, a ocorrência de uma falha em um único nó impede uma visão completa do banco de dados. Se considerarmos que a probabilidade de um processador falhar é ρ em um sistema massivamente paralelo com n nós - ou seja, com n tendendo a infinito - a probabilidade que todos dados estejam disponíveis é zero:

$$\lim_{n \to \infty} (1 - \rho)^n = 0 \tag{1.1}$$

Adicionalmente, na arquitetura *shared nothing*, uma transação muitas vezes atualiza dados dispersos em nós distintos. A coordenação destas atualizações exige o controle através de um protocolo de terminação, tipicamente o *two-phase commit* [22] ou alguma variação deste protocolo [35, 36]. É fato que não há nenhum protocolo de terminação que, em casos de falhas em nós ou na rede de comunicação, não bloqueie o processamento de um nó sem falhas [53, 18]. Esta situação de bloqueio torna a fração do banco de dados alocada neste nó indisponível, o que também contribui para impedir uma visão total do banco de dados.

Essas limitações exigem que os sistemas baseados nas arquiteturas *shared nothing* disponham de algum tipo de mecanismo de replicação de dados. O objetivo destes mecanismos é impedir que uma fração do banco de dados torne-se indisponível no caso de uma eventual falha no nó que a contém. Isto é feito a partir da manutenção automática de cópias dos dados em outros nós do sistema. Os mecanismos tradicionais de replicação de dados em arquiteturas distribuídas ou exigem muitos recursos para a manutenção do sincronismo entre as diversas cópias de um mesmo dado, ou não permitem o uso normal dos dados replicados, que tornam-se disponíveis apenas na ocorrência de falhas no ambiente.

1.2 Objetivo do Trabalho

O objetivo desta dissertação é propor uma arquitetura transacional distribuída baseada no modelo *shared nothing* que provê mecanismos adicionais de replicação que garantam uma maior disponibilidade aos dados armazenados. Esta segurança é obtida através do mecanismo de recuperação de falhas da arquitetura, que prevê o suporte a replicação de dados. A principal característica deste mecanismo é sua flexíbilidade, oferecendo um alto grau de segurança às aplicações críticas, e atuando com economia de recursos nas aplicações que não possuam requisitos de disponibilidade diferenciados. Adicionalmente, a arquitetura transacional distribuída oferece uma alternativa econômica para o processamento de sistemas de apoio a decisão, e outras aplicações com requisitos mínimos de consistência de informações, sobre dados replicados, permitindo um uso mais eficiente da informação redundante do ambiente.

Os nós servidores desta arquitetura são divididos em duas classes de nós: os servidores de armazenamento e os servidores de transação. Como o nome indica, os servidores de armazenamento são responsáveis pelo armazenamento dos dados, enquanto que os nós de transação se encarregariam do controle das transações distribuídas. Através desta separação de funções e da presença da estrutura central de comunicação, algumas limitações do *two-phase commit* são evitadas e há maior liberdade na divisão de funções dos processadores.

1.3 Estrutura do Trabalho

O **capítulo 2** revisa os conceitos e mecanismos presentes em sistemas de banco de dados vinculados à segurança e disponibilidade no armazenamento de informação. Descrições claras de termos como: sistemas, falhas, ações atômicas e recuperação de falhas são apresentadas, visando estabelecer definições básicas para todo o trabalho. O conceito de transação é visto em detalhes, pois é o paradigma central dos métodos de controle de concorrência e de recuperação de falhas. Estes métodos, por sua vez, são examinados em seguida. Por último, os mecanismos adicionais de segurança presentes nos sistemas descentralizados, em particular o término de transações distribuídas e as diferentes técnicas de replicação de dados, são expostos com precisão.

Os diversos estudos e comparações sobre as diferentes formas de grupamento de processadores para o processamento paralelo de transações são objeto de estudo do **capítulo 3**. Além de fornecer embasamento para a arquitetura distribuída proposta nesta dissertação como um todo, estes estudos e comparações oferecem conclusões particularmente relevantes para a escolha do modelo de replicação mais adequado a esta arquitetura. Adicionalmente, este capítulo examina o uso de relações de afinidade na alocação de processadores e dados, visando otimizar o desempenho de arquiteturas de banco de dados distribuídas.

A arquitetura transacional distribuída e seus principais componentes são introduzidos no **capítulo 4**. O uso de um elemento central de comunicação é exposto em detalhes, bem como a propriedade de *comunicação atômica* garantida com a presença deste elemento central. A comunicação atômica, na verdade, é discutida no contexto de um modelo de falhas que, além de falhas em comunicação de dados, descreve também quais os tipos de falhas que são toleradas pelos nós da arquitetura distribuída.

Este mesmo capítulo apresenta o modelo de replicação da arquitetura transacional distribuída. O modelo proposto permite a classificação das transações da arquitetura segundo as suas exigências de confiabilidade na manipulação de informação. Tal classificação é examinada em seguida, sob o aspecto das relações de compromisso entre serviços de alta disponibilidade e os que agem com economia de recursos do ambiente. A divisão de funções entre servidores de armazenamento e de transação também é vista no **capítulo 4**, discutindo-se também o papel dos servidores de reserva, do servidor de diretório e dos clientes na arquitetura transacional distribuída.

O mecanismo de recuperação de falhas da arquitetura transacional distribuída, suas estruturas de dados e principais operações são descritos em detalhes no **capítulo 5**. As estruturas de dados compreendem os registros dos *logs* dos servidores de armazenamento e transação e tabelas adicionais para o controle das transações em andamento e dos dados potencialmente inconsistentes manipulados por estas transações. O funcionamento das operações da arquitetura também é descrito neste capítulo, sendo complementando pelo pseudo-código de todas as operações localizado no **apêndice A**.

O **capítulo 6** é reservado para destacar as principais contribuições deste trabalho, relatando também as conclusões que podemos tirar do estudo de alternativas para o aumento da disponibilidade em arquiteturas distribuídas, que culminou na confecção desta dissertação. Finalmente, apresentamos algumas sugestões para trabalhos futuros, relatando possíveis extensões e melhorias à arquitetura aqui proposta.

Capítulo 2

Segurança e Disponibilidade na Manipulação de Informação Armazenada

2.1 Introdução

Um banco de dados é dito confiável quando garante a continuidade do processamento de consultas e atualizações mesmo na presença de eventuais falhas em alguns de seus componentes. A confiabilidade relaciona-se igualmente aos mecanismos que o banco de dados deve dispor para assegurar a integridade das informações que tem sob sua responsabilidade, também sob situações adversas.

Para que esta propriedade de um banco de dados seja assegurada, sua implementação deve contemplar diversos mecanismos específicos para garantir a consistência do banco de dados, bem como contar com a presença de replicação de dados sensíveis. Nos banco de dados distribuídos, tais mecanismos possuem a complexidade adicional de lidar com a coordenação dos diversos bancos de dados e com falhas na comunicação entre processadores, que não ocorrem no caso centralizado.

Neste capítulo descrevemos os conceitos e mecanismos presentes em sistemas de banco de dados que visam dar segurança à manipulação dos dados armazenados.

2.2 Falhas, Ações Atômicas e Recuperação de Falhas

O conceito de ação atômica é central para a estruturação dos mecanismos de recuperação de falhas presentes em sistemas de banco de dados. Estes mecanismos visam tornar o banco de dados consistente após a ocorrência de uma falha em um de seus componentes.

Além de discutir as premissas de um mecanismo de recuperação de falhas, esta seção apresenta definições precisas de conceitos fundamentais no contexto de um sistema de banco de dados confiável.

2.2.1 Sistemas, Estados e Falhas

Um *sistema* é um mecanismo que consiste de um conjunto de componentes e interaje com o ambiente através de estímulos, produzindo resultados específicos [45, 41]. Cada componente do sistema, por sua vez, também é um sistema *per se*, sendo chamado de *subsistema*. No contexto de um sistema, o *ambiente* de um determinado componente consiste dos outros componentes com os quais está diretamente relacionado. O *ambiente externo* de um sistema é, por sua vez, composto pelos outros sistemas com os quais o sistema interaje diretamente.

O *estado externo* de um sistema é a resposta que o sistema fornece a um estímulo de seu ambiente. O *estado interno* de um sistema é a agregação dos estados externos de todos os seus componentes. Na verdade, o estado externo de um sistema é um sub-conjunto dos estados externos de seus compontentes, sendo então uma *abstração* de seu estado interno. Os relacionamentos e interações entre os diversos componentes de um sistema são expressos através de *assertivas* sobre o seu estado interno. Dizemos que o estado interno do sistema está *internamente consistente* se este satisfaz todas as suas assertivas.

O comportamento de um sistema com o ambiente é ditado através de sua especificação [58]. A *especificação* de um sistema é definida somente em termos de seus estados externos, determinando as operações (ou estímulos) que podem ser aplicadas ao sistema, os resultados destas operações e todas as transições entre os estados externos causadas por estas operações. Ou seja, a especificação restringe o comportamento do sistema observado pelo ambiente externo.

Um *histórico de operações* é uma seqüência de operações da especificação do sistema. Dizemos que o estado do sistema é *externamente consistente*, se este corresponde ao histórico de operações que foi aplicado ao sistema. Em outras palavras, a consistência externa é o requisito que o sistema corresponda as percepções de seu ambiente externo. Finalmente, dizemos que um estado do sistema é *consistente*, se este é internamente e externamente consistente.

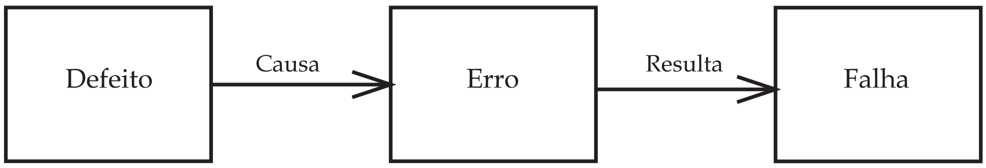

Figura 2.1: Defeitos, erros e falhas

Qualquer desvio no comportamento do sistema descrito na sua especificação é considerado uma *falha* do sistema. Dizemos que um estado interno de um sistema é um *estado errôneo* quando existem circunstâncias (no contexto da especificação do sistema) que levariam uma operação normal a produzir uma falha que não seria atribuída a um defeito subseqüente. Um *erro* em um sistema é, portanto, a parte de seu estado interno que está incorreta. A presença de qualquer erro em um estado interno de um sistema é chamada de um *defeito* no sistema. Em resumo, um defeito no sistema causa um erro que resulta numa falha do mesmo sistema.

2.2.2 Tipos de Falhas e Comportamento do Sistema

Em um ambiente real não há limites para os tipos de falhas de diferentes severidades que um sistema pode sofrer. Apesar de altamente desejável, é impossível a confecção de um sistema que tolere todos os tipos de falhas. Na verdade, um sistema tolerante a falhas suporta apenas um determinado conjunto de falhas restritas a um modelo de falhas suportado pelo sistema.

A elaboração de um modelo de falhas consiste em delimitar as classes de falhas que serão tratadas pelo sistema. Isto é, a especificação do sistema descreve o comportamento do sistema apenas nas situações de falha previstas no modelo de falhas. As classes de falhas, que não fazem parte do modelo de falhas, não estão presentes na especificação do sistema, não sendo assim toleradas pelo sistema. Na ocorrência de uma falha não prevista, o sistema simplesmente não terá um comportamento definido e seu estado possivelmente será inconsistente.

A principal tarefa na elaboração de um modelo de falhas é arbitrar quais falhas serão tratadas pelo sistema ou simplesmente excluídas do modelo. As falhas não toleradas pelo sistema devem ser as mais raras possíveis, na tentativa de obtenção de um maior grau de confiabilidade (porém não perfeito) no comportamento do sistema.

As falhas do sistema são divididas em duas classes: *aceitáveis* e *inaceitáveis*. A premissa básica de um modelo de falhas é que as falhas inaceitáveis nunca ocorrem. De outra forma, podemos dizer que o comportamento de um sistema é *intolerável* na ocorrência de uma falha inaceitável. O comportamento do sistema é dito *tolerável* na presença de falhas previstas no modelo de falhas. Finalmente, dizemos que o comportamento do sistema é *normal* quando não há nenhum tipo de falha.

2.2.3 Recuperação de Falhas

Os mecanismos de recuperação de falhas visam colocar o banco de dados em um estado consistente que permita o prosseguimento normal do processamento. Um requisito importante destes mecanismos é que sejam *congruentes* com o funcionamento do sistema [39]. Isto é, a presença de falhas pode impedir o sistema de responder a alguns estimulos, até mesmo forçando que alguns resultados sejam desfeitos, mas o sistema deve tentar agir o mais próximo possível do que lhe foi requisitado.

As técnicas de recuperação em banco de dados são baseadas em estratégias de *recuperação retroativa*. Estas estratégias tentam restaurar o sistema a um estado anterior supostamente sem erros, antes de prosseguirem com a operação normal do sistema.

As técnicas de recuperação retroativa dependem da provisão de *pontos de recuperação* e do *escopo de recuperação*. Os pontos de recuperação são os meios pelos quais o estado do banco de dados pode ser registrado e reinstalado posteriormente. Várias técnicas podem ser empregadas para a obtenção de pontos de recuperação. Os mecanismos baseados em *checkpoints* envolvem o conhecimento anterior de todos os recursos que os processos podem modificar, o que em muitos casos significa todo o armazenamento. As técnicas baseadas em registros de inspeção requerem a gravação de todas as modificações que foram feitas por um processo. Finalmente, os mecanismos de recuperação baseados em *cache* constituem um compromisso realista entre as duas técnicas anteriores, armazenando apenas os estados originais dos registros que foram modificados.

O escopo de recuperação, como o nome indica, delimita a parte do estado interno a ser registrada para uso posterior em uma eventual recuperação. Idealmente, o escopo de recuperação deveria corresponder exatamente à parte do estado interno do sistema onde se resume o erro. Entretanto, a previsão dos possíveis danos de um erro é uma tarefa difícil, pois envolve o registro de todo o fluxo de informação de um processo e a presença simultânea de diversos mecanismos de verificação de consistência do estado do sistema. Uma alternativa muito mais simples para a delimitação do escopo de recuperação é simplesmente definir uma *área suspeita* que concentraria o foco da recuperação.

A delimitação de áreas suspeitas é naturalmente obtida se o sistema for abstraído como sendo composto por ações atômicas executadas por seus componentes. Através desta simplificação, as relações entre os processos podem ser definidas em alto nível, especificando as restrições de integridade do sistema, sem entrar em detalhes de como isso é feito pelos componentes. Este alto nível de abstração é obtido pela propriedade de uma ação atômica encapsular os fluxos de informação e os estados internos dos componentes [45]. Com esta propriedade, os pontos de recuperação podem ser salvos a cada operação atômica e armazenadas somente até o término desta ação. Isto define exatamente os pontos e o escopo da recuperação, fundamentais para a implementação dos mecanismos de recuperação para trás.

O conceito de ação atômica é de grande relevância em sistemas de banco de dados, pois provê uma estrutura simples para a os mecanismos de recuperação de erros. Nesta estrutura, se uma falha impedir o exito de uma ação atômica, o mecanismo de recuperação se resume a restaurar os estados de todos os objetos que esta ação modificou, para o estado anterior à sua execução. Caso contrário, a presença de falhas permitiria que os estados intermediários dos objetos resultantes de execuções parciais fossem observados por ações externas ao escopo da ação atômica.

Uma ação atômica é na verdade uma generalização do conceito de transação, largamente empregado no contexto de sistemas de banco de dados. Este conceito é examinado em detalhes na próxima seção.

2.3 O Conceito de Transação

A manipulação de dados por um programa ou usuário em um ambiente multiusuário normalmente exige algum tipo de proteção contra ações indesejáveis ou que não se tem controle. Durante a execução de um programa é comum que as informações manipuladas encontrem-se em algum estado temporário ou preliminar. Algum dado, por exemplo, pode estar errado ou desatualizado, sendo alterado ou até removido posteriormente no decorrer do programa. Se outros programas tiveram acesso a este dado e tomaram decisões baseando-se no seu valor, certamente encontrarão problemas. Adicionalmente, se alguma falha interromper a execução do programa, as informações temporárias podem ficar refletidas permanentemente no banco de dados, o que torna o seu estado inconsistente.

Estas situações e dependências podem ser evitadas, desde que os dados manipulados por um processo sejam de alguma forma separados do outros processos. Este isolamento é alcançado com a introdução do conceito de *transação* [17]. No contexto dos sistemas de banco de dados, a transação é a unidade básica de consistência e confiabilidade, sendo conseqüentemente o maior paradigma nas técnicas de sincronismo e recuperação de falhas nestes sistemas.

De uma forma geral, o conceito de transação é empregado de duas formas distintas para preservar a consistência dos dados: *falha atômica* e *sincronismo* [39]. Falha atômica é uma propriedade de uma transação que garante todas as ações de uma transação ocorram ou, caso contrário, nenhuma delas. O sincronismo, por sua vez, é o mecanismo que assegura que não há inconsistências em acessos concorrentes a um dado. Isto é obtido, fazendo-se com que a execução de duas transações sempre aparentem ter ocorrido sequencialmente.

Para que uma transação seja um conceito eficaz na manutenção da consistência e confiabilidade de um banco de dados, quatro propriedades são fundamentais: Atomicidade, Consistência, Isolamento e Durabilidade. Estas propriedades são conhecidas pela contração *ACID* e são examinadas em detalhes nas próximas seções.

2.3.1 Atomicidade

Se uma ação é interrompida por uma falha qualquer durante a sua execução, não há nenhuma garantia de que o estado do banco de dados será consistente após esta falha. Logo, as ações devem executar completamente, ou *atomicamente*.

As únicas ações atômicas existentes, são as que podem ser diretamente implementadas em *hardware*. Infelizmente, as operações de manipulação de banco de dados são muito complexas, o que inviabiliza o seu mapeamento direto em *hardware*. Em face desta dificuldade, optou-se por garantir uma propriedade mais fraca, porém suficiente para manter a consistência dos dados. A abordagem consiste em estender a garantia de atomicidade a um conjunto de operações - a transação - em vez de assegurar as ações individualmente.

Uma transação é um conjunto de ações que possui a propriedade de *falhar atomicamente*. Todas as ações executadas no contexto de uma transação são indivisíveis: ou todas as ações estão refletidas no banco de dados ou nenhuma alteração foi efetuada. Isto é, se uma transação é executada sem falhas, dizemos que a transação *terminou corretamente*. Caso contrário, dizemos que a transação *abortou*.

2.3.2 Consistência

Uma transação que termina corretamente, e que por conseqüência compromete-se com os seus resultados, sempre preserva a consistência do banco de dados. Ou seja, uma transação que termina corretamente é uma operação indivisível que mapeia um estado consistente do banco de dados em um outro estado também consistente. Adicionalmente, como os resultados de uma transação que abortou não são refletidos no banco de dados, podemos dizer que a execução de uma transação não invalida a consistência do banco de dados, independente de seu sucesso ou fracasso.

A propriedade de *consistência* garante que o estado de um banco de dados é correto se e somente se ele contém os resultados das transações terminadas corretamente. Este estado é chamado de *consistente por transações* ou *logicamente consistente* [28, 25]. Por sua vez, uma transação não observa nada além dos efeitos de transações corretas e, por definição, sempre gera uma atualização consistente do banco de dados.

Os requisitos de consistência em um banco de dados variam de acordo com a liberdade que as transações possuem ao manipular dados alterados por outras transações ainda não terminadas, ou *dados sujos* [21]. Baseando-se então nas restrições impostas às transações ao operar com dados sujos, pode-se classificar o grau de consistência mantida por um banco de dados em quatro níveis sucessivos, onde um nível de maior consistência engloba as imposições de todos os níveis inferiores. Os graus são:

Grau 0: Uma transação T observa grau 0 de consistência se:

1. T não reescreve dados sujos de outras transações.

Grau 1: Uma transação T observa grau 1 de consistência se:

1. T não reescreve dados sujos de outras transações.
2. T não assegura nenhuma alteração enquanto todas as suas alterações tiverem terminadas (i.e., até o fim da transação).

Grau 2: Uma transação T observa grau 2 de consistência se:

1. T não reescreve dados sujos de outras transações.
2. T não assegura nenhuma alteração enquanto todas as suas alterações não estejam terminadas (i.e., até o fim da transação).
3. T não lê dados sujos de outras transações.

Grau 3: Uma transação T observa grau 3 de consistência se:

1. T não reescreve dados sujos de outras transações.

2. T não assegura nenhuma alteração enquanto todas as suas alterações não estejam terminadas (i.e., até o fim da transação).

3. T não lê dados sujos de outras transações.

4. Outras transações não sujam nenhum dado lido por T antes do término de T.

Em particular, o grau 3 de consistência garante *consistência total* ou *consistência estrita* ao banco de dados, uma vez que não permite qualquer tipo de acesso a dados sujos por uma transação. Desta forma, todos os resultados parciais (sem garantias de validade) de uma transação ficam isolados das demais transações. O isolamento entre transações é tratado a seguir.

2.3.3 Isolamento

Isolamento é a propriedade que assegura que todas as transações sempre observem um banco de dados consistente. Ou seja, os resultados parciais de uma transação ainda não terminada precisam ser invisíveis a outras transações executando concorrentemente. Se uma transação lê um dado que esteja sendo atualizado concorrentemente por uma outra transação, não pode-se garantir que o valor lido pela primeira transação será correto.

A restrição de não se permitir o acesso a resultados incompletos por outras transações endereça o problema das *atualizações perdidas*. De outra forma, se uma transação em andamento permite a leitura de seus dados parciais e posteriormente a sua execução é abortada, todas as outras transações que leram os seus dados temporários também terão de abortar. Este problema é conhecido como *abortos em cascata*.

A partir dos graus de consistência apresentados na seção anterior, podemos classificar as transações em diferentes níveis de isolamento. Quanto maior o grau de consistência de um banco de dados, maior o isolamento entre as transações. Particularmente, o grau 3 de consistência garante isolamento total entre as transações. Neste caso, quando duas transações entram em conflito no acesso a um dado, uma delas deverá esperar o término da execução da outra. Mesmo na presença de conflitos, com este grau de consistência, a execução de duas transações sempre aparenta ter ocorrido seqüencialmente. Esta seqüência é chamada de *seqüência estrita de execução*, sendo de especial interesse nas técnicas de controle de concorrência descritas na seção 2.4.

2.3.4 Durabilidade

Após o término de uma transação, os seus resultados estão compromissados no banco de dados. O sistema precisa garantir que estes resultados irão resistir a possíveis falhas subseqüentes. Ou seja, durabilidade é a propriedade que garante que os resultados de uma transação que terminou corretamente serão permanentes e não podem ser apagados do banco de dados. Por outro lado, como os resultados de uma transação terminada são corretos por definição, os efeitos de uma transação concluída que tenha erros (dados incorretos, por exemplo) só podem ser removidos por meio de outra transação.

A durabilidade de uma transação é uma propriedade que está relacionada com a recuperação de falhas no banco de dados. Isto é, como recuperar um estado consistente do banco de dados após a ocorrência de uma falha, onde todas as ações compromissadas estejam refletidas. Esta questão é discutida em maiores detalhes na seção 2.5.

2.4 Controle de Concorrência

Os mecanismos de *controle de concorrência* [6] são responsáveis pela manutenção das propridades de consistência e de isolamento entre transações. Estas propriedades são mantidas assegurando-se o sincronismo no acesso concorrente a dados compartilhados por transações distintas. Este acesso concorrente deve ser totalmente transparente às transações que, por sua vez, têm a ilusão que executam serialmente [42], uma após a outra. Os mecanismos de concorrência usualmente são classificados em *pessimistas*, no qual o controle de concorrência acredita que muitas transações conflitarão entre si, e em *otimistas*, que supõem que não haverá um número significativo de conflitos entre transações.

2.4.1 Controle de Concorrência Pessimista

Os mecanismos de controle de concorrência pessimistas são divididos de acordo com a premissa que emprega para assegurar o sincronismo no acesso a dados compartilhados. As premissas usadas nestes mecanismos são *locking*, *timestamps* e *métodos híbridos*.

2.4.1.1 Locking

O método mais comum de controle de concorrência pessimista é baseado no conceito de *lock* [17, 22]. Neste método, cada elemento de dado está associado a um *lock* que indica se o dado está sendo usado por uma determinada transação. Ou seja, para que uma transação t_1 tenha acesso a um elemento de dado d, o *lock* de d deve ser examinado primeiro. Caso nenhuma outra transação possua este *lock*, o acesso a d é liberado para t_1 e o *lock* deste elemento é atribuído a t_1. De outro modo, se uma outra transação t_2 possui o *lock* de d, t_1 terá que esperar que t_2 libere o *lock* deste elemento para que consiga acessar d. A principal idéia do mecanismo de *lock* é garantir que dados compartilhados sejam manipulados por uma transação de cada vez.

Mais especificamente, o objetivo dos mecanismos de *locks* é evitar a manipulação de um mesmo dado por operações conflitantes. Dizemos que duas operações são *conflitantes* se a execução concorrente destas operações sobre o mesmo dado em transações distintas fere a consistência do banco de dados. Se não há conflito entre operações de diferentes transações que manipulem o mesmo dado d, o grau de concorrência entre estas transações pode ser melhorado, permitindo-se que mais de uma transação atribua um *lock* a d.

A sincronização de operações conflitantes é realizada através da classificação dos *locks* em *exclusivos* ou *compartilhados*, que se relacionam respectivamente com as operações de atualização e leitura de elementos de dados. Quando uma transação t_i deseja ler um elemento de dado d, deve antes obter um *lock* do tipo compartilhado associado a d. Analogamente, no caso de operações de atualização, o *lock* a ser obtido seria do tipo exclusivo.

O acesso concorrente a dados por operações não conflitantes de transações distintas é estabelecido a partir da definição das relações de compatibilidades entre os tipos de *locks*. Dois tipos de *locks* são ditos *compatíveis* se duas transações que acessam um mesmo dado podem obter *locks* destes tipos ao mesmo tempo (tabela 2.1).

	Compartilhado	Exclusivo
Compartilhado	compatível	não compatível
Exclusivo	não compatível	não compatível

Tabela 2.1: Compatibilidade de tipos de *locks*

Em um ambiente onde cada transação pode potencialmente modificar qualquer objeto que acessa, todas as transações devem seguir algumas regras na atribuição de *locks* que visam garantir o seu total isolamento e sua atomicidade. Isto é feito através das definições de transações bem formadas em duas fases [17]. Mais especificamente, uma transação é *bem formada em duas fases* se:

1. obtém um *lock* compartilhado associado a um elemento de dado d antes de efetuar uma leitura em d

2. obtém um *lock* exclusivo associado a um elemento de dado d antes de efetuar uma atualização em d

3. não obtém um *lock* l associado a um dado d caso d já possua um *lock* incompatível a l.

4. antes de seu término, remove todos os seus *locks*

5. não obtém nenhum *lock* após ter removido pelo menos um *lock* de qualquer elemento de dado.

Em particular, a última regra é conhecida por assegurar o *two-phase locking*. Este método executa as transações em duas fases distintas e comprovadamente [17] assegura que as transações executadas desta forma são serializáveis. Cada transação possui uma fase de *crescimento*, onde obtém *locks* e acessa elementos de dados, e uma fase de *redução*, na qual libera todos os seus *locks*. O algoritmo de *two-phase locking estrito*, por sua vez, só permite a liberação de *locks* no momento do término da transação. Esta restrição adicional evita que outras transações acessem dados parcialmente modificados por uma transação t antes do seu término, o que implicaria no aborto destas outras transações caso t abortasse.

2.4.1.2 Timestamps e Métodos Híbridos

Os algoritmos de controle de concorrência baseados em *timestamps* [6] organizam a ordem de execução das transações através da desginação de *timestamps* para as transações e também para os elementos de dados. Os algoritmos híbridos [6], por sua vez, combinam o mecanismo de *locking* tradicional com *timestamps* para otimizar o nível de concorrência entre os dados [27]. Com a exceção honrosa do *Commit_LSN* [31], um mecanismo híbrido implementado em diversos banco de dados comerciais, os algoritmos baseados em *timestamps* e híbridos são pouco difundidos em sistemas comerciais ou em protótipos acadêmicos.

2.4.2 Controle de Concorrência Otimistas

Os mecanismos de controle de concorrência otimistas [22, 50] foram desenvolvidos com o objetivo de funcionarem sem as restrições de bloqueio de acesso intrínsecas às técnicas baseadas em *locks*. Estes mecanismos são otimistas no sentido que eles assumem explicitamente que os conflitos entre transações são eventos de rara ocorrência.

Desta forma, os mecanismos otimistas se apoiam na eficiência na esperança de que conflitos não ocorram.

Nestes mecanismos, o controle da concorrência efetivamente ocorre no término da transação, quando o mecanismo enfim checa a existência de conflitos potenciais. Se um conflito concebível é detectado, a transação simplesmente é abortada. Em resumo, os mecanismos de controle de concorrência otimistas consistem de três fases: *leitura*, *validação* e *atualização*, onde o término da fase de leitura corresponde ao término usual de uma transação t, onde o direito de t abortar uniteralmente é abdicado.

Os mecanismos de controle de concorrência otimistas, entretanto, possuem sérias limitações no suporte a transações distribuídas, estruturas de acesso, *rollbacks* parciais, sub-transações e diferentes níveis de isolamento de uma transação [32, 23]. Estas limitações fazem deste tipo de controle de concorrência pouco apropriado para o contexto deste trabalho.

2.5 Técnicas de Recuperação Centralizada

A manutenção das propriedades de atomicidade e durabilidade de uma transação é atribuição do mecanismo de recuperação de um banco de dados. Este mecanismo é responsável pelas operações sobre transações (iniciar, terminar ou abortar), bem como pelas operações de leitura e atualização de dados. Adicionalmente, o mecanismo de recuperação tem a função de recuperar a consistência do banco de dados após a ocorrência de uma falha. Nesta seção nos restringimos ao contexto de um banco de dados centralizado. As questões relativas ao ambiente distribuído, em particular a coordenação de atualizações em múltiplos nós e a replicação de dados em localidades remotas, são tratadas respectivamente nas seções 2.6 e 2.7.

2.5.1 Relacionamento com a Gerência de Buffers

Tradicionalmente, os mecanismos de recuperação se baseiam na premissa de *armazenamento estável*, assumindo que o banco de dados encontra-se seguramente armazenado em memória secundária. Chamamos a versão do banco de dados em memória secundária de *banco de dados estável*. A manipulação do banco de dados, entretanto, somente é realizada por intermédio de um conjunto de *buffers* em mémoria principal com o objetivo de se aumentar o desempenho no acesso aos dados. Chamamos este conjunto de *buffers* de *banco de dados volátil*.

A transferência dos dados entre a versão volátil e a estável do banco de dados é responsabilidade do *mecanismo de gerência de buffers* [16]. Este mecanismo efetua a transferência dos dados em memória secundária para os *buffers* em memória principal, bem como a gravação dos dados em armazenamento estável após a execução de uma operação de atualização.

Como o acesso físico aos dados em disco é muito mais caro do que os feitos em memória principal, o principal objetivo do mecanismo de *buffers* é otimizar o desempenho do servidor através da redução do número de acessos físicos a disco. Idealmente, o mecanismo de gerência de *buffers* deve adotar políticas próprias na transferência dos dados atualizados para memória estável, baseando-se em critérios relevantes ao desempenho do sistema, abstraindo-se de assegurar as propriedades *ACID* das transações. Desta forma, é possível que em um determinado instante as alterações parciais de uma transação sejam copiadas em disco antes de seu término, bem como que algumas alterações esteja refletida em memória estável ao final da transação. Esta liberdade, na eventual ocorrência de uma falha, faria com que estas inconsistências temporárias se refletissem na versão estável do banco de dados, o que exigiria ações corretivas do mecanismo de recuperação de falhas. Ou seja, o mecanismo de recuperação teria que desfazer as atualizações de transações incompletas e refazer as atualizações das transações que terminaram normalmente.

De outra forma, se restringirmos a política de atualização do gerente de *buffers*, exigindo que as atualizações de uma transação ficassem retidas em *buffers* até o término da transação, quando seriam transferidas para memória estável, nenhuma ação corretiva seria necessária na recuperação de falhas. Esta alternativa, entretanto, não é atraente, pois exigiria uma grande quantidade de *buffers* em memória principal para armazenar todas as atualizações retidas. Além de interferir na política de tranferência do gerente de *buffers*, o que impactaria significativamente o desempenho do sistema [25]. Normalmente, opta-se por não restringir a atuação do gerente de *buffers*, privilegiando o desempenho do sistema na ausência de falhas, adotando-se em contra-partida um mecanismo de recuperação mais complexo e demorado.

2.5.2 Informação para Recuperação

Na ocorrência de uma falha do sistema, a versão estável do banco de dados pode estar inconsistente devido a perda das atualizações que ainda se encontravam em memória principal e também aos resultados parciais de transações inacabadas. Para que o banco de dados recupere a sua consistência, é necessário que se mantenha alguma informação adicional sobre o estado do banco de dados no momento da falha. Dá-se o nome de *informação para recuperação* a esta informação adicional mantida para a recuperação do estado do banco de dados.

O tipo de informação para recuperação necessária para a restauração do banco de dados é dependente do método que o banco de dados realiza as suas atualizações. Existem duas alternativas para esta operação que são descritas a seguir.

2.5.2.1 Atualização sem Superposição

Se a cada atualização, o sistema não altera o dado em memória estável, mantendo o novo valor em separado, dizemos que a atualização é *sem superposição*. A técnica mais comum de atualização sem superposição baseia-se no conceito de *shadow pages*

[30]. Esta técnica emprega duas cópias para cada dado, onde a segunda cópia é mantida intacta durante as operações de atualização do dado. Caso haja necessidade de desfazer uma atualização, o conteúdo da primeira cópia é restaurado a partir da segunda. Os métodos baseados em atualização sem superposição, apesar de sua simplicidade, impõem dificuldades na implementação de mecanismos de recuperação mais eficientes [20, 34].

2.5.2.2 Atualização com Superposição

No método de atualização *com superposição*, como o nome indica, uma operação de atualização escreve por sobre o conteúdo antigo do dado. Desta forma, é necessário que alguma informação adicional seja mantida para auxiliar na recuperação de falhas do banco de dados. Esta informação tipicamente é mantida em um *log*. Desta forma, a cada operação de atualização, um registro é acrescido ao *log* contendo a versão anterior e posterior do dado atualizado. As versões anterior e posterior de um dado são respectivamente empregadas na recuperação de falhas para desfazer alterações de transações incompletas e refazer alterações de transações terminadas.

De forma semelhante ao banco de dados volátil, o *log* também é mantido em *buffers* na memória principal que é igualmente transferido para o *log estável* em memória secundária. O *log* pode ser gravado em memória estável de duas formas distintas. Ele pode ser gravado *sincronamente* (ou *forçado*), quando a cópia em armazenamento secundário é atualizada simultaneamente à inserção de um novo registro ao *log*. De outra forma, o *log* é gravado *assincronamente* quando a transferência para memória secundária se dá no melhor momento segundo critérios de desempenho. Quando um registro é forçado no *log*, o processamento da transação é interrompido até que o registro esteja gravado em disco, o que aumenta o tempo de processamento da transação. Este procedimento, em compensação, facilita a operação de recuperação de falhas.

Independente do modo que o *log* é gravado em memória estável, é importante que se estabeleça um protocolo que garanta a consistência do banco de dados. O protocolo deve assegurar que o *log* estável seja sempre atualizado antes da versão estável do banco de dados. Isto evita que uma eventual falha impeça a atualização do *log* após a gravação de uma atualização, o que impossibilitaria a recuperação da consistência do banco de dados. Este protocolo é conhecido como *write-ahead logging (WAL)* [22] e pode ser definido pelas seguintes regras:

1. Antes da atualização da versão estável de um dado, a sua imagem anterior à atualização deve estar armazenada no *log* estável.

2. Quando uma transação termina, as imagens posteriores das atualizações devem ser armazenadas no *log* estável antes da atualização do banco de dados estável.

2.5.3 Checkpoints

Os mecanismos de recuperação se valem de *checkpoints* para reduzir o trabalho em caso de falhas [22], o que por consequência também reduz o tempo de indisponibilidade do banco de dados. Um *checkpoint* é na verdade um marco no qual se garante a consistência do banco de dados, o que elimina a necessidade de qualquer esforço de recuperação a estados anteriores ao *checkpoint*.

Usualmente, um *checkpoint* não é uma operação eficiente pois exige a interrupção total no processamento de transações. Isto não acontece com o uso de *fuzzy checkpoints* que resguardam incrementalmente o estado do banco de dados, não impedindo a execução normal de transações.

2.6 Terminação em Ambiente Distribuido

Em um banco de dados distribuído, os dados encontram-se fisicamente dispersos ao longo de diversas localidades distintas. Neste ambiente, o processamento de uma transação muitas vezes envolve a atualização de dados em vários locais, o que torna complexa a tarefa de se garantir a atomicidade de uma transação distribuída. Assumindo que cada localidade possui uma estratégia de recuperação que garante a atomicidade localmente, o problema consiste em garantir que todos locais unanimemente abortem ou terminem corretamente a transação distribuída. Caso a decisão sobre o término da transação seja apenas parcial, o banco de dados conterá inconsistências.

A garantia da atomicidade global de uma transação distribuída é obtida através de *protocolos de terminação*, onde o protocolo mais difundido é o *two-phase commit* [22], visto em detalhes na próxima seção.

2.6.1 Two-Phase Commit

O *two-phase commit* é um protocolo muito simples e elegante que garante a terminação atômica de transações distribuídas. Este protocolo estende os efeitos das terminações atômicas locais para as transações distribuídas, insistindo que todas as localidades envolvidas cheguem a um acordo sobre a terminação da transação antes que seus efeitos tornem-se permanentes no banco de dados.

O princípio básico do *two-phase commit* é muito claro. Como o próprio nome sugere, a terminação da transação é feita em duas fases. Na primeira fase do protocolo, antes de qualquer localidade se compromissar com os resultados da transação, todas as localidades devem abdicar de seu direito de abortar a transação unileralmente. Finalmente na segunda fase, se todos os participantes abriram mão de seu direto de abortar, a transação termina normalmente e os seus resultados ficam refletidos no banco de dados. De outro modo, a transação é abortada na segunda fase, caso pelos menos um participante faça uso de seu direito de veto na fase anterior do protocolo.

No modelo de transação distribuída adotado pelo *two-phase commit*, dois tipos de componentes participam de uma transação: *Gerenciadores de Recursos Locais*

(GRLs) e *Gerenciadores de Transações (GTs)*. Os GRLs são os participantes que somente possuem o controle sobre os seus recursos locais, tipicamente são banco de dados ou sistemas de arquivos. Os GTs são responsáveis pela coordenação de múltiplos participantes, que podem ser GRLs ou outros GTs remotos.

Entre os GRLs e GTs que participam do *two-phase commit*, existe a figura do *coordenador da transação* e de um ou mais *subordinados*. O coordenador é um GT que age em nome da aplicação que iniciou a operação de término da transação distribuída. Os subordinados são GRLs ou outros GTs remotos atuando em nome de outros processos da transação distribuída. Os GTs remotos também podem possuir GRLs ou GTs subordinados, formando uma estrutura hierárquica de coordenação da transação. Durante a execução da transação, os subordinados se comunicam apenas com o seu coordenador imediatamente superior na hierarquia, não havendo comunicação direta entre estes. O coordenador da transação tem a responsabilidade de chegar a decisão final sobre o desfecho da transação, seja terminar corretamente ou abortar. Os coordenadores subordinados, por sua vez, propagam a decisão do coordenador aos seus GTs e GRLs dependentes.

Durante a operação de terminação, todos os participantes da transação armazenam informações sobre os estados intermediários da execução, de forma a permitir o seu reestabelecimento após uma eventual falha. Estas informações são escritas no *log* do participante de duas formas distintas: *escritas forçadas* ou *escritas não forçadas*. Durante a execução de uma escrita forçada, qualquer operação do participante é suspensa e este nada faz enquanto o registro não está seguramente armazenado em memória não volátil. As escritas não forçadas, pelo contrário, não suspendem a operação do participante, mas não oferecem garantias de sobrevivência a uma falha do sistema. Apesar de representar uma grande sobrecarga no tempo de processamento da terminação, toda informação vital para o processamento correto após a ocorrência de uma falha deve ser armazenada através de escrita forçada.

Na ausência de falhas, o funcionamento do *two-phase commit* sem GTs subordinados é composto dos seguintes passos:

1. O coordenador inicia a primeira fase do protocolo de terminação enviando em paralelo a mensagem `PREPARAR` para todos os seus subordinados, perguntando se estes estão inclinados a terminar corretamente a transação.

2. Cada subordinado que esteja inclinado a permitir que a transação seja terminada corretamente, em primeiro lugar força a escrita de um registro `PREPARADA` no *log*, em seguida, envia uma mensagem `VOTO SIM` ao coordenador e aguarda a decisão final do coordenador (terminar corretamente ou abortar). Neste ponto, o processo encontra-se no estado *preparado*. Cada subordinado que deseje que a transação seja abortada, força a escrita de um registro `ABORTADA` no *log* e envia uma mensagem `VOTO NÃO` ao coordenador. Uma vez que o `VOTO NÃO` é na verdade um veto, o subordinado tem certeza que a transação será definitivamente abortada. Portanto, este subordinado não precisa aguardar nenhuma outra mensagem relativa a esta transação do coordenador e pode liberar os *locks* ou qualquer outra informação local sobre a transação recém-abortada.

3. Após o recebimento dos votos de todos os subordinados, o coordenador inicia a segunda fase do protocolo. Se todos os votos foram `VOTO SIM`, o coordenador força a escrita de um registro `TERMINADA` no *log* e envia a mensagem `TERMINADA` a todos os subordinados. Após a escrita do registro no *log*, a transação atinge o seu *ponto de terminação*. Se o coordenador receber pelo menos um `VOTO NÃO`, ele força a escrita de um registro `ABORTADA` no *log* e envia mensagens *ABORTADA* a todos os participantes que estiverem no estado preparado, ou seja, que não votaram pelo aborto da transação.

4. Após receber a mensagem `TERMINADA` do coordenador, cada subordinado força a escrita de um registro `TERMINADA` no *log*, envia uma mensagem `RECEBIDO` ao coordenador e termina a transação. Após receber a mensagem `ABORTADA` do coordenador, cada subordinado força a escrita de um registro `ABORTADA` no *log*, envia uma mensagem `RECEBIDO` ao coordenador e aborta a transação.

5. O coordenador, após receber as mensagens `RECEBIDO` de todos os participantes aos quais enviou mensagens na segunda fase, escreve um registro `FIM` no *log*.

O motivo de se exigir que o armazenamento dos registros `TERMINADA` e `ABORTADA` seja feito no *log* através de escrita forçada, antes do envio da mensagem `RECEBIDA` ao coordenador, é evitar que no caso de falha em um dos subordinados, este último nunca tenha que perguntar qual foi o resultado final de uma transação ao coordenador, tendo confirmado o recebimento da mensagem. O recebimento da mensagem `RECEBIDA`, portanto, indica com precisão ao coordenador que todos os subordinados estão informados do desfecho final da transação.

Para que a atomicidade da transação seja garantida, sempre que um subordinado acusar o recebimento de uma mensagem ao coordenador através da mensagem `RECEBIDA`, o subordinado deve ter total certeza que nunca terá que perguntar novamente ao coordenador sobre a mensagem que recebeu. Isto é garantido forçando-se a escrita imediata de um registro no *log* antes do envio da resposta ao coordenador.

No caso da presença coordenadores subordinados em uma transação distribuída, a decisão do coordenador da transação é propagada pelos coordenadores subordinados a seus dependentes. De forma semelhante, as respostas dos subordinados são passada ao coordenador imediatamente superior na hierarquia até chegar ao coordenador da transação. Geralmente, um participante da hierarquia não sabe se o seu superior é o coordenador da transação ou um nó intermediário, do mesmo modo que um coordenador não sabe se seus subordinados são nós intermediários ou finais da hierarquia.

O fluxo de propagação de votos em uma hierarquia de coordenação no *two-phase commit* é análogo à situação sem coordenadores intermediários. No caso de hierarquia de coordenação, o coordenador intermediário comunica a sua decisão (parcial) sobre a terminação a seu coordenador superior, em vez de passar a decisão (que seria final) aos seus subordinados, como seria no caso de ausência de hierarquia. Mais precisamente, o coordenador intermediário envia uma mensagem `VOTO SIM` a seu coordenador superior, se todos os seus subordinados foram unânimes no seu desejo de terminar corretamente a transação. De outra forma, um coordenador intermediário manifesta o desejo de abortar uma transação, enviando uma mensagem `VOTO NÃO` a seu coordenador superior, caso pelo menos um de seus subordinados tenha interesse no aborto da transação.

2.6.2 Terminação sem Bloqueio

Apesar da comprovada eficacia com que o *two-phase commit* garante a atomicidade de transações distribuídas, este protocolo na presença de falhas em partes do sistema pode bloquear indefinidamente a execução normal de um participante sadio [53, 18]. Isto impede qualquer decisão sobre o término da transação e não permite que o participante libere os recursos que alocou para uso exclusivo, principalmente *locks* em fragmentos de dados. Desta forma, os recursos reservados tornam-se indisponíveis para outras transações enquanto não há um desfecho para a transação bloqueada. Particularmente, no caso de falha do coordenador da transação, todos os outros participantes ficarão bloqueados. A diminuição da disponibilidade do sistema decorrente das situações de bloqueio é inaceitável para muitas aplicações.

Em algumas variações do *two-phase commit* [26, 57], as localidades bloqueadas baseiam-se em heurísticas para decidir unilateralmente se abortam ou terminam corretamente uma transação bloqueada. Infelizmente, estas heurísticas podem ocasionar inconsistências, já que os participantes bloqueados podem optar por desfechos distintos para a transação.

O *Three-Phase Commit* [52] é um protocolo de terminação projetado para não acarretar situações de bloqueio quando as falhas são restritas a uma localidade apenas. Na ocorrência de falhas em múltiplas localidades, existem casos de bloqueio, mas o protocolo preserva a consistência da decisão de terminação. Entretanto, na ocorrência de falhas de comunicação, este protocolo pode causar inconsistências no desfecho da transação.

2.6.3 Otimizações nos Protocolos de Terminação

O desempenho de um protocolo de terminação afeta substancialmente o volume de transações que um sistema pode processar [49]. Um protocolo de terminação otimizado pode aumentar o desempenho de um ambiente distribuído de duas formas distintas:

1. Reduzindo o tempo de processamento da terminação de cada transação e
2. Permitindo que os *locks* sejam liberados mais cedo, o que reduz o tempo de espera de outras transações.

Existem duas abordagens para a otimização de protocolos de terminação. O primeiro método concentra-se em reduzir o tempo de recuperação de uma transação. Os protocolos de terminação em diversas situações de falha causam o bloqueio de transações e, por conseqüência, de todos os *locks* adquiridos pela transação. Se o tempo de recuperação da falha é reduzido, os *locks* bloqueados podem ser liberados mais cedo, o que diminui o tempo de espera de outras transações (seção 2.6.2). Diversas variações do *two-phase commit* foram desenvolvidas com objetivo de se reduzir o tempo de bloqueio de uma transação. Entretanto, todas estas variações envolvem uma troca maior de mensagens entre os participantes, o que reduz o desempenho do protocolo de terminação no processamento normal de transações sem falhas.

Em ambientes voltados para o processamento de grandes volumes de transações, a redução do desempenho no processamento normal em troca de um tempo menor de recuperação muitas vezes não é aceitável. A segunda abordagem para o aumento do desempenho de um protocolo de terminação visa justamente atender estes ambientes. Durante a execução normal de um protocolo de terminação, duas operações se destacam pela sua longa duração e são usadas com critério nos protocolos otimizados:

1. Transmissão de mensagens pela rede entre o coordenador e os participantes e

2. Escritas no log (em particular as escritas forçadas que suspendem a execução da transação até a conclusão da operação).

Os protocolos *presumed abort* [35] e *presumed commit* [35, 36] são variações do *two-phase commit* que melhoram o desempenho da terminação normal de uma transação. Estes protocolos foram muito estudados no ambiente acadêmico e inúmeros produtos comerciais implementaram o *presumed abort* [7, 57, 51, 3, 4] e o *presumed commit* [26].

2.6.3.1 Presumed Abort

A modificação introduzida pelo protocolo *presumed abort* é o fato dos participantes não precisarem mais forçar a escrita de um registro de aborto no *log* antes de enviarem uma mensagem ao coordenador comunicando o recebimento de um comando de abortar. Se depois de uma falha, um subordinado encontra um registro de `PREPARADA` no *log*, o subordinado envia uma mensagem ao coordenador solicitando a recuperação da transação. Caso o coordenador não possua informações sobre a transação interrompida, ele presume que a transação abortou e envia ao subordinado um comando para abortar.

No *presumed abort*, o coordenador pode apagar todas as informações relativas a uma transação imediatamente após ter decidido abortá-la. O coordenador simplesmente escreve (sem forçar) um registro de aborto no *log*, não precisa da resposta dos subordinados ao comando de aborto e tão pouco escrever o registro `FIM` que indica o término da transação.

Em resumo, nos casos de transações abortadas, o *presumed abort* elimina o envio de mensagens de confirmação dos subordinado, a necessidade de que o registro do aborto seja forçado e a gravação do registro do término da transação. Logo, este protocolo consegue ser mais eficiente do que o *two-phase commit* original.

2.6.3.2 Presumed Commit

O protocolo *presumed commit*, por sua vez, é a variação do *two-phase commit* que assume que, caso não haja nenhuma informação registrada sobre uma determinada transação, deve-se considerar que esta terminou corretamente. Este protocolo, entretanto, não é exatamente uma variação dual do *presumed abort* [35]. Uma implementação desta natureza, no caso de uma falha do coordenador durante ou logo após o envio das mensagens `PREPARAR`, levaria os subordinados a assumir que a transação terminou corretamente. Como não há meios de se comunicar com o coordenador, os participantes assumem que a transação está correta. Quando o coordenador se recuperasse da falha, este decidiria pelo aborto da transação, o que é uma situação de inconsistência.

Na implementação correta do *presumed commit*, este problema é solucionado com o registro no *log* do coordenador de todos os nomes de seus subordinados, antes que estes entrem no estado de preparado. Deste modo, durante a recuperação, no caso de uma falha do coordenador após o envio das mensagens `PREPARAR`, o coordenador teria condições de se informar sobre o aborto da transação.

2.7 Replicação de Dados

Os mecanismos de recuperação tradicionais, baseados em *logs* e *checkpoints*, não satisfazem os requisitos de alta disponibilidade de inúmeras classes de aplicações. Nestes mecanismos, os dados permanecem indisponíveis durante o tempo de recuperação da falha, o que é inaceitável para estes tipos de aplicações.

No contexto de ambientes distribuídos, esta situação é contornada com o uso de técnicas de replicação de dados, onde cópias dos dados são armazenadas de forma redundante em múltiplos servidores. Com o uso de dados replicados, o sistema pode operar mesmo na presença de falhas em alguns nós do ambiente.

As técnicas de replicação de dados, entretanto, agravam os problemas de controle de concorrência e de recuperação de falhas [9, 29], exigindo a presença de protocolos de gerência de replicação, responsáveis pela coordenação da atualização e a interação dos clientes com as inúmeras cópias de um mesmo dado. Para que os benefícios da replicação sejam úteis, o custo da manutenção da consistência entre as cópias dos dados deve ser mantida em níveis aceitáveis, a despeito da complexidade adicionada ao ambiente distribuído. Este custo adicional de processamento e de comunicação deve-se aos diversos ciclos de troca de mensagens entre os servidores no controle dos dados replicados.

2.7.1 Consenso por Quorum

Diversos métodos para o controle de replicação são baseados na formação de quorums [6, 10], onde os algoritmos destes métodos são chamados de *algoritmos de consenso por quorum*.

A idéia básica destes algoritmos é atribuir um número de votos a cada servidor, onde este voto é usado no estabelecimento de quorums para a execução de operações. Para realizar uma operação de leitura ou atualização, uma transação precisa montar um quorum de votos que deve ser superior a um valor pré-determinado, dependendo do tipo da operação. Respectivamente, o *quorum de leitura*, Q_L, e o *quorum de atualização*, Q_A, estabelecem as quantidades mínimas de votos para as operações de leitura e atualização, estando relacionados com o número total de votos, N_V, pelas seguintes regras:

$$Q_L + Q_A > N_V \tag{2.1}$$

$$Q_A > \frac{N_V}{2} \qquad (2.2)$$

Estas duas condições são conhecidas como *invariantes de interseção de quorum*. A inequação 2.1 garante que uma operação de leitura e atualização não ocorrem simultaneamente enquanto que a inequação 2.2 impede que duas operações de atualização sejam realizadas ao mesmo tempo. É importante ressaltar que estes dois invariantes não estabelecem valores únicos para Q_L e Q_A, admitindo-se assim diversos conjuntos de valores para estas variáveis e para o peso do voto de um servidor e, consequentemente, para os algoritmos de consenso por quorum.

O algoritmo *read one, write all* é um caso particular da família de algoritmos de consenso por quorum, onde o peso do voto de um servidor e Q_L são iguais a 1 e Q_A tem o valor de N_V. Este algoritmo permite que a leitura seja feita indistintamente em qualquer servidor, mas em contra-partida exige que a atualização seja realizada simultaneamente em todos os servidores. Consequentemente, o algoritmo *read one, write all* obtém um melhor desempenho nas operações de leitura as custas de uma carga mais elevada nas atualizações, adequando-se bem aos ambientes cujas transações são predominantemente de leitura.

O *read one, write all* é um exemplo de *algoritmo estático*, uma vez que os valores de Q_L, Q_A e N_V são previamente determinados. Os algoritmos estáticos, contudo, não possuem desempenho satisfatório na presença de falhas em servidores. Nesta situação, caso não haja quorum para a realização de uma operação, o processamento da transação simplesmente é bloqueado até que o número mínimo de servidores esteja novamente disponível.

Esta restrição, entretanto, não se aplica aos *algoritmos dinâmicos*, nos quais os valores dos quorums são ajustados à medida que N_V varia com a disponibilidade dos servidores do ambiente. Exemplos destes algoritmos são os métodos de *available copies* [5], *missing writes* [15] e *virtual partition* [1].

Estes algoritmos, na ausência de falhas, atuam de forma semelhante ao *read one, write all*, permitindo a leitura indistinta em qualquer servidor e exigindo a atualização em todas as cópias. Na ocorrência de uma falha, entretanto, cada algoritmo adota estratégias distintas na atribuição dos quorums das operações. No caso do *missing writes*, o quorum de leitura é aumentado e o de atualização reduzido até que a falha seja corrigida. No algoritmo *virtual partition*, cada servidor mantém uma relação de todos os servidores com que consegue se comunicar, usada para estabelecer os valores dos quorums de suas operações. Finalmente, o método *available copies* tenta atualizar o dado nos servidores que se encontram disponíveis, realizando operações adicionais para validar suas atualizações e o acesso aos dados enquanto perdurar a falha no sistema.

2.7.2 Abordagem por Cópia Primária

A abordagem por *cópia primária* [2] é uma alternativa simples para a gerência de replicação. Seu principal atrativo é o fato de necessitar poucos recursos em sua atuação, sendo assim menos custosa para os servidores do ambiente distribuído. Estas características tornaram este método muito popular, fazendo-o prevalecer na prática sobre os complexos métodos baseados em quorums [8].

Esta abordagem consiste em um conjunto de servidores onde não mais do que um destes detém a cópia primária de um dado em qualquer instante. Um cliente somente manipula um dado através do servidor que ele acredita que seja o primário. As demais cópias do dado, chamadas de *cópias secundárias*, são atualizadas automaticamente pelo sistema na medida em que a primária sofre modificações. As cópias secundárias de um dado só são usadas por uma transação na ocorrência de uma falha no servidor que armazena a cópia primária do dado. Neste caso, os clientes devem ser notificados para que direcionem suas operações para a cópia secundária do dado.

A distinção nas permissões de acesso às cópias primárias e secundárias é um contraste com a liberdade entre as cópias de um dado que podem ser acessadas em operações de leitura nos algoritmos baseados em consenso por quorum. Esta distinção torna desnecessário o uso de mecanismos complexos de controle de concorrência distribuídos, uma vez que apenas uma única cópia de um dado está disponível para o processamento de transações. Este controle, necessário quando se permite o acesso indistinto a múltiplas cópias de um mesmo dado, requer ciclos adicionais de troca de mensagens, o que onera significamente o custo do mecanismo de gerência de replicação.

A propagação da atualização das cópias secundárias pode ser feita de duas formas distintas [19, 43]. A primeira alternativa, chamada de *2-safe*, as cópias primária e secundária são mantidas em perfeito sincronismo, sendo ambas atualizadas simultaneamente no decorrer da transação. De outra forma, a alternativa *1-safe* propaga as atualizações para a cópia secundária após o término da transação. Neste caso, a propagação é feita em lotes de transações, com o objetivo de otimizar os recursos do sistema, seja através do melhor uso de operações de E/S, bem como por amortizar o custo das operações de sincronismos entre todas as transações de um mesmo lote.

O método *2-safe* tem a vantagem de oferecer proteção total aos dados uma vez que as cópias secundárias estão sempre perfeitamente atualizadas. Este método, entretanto, exige a presença de um protocolo de terminação (seção 2.6) para coordenar as atualizações no decorrer da transação, o que incorre no acréscimo no número de mensagens transmitidas na rede. De outra forma, a propagação *1-fase* evita a sobrecarga do protocolo de terminação as custas de potencialmente perder algumas transação já concluídas em uma eventual falha que impeça a propagação de suas atualizações para as cópias secundárias. Algumas classes de aplicações podem tolerar a perda de algumas transações e optam por obter melhor desempenho com o uso de replicação *1-safe*.

Capítulo 3

Arquiteturas Paralelas para o Processamento de Transações

3.1 Introdução

As primeiras tentativas de obtenção de arquiteturas massivamente paralelas para o processamento de transações concentraram-se na construção de equipamentos específicos para este fim, as chamadas *máquinas de banco de dados* [11, 12]. Em particular, grande parte deste trabalho baseava-se no desenvolvimento de *hardware* especializado - tais como novas tecnologias de memória não-volátil e novos discos magnéticos ou óticos - na tentativa de diminuir o baixo desempenho dos discos magnéticos face os processadores e a memória principal. Como nenhuma destas tecnologias atingiu os seus objetivos, havia um forte sentimento de que os sistemas paralelos iriam ser inviáveis enquanto persistisse esta limitação no desempenho dos meios de armazenamento.

Hoje em dia, a diferença entre o desempenho dos discos e os processadores é ainda maior do que na época das máquinas de banco de dados. Entretanto, os ambientes paralelos para banco de dados já são uma realidade. Há mais de uma década, diversas empresas desenvolvem e comercializam máquinas paralelas para o processamento de transações. A geração atual destas máquinas não é mais baseada em equipamentos não convencionais altamente especializados, mas sim no uso de computadores comuns, de mercado, interligados em redes de alta velocidade.

Os microcomputadores e estações de trabalho atuais possuem um grande poder de processamento e armazenamento, com custos por unidade de processamento ou por megabyte muito menores do que os tradicionais computadores de grande porte. Os equipamentos de menor porte são muito modulares, o que torna fácil a sua expansão através da adição de mais mémoria, discos ou processadores mais potentes. Com a disseminação da tecnologia das redes locais de alta velocidade e do modelo cliente-servidor, os microcomputadores e estações de trabalho tornaram-se uma excelente plataforma para os ambientes de banco de dados distribuídos ou paralelos voltados para o processamento *on-line* de transações (*OLTP - On-Line Transaction Processing*).

3.2 Métricas e Objetivos do Paralelismo

O principal objetivo em se agrupar diversos processadores em paralelo é o de somar as suas capacidades individuais de processamento e armazenamento, visando obter um sistema único equivalente, operando em sinergia. Idealmente, o grande desafio das pesquisas em paralelismo é [12]:

> *O desafio é construir um processador infinitamente rápido a partir de uma infinidade de processadores de velocidade finita e construir uma memória infinitamente grande com desempenho infinito usando-se infinitas unidades de armazenamento de velocidade finita.*

Além das limitações físicas de velocidade e de capacidade de memória de um processador real, também existem dificuldades intrínsicas a um acoplamento ideal de processadores que precisam ser superadas. Ou seja, o objetivo é construir um sistema paralelo a partir de equipamentos convencionais com as seguintes características [12]:

1. Crescimento linear em desempenho
2. Crescimento linear em escala

Dado uma determinada tarefa executada em um sistema de pequeno porte, posteriormente submetida em um sistema maior, o *crescimento em desempenho* é dado por:

$$\text{crescimento em desempenho} = \frac{\text{tempo de execução no sistema menor}}{\text{tempo de execução no sistema maior}} \tag{3.1}$$

O crescimento em desempenho é dito *linear* se um sistema N vezes maior ou mais caro leva a um crescimento em desempenho de $C * N$, onde C é uma constante.

O crescimento em desempenho é medido mantendo-se o tamanho da tarefa constante e aumentando-se o tamanho do sistema. Por sua vez, o *crescimento em escala* avalia o caso em que se amplia simultaneamente a tarefa a ser realizada e a dimensão do sistema. Ou seja, o crescimento em escala é definido com a habilidade de um sistema N vezes maior executar uma tarefa também N vezes maior no mesmo tempo de execução do sistema original. O crescimento em escala é medido por:

$$\text{crescimento em escala} = \frac{\text{tempo de execução no sistema menor da tarefa menor}}{\text{tempo de execução no sistema maior da tarefa maior}} \tag{3.2}$$

Se o resultado desta equação é uma constante, então o crescimento em escala é dito *linear*. A medição do crescimento em escala depende evidentemente da natureza do problema a ser resolvido. Por exemplo, o custo de uma operação de ordenação é de $O(nlog(n))$, o que não permite um crescimento linear em escala de uma tarefa desta natureza.

O que na verdade é avaliado quando se mede o crescimento em escala ou em desempenho de um sistema paralelo para uma determinada tarefa são os aspectos intrínsecos à arquitetura paralela do sistema que influem em seu crescimento. As barreiras mais comuns para o crescimento linear em escala e em desempenho são:

inicialização O tempo necessário para iniciar uma operação paralela. Se milhares de processos precisam ser inicializados, este tempo pode facilmente sobrepor-se sobre o tempo efetivo de computação.

interferência A redução de desempenho que um novo processo impõe aos outros processos quando acessa recursos compartilhados.

distorção Na medida em que o número de etapas paralelas cresce, o tamanho médio de cada etapa diminui, mas a variância pode exceder em muito a média. O tempo de serviço de uma tarefa executada em paralelo é o tempo de serviço de sua etapa mais lenta. Quando a variância domina a média, um aumento no paralelismo pouco melhora o tempo de processamento.

Quando vários processadores são agrupados em paralelo, inevitavelmente há interferência entre suas ações e vários mecanismos de controle se fazem necessários para coordenar do conjunto de sistemas como um todo. Os requisitos de coordenação e interferência assumem várias formas nas diferentes arquiteturas paralelas. De uma forma geral, os requisitos de coordenação entre os processadores podem ser classificados em quatro categorias:

1. Serialização global das transações entre os nós
2. Coordenação e coerência entre os diversos *buffers*
3. Acessos a dados remotos ou acessos através de outros nós
4. Término de transações que envolvem atualizações de diversos nós

Estes requisitos podem tomar diferentes formas nas diversas arquiteturas existentes e possuem efeitos distintos sobre a eficiência de seu paralelismo. Ou seja, cada arquitetura possui um conjunto distinto de desafios técnicos e problemas para o processamento eficiente de transações em paralelo.

3.3 Arquiteturas Paralelas

Existem diversas abordagens, para o agrupamento de processadores, destinadas ao processamento paralelo de transações. Um método para o acoplamento de processadores é obtido através do compartilhamento de dados, onde todos os nós agrupados possuem acesso direto aos dados armazenados em disco. Esta abordagem é chamada de *arquitetura shared disks* [37, 24]. Outra técnica de agrupamento é realizada através do particionamento dos dados entre os diversos processadores, onde nenhum disco ou memória é compartilhada. Esta forma de agrupamento é conhecida como *arquitetura shared nothing* [54]. Nas próximas seções examinaremos melhor estas duas arquiteturas.

Além das arquiteturas *shared disks* e *shared nothing*, encontramos na literatura duas outras abordagens: as arquiteturas *shared intermediate memory* e a *shared everything*. A primeira abordagem caracteriza-se por introduzir um hardware especializado entre os processadores, que permite que estes tenham acesso a um *buffer* global compartilhado. Esta arquitetura é baseada no caso específico de controladoras de discos, que possuem mémoria local presentes em diversos *mainframes* do mercado, sendo específica para este ambiente em particular.

A arquitetura *shared everything*, por sua vez, caracteriza-se por compartilhar os discos e a memória principal de todos os processadores. Esta arquitetura, na verdade um multiprocessador fortemente acoplado, também não será examinada aqui devido a sua comprovada limitação de crescimento em escala. As principais origens das limitações desta arquitetura são os requisitos de desempenho da memória principal e coerência da memória *cache* dos processadores, entre outros.

3.3.1 Arquitetura Shared Disks

Na arquitetura *shared disks* (figura 3.1), todos os discos que compõem o banco de dados são compartilhados por todos os processadores. Cada processador possui a sua própria memória principal e executa uma instância particular do gerenciador de banco de dados distribuído. Isto permite que todos os processadores acessem e modifiquem qualquer porção do banco de dados.

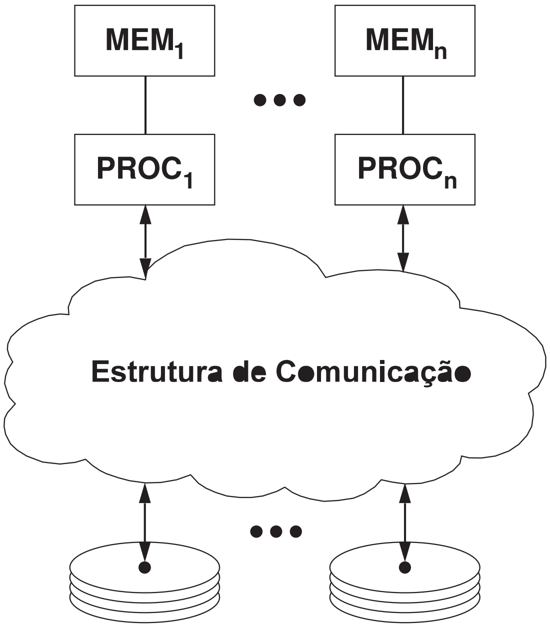

Figura 3.1: Arquitetura Shared Disks

Uma vez que cada processador possui acesso direto ao banco de dados, é necessário adotar mecanismos que disciplinem o acesso aos discos compartilhados. Para que os acessos aos dados sejam serializados, a presença de um mecanismo de controle de concorrência global se faz necessário. Com isto, evita-se conflitos em operações simultâneas sobre o mesmo dado por mais de um processador. Estes mecanismos são baseados em protocolos de *lock* global que podem ser centralizados ou distribuídos.

Cada instância do gerenciador de banco de dados possui o seu próprio *buffer pool*. Com isto, a arquitetura *shared disks* exige a presença de protocolos que assegurem a coerência do conteúdo destes. Quando um nó atualiza um dado, as cópias deste dado presentes em outros *buffers* locais tornam-se obsoletas e precisam ser invalidadas na conclusão da transação. Isto diminui a probabilidade de acerto dos *buffers*, além de proporcionar um uso menos eficiente dos *buffers* devido a grande quantidade de dados replicados entre estes.

Adicionalmente, os dados atualizados localmente necessitam ser gravados em disco antes do término da transação e os *locks* liberados. Estes fatores aumentam o tempo de bloqueio dos *locks* e, consequentemente, a probabilidade de contenção dos dados.

3.3.2 Arquitetura Shared Nothing

A arquitetura *shared nothing* (figura 3.2) caracteriza-se pela ausência de qualquer tipo de compartilhamento de memória entre os diversos processadores. Nesta abordagem, os nós se comunicam somente através de uma rede de comunicação e não dividem o acesso a nenhuma fração do banco de dados. Esta técnica é um grande contraste à abordagem por compartilhamento de dados, onde o banco de dados pode ser acessado diretamente por qualquer elemento.

Cada processador possui uma porção do banco de dados e somente esta porção pode ser diretamente manipulada por este nó do sistema. Isto é, o banco de dados é particionado entre os diversos processadores.

Como não há nenhum compartilhamento de dados, seja em termos de *buffer pool* ou de memória secundária, não há necessidade de se adotar mecanismos de sincronismo entre os processadores na arquitetura *shared nothing*. Isto permite que todos os *locks* sejam tratados localmente, em oposição à arquitetura *shared disks* que exige a manutenção de protocolos distribuídos de *locks* globais e de coerência de *buffer pools*.

Na arquitetura *shared nothing*, o acesso a dados que não estejam no disco local, é feito através de um pedido remoto ao nó proprietário do dado. Estes pedidos remotos

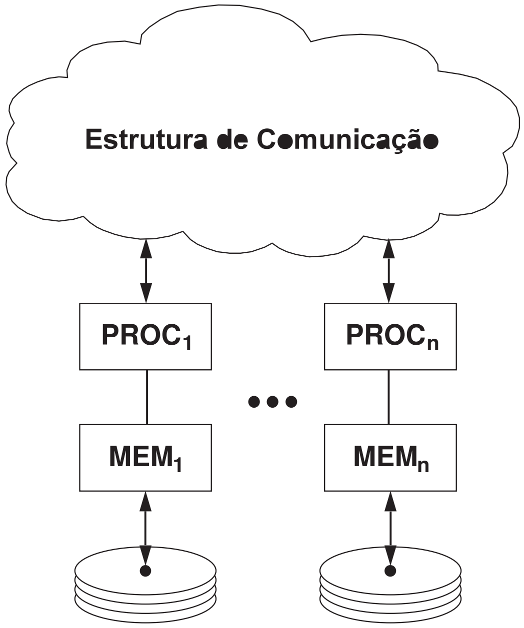

Figura 3.2: Arquitetura Shared Nothing

sem dúvida acrescentam uma sobrecarga no tempo de acesso ao dado não-local. No caso de operações de atualização em mais de um nó remoto, se faz necessário o uso de um protocolo de terminação (seção 2.6.2) para garantir a atomicidade da transação executada.

3.3.3 Comparações entre Arquiteturas

Na arquitetura *shared nothing*, existem custos adicionais decorrentes dos acessos remotos a dados não-locais. Adicionalmente, um protocolo de *two phase commit* se faz necessário. Uma vez que múltiplos nós participam da mesma transação, todos os nós envolvidos em atualizações de dados desta transação obrigatoriamente tem que fazer parte deste protocolo. Esta participação não envolve apenas troca de mensagens adicionais pela rede e carga nos processadores, mas requer principalmente operações síncronas de E/S em cada um dos nós envolvidos com atualização. O custo do processamento do *two phase commit* em ambientes distribuídos é muito elevado [49].

Na abordagem por compartilhamento de dados, um mecanismo global de controle de concorrência é necessário. A presença deste mecanismo ocasiona uma sobrecarga adicional decorrente da obtenção e liberação de *locks* globais. Isto é um contraste com a arquitetura *shared nothing*, onde o banco de dados é particionado e conseqüentemente todos os *locks* são locais ao nó proprietário do dado. Ou seja, na abordagem *shared nothing* todas as informações de *locks* são mantidas localmente na memória principal dos processadores.

Outro fator que influi muito no desempenho da arquitetura *Shared Disks* é a invalidação de *buffers*. Como todos os processadores têm acesso aos discos, é necessário ter certeza que um dado armazenado em um *buffer* local não foi invalidado por uma atualização no disco feita por outro processador. Isto é feito através de mecanismos de controle de coerência dos *buffers* locais. Existem diversas alternativas para esta validação, mas todas envolvem troca de mensagens adicionais na rede ou reduções na taxa de acerto do *buffer*.

Um dos métodos mais eficientes para o controle da coerência de *buffers* é integrá-lo ao mecanismo de controle de concorrência [14]. Neste caso, o que se tem na verdade é um mecanismo integrado de controle global de coerência e concorrência para ambientes de dados compartilhados. Esta abordagem aproveita-se do fato de que *locks* exclusivos acarretam a invalidação das cópias do dado em outros *buffers* e, com isso, obtém uma redução no número de mensagens que trafegam pela rede.

Por comparação, não há invalidação de *buffers* e nem mecanismos globais de controle de concorrência na arquitetura *Shared Nothing*.

Nos ambientes baseados em compartilhamento de dados, a política de gerenciamento dos *buffers* é descentralizada, uma vez que cada processador executa a sua própria instância do gerenciador de banco de dados distribuído. Na verdade, não há cooperação ou divisão de trabalho entre os gerenciadores. Se as transações são designadas aos processadores de uma maneira cíclica (*round robin*), sem explorar a afinidade existente entre as transações, cada *buffer* local conterá dados similares.

Havendo, portanto, grande ocorrência de replicação entre os *buffers* dos diversos nós.

Na arquitetura *shared nothing*, em contraste, cada nó armazena no *buffer* apenas os seus dados locais, não havendo portanto replicação de conteúdo. Mesmo se não levarmos em conta o efeito da invalidação, a expectativa de taxa de acerto dos *buffers* é menor nos ambientes baseados em compartilhamento de dados. Nestes ambientes, o *buffer* local armazena frações de todo o banco de dados, enquanto que na arquitetura *shared nothing* os dados armazenados localmente se referem apenas a uma partição do banco de dados. Isto faz com que a probabilidade de um determinado dado encontrar-se no *buffer* seja muito maior quando não há compartilhamento de dados.

O que distingue, em síntese, as arquiteturas *shared nothing* e *shared shared* é o compromisso entre processamento remoto com a sobrecarga adicional da rede, na primeira arquitetura, e processamento local com menor taxa de acerto nos *buffers* e os custos adicionais do controle global de concorrência, na segunda abordagem.

A interferência nos *buffers* locais da arquitetura *shared disks* cresce na medida que se introduzem novos processadores a esta arquitetura, o que limita o seu crescimento em escala [56]. Tais limitações, entretanto, não ocorrem na arquitetura *shared nothing*, onde se verificou a possibilidade da obtenção de um crescimento em escala aproximadamente linear no processamento *online* de transações e de consultas relacionais complexas [13, 55]. Em face destes resultados, não há muitas justificativas para investir-se nas complexas soluções de *hardware* e *software* exigidas pela arquitetura *shared disks*.

3.4 Grupamento por Afinidade

Em ambientes transacionais existem diversas aplicações e múltiplos conjuntos de dados. Estes conjuntos de dados seriam relações em um ambiente relacional, banco de dados físicos em banco de dados hierárquicos, classes de objetos nos ambientes orientados a objetos ou arquivos em um sistema de armazenamento simples. Dizemos que as transações iniciadas pelo mesmo programa de aplicação formam uma *classe de transações* [59]. Cada classe de transações pode possuir afinidade com certos conjuntos de dados. Mais precisamente, podemos dizer que a *afinidade* de uma classe de transações para um conjunto de dados é a fração deste conjunto que é referenciado por esta classe de transações.

Se classes de transações com alta afinidade com determinados conjuntos de dados são executadas no mesmo nó, uma alta taxa de acerto é alcançada nos *buffers* locais na arquitetura *shared disks*. De forma semelhante, na arquitetura *shared nothing*, se classes de transações são executadas nos mesmos nós que contém os dados com os quais têm afinidade, a sobrecarga do acesso remoto será reduzida. Ou seja, transações com afinidades similares devem ser agrupadas juntas. Chama-se de *agrupamento por afinidade* o processo de se particionar as transações em *grupos de afinidade*, de acordo com as suas referências ao banco de dados.

Nos ambientes baseados em compartilhamento de dados, somente é possível agrupar classes de transações, uma vez que todos os discos são compartilhados. Entretanto, na arquitetura *shared nothing*, o agrupamento por afinidade pode ser empregado de duas formas: para a alocação de conjuntos de dados ou para a designação de transações para os múltiplos nós.

A principal dificuldade do agrupamento por afinidade é particionar as classes de transações sobre os nós de forma a permitir um balanceamento de carga homogêneo e que satisfaça as restrições de afinidade. O particionamento de transações em grupos de afinidade torna-se acentuadamente mais difícil quanto maior o número de nós envolvidos. O que é um fator dificultante no uso desta técnica em ambientes massivamente paralelos.

Capítulo 4

Uma Proposta de Arquitetura Transacional Distribuída

4.1 Introdução

Neste capítulo descrevemos a arquitetura transacional distribuída, uma variação da arquitetura *shared-nothing* que conta com mecanismos adicionais de segurança por garantir maior disponibilidade ao armazenamento de dados. Esta arquitetura caracteriza-se por dividir as funções de um servidor em dois tipos de nós distintos: *nós de transação* e *nós de armazenamento*. O primeiro tipo de nó é responsável estritamente pelo controle das transações executadas no ambiente, enquanto que o outro tipo de nó servidor dedica-se exclusivamente ao armazenamento e controle de réplicas dos dados.

Visando diminuir a sobrecarga dos mecanismos de segurança no processamento de transações na ausência de falhas, a arquitetura baseia-se também em uma estrutura de comunicação centralizada e na classificação das transações de acordo com os requisitos de confiabilidade e durabilidade dos dados que manipulam. A comunicação centralizada viabiliza o uso de protocolos de terminação mais simples e eficientes, enquanto que a classificação de transações permite que a arquitetura adote mecanismos de controle e replicação diferenciados, adequando com maior precisão os recursos do ambiente aos graus distintos de segurança exigidos por cada classe de transações.

4.2 Estrutura de Comunicação

O ambiente transacional distribuído é baseado em uma estrutura central de comunicação que interliga diretamente todos os componentes do sistema. Assumimos também que o custo de transmissão de mensagens nesta estrutura é baixo. A abordagem centralizada com baixo custo de transmissão está presente em inúmeras tecnologias comerciais de comunicação de dados, tais como o Ethernet 10baseT, Token-Ring, FDDI ou ATM. Cada uma destas tecnologias pode conter uma estrutura central de comunicação, o que possibilita o seu uso no ambiente distribuído. Não havendo portanto imposição de equipamentos de comunicação especializados no ambiente distribuído, o que restringiria consideravelmente a sua aplicabilidade.

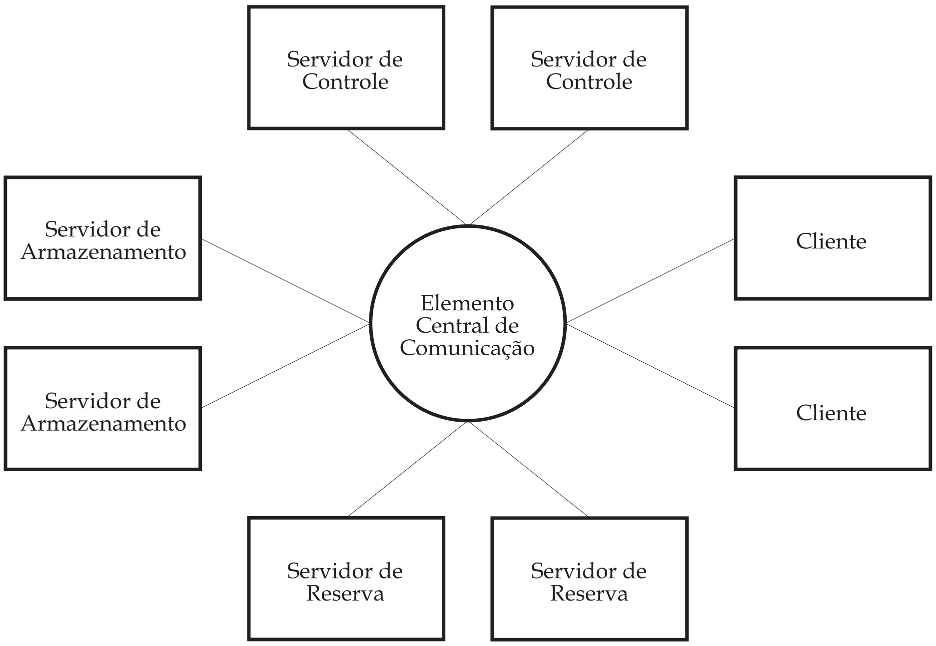

Figura 4.1: Arquitetura Transacional Distribuída

O elemento central de comunicação garante mais confiabilidade ao ambiente distribuído pois não permite a ocorrência de falhas de difícil contorno para os protocolos de terminação de transação. Estes protocolos não possuem meios de evitar o bloqueio de transações quando há falhas na rede de comunicação que causem a divisão dos processadores em duas ou mais partições isoladas [53]. Como mostraremos neste capítulo, a presença do elemento central de comunicação inibe o surgimento de partições dos nós do ambiente distribuído, o que garante uma disponibilidade superior ao processamento de transações.

4.3 Modelo de Falhas da Arquitetura

Nesta seção descrevemos o modelo de falhas (seção 2.2.2) da arquitetura transacional distribuída. Este modelo arbitra as classes de falhas que são toleradas pela arquitetura. Para estas falhas, o ambiente provê mecanismos de recuperação que levem o ambiente a um estado consistente após a ocorrência da falha, permitindo o prosseguimento normal do processamento de transações.

4.3.1 Falhas em Nós

Na ocorrência de uma falha em um nó do ambiente distribuído, o processamento deste nó é totalmente interrompido e todo o conteúdo da memória volátil é destruído. Neste caso, dizemos que o nó *falhou*.

No modelo de falhas do ambiente transacional distribuído, um nó sempre está funcionando corretamente ou encontra-se totalmente inoperante. Um nó nunca executa operações incorretas. Este tipo de comportamento é chamado de *fail-stop*, pois os nós apenas falham por interrupção completa em seu processamento.

Este modelo de falhas para nós certamente é uma idealização do comportamento das falhas que podem ocorrer em um nó. Um computador pode agir incorretamente na ocorrência de erros de *software* ou de *hardware*. Existem contudo diversas técnicas de depuração de software e de redundância de hardware que auxiliam na construção de equipamentos cujo comportamento em muito se aproximam ao modelo *fail-stop*. Estas técnicas fogem ao escopo deste trabalho e por simplicidade assumimos que os nós possuem comportamento *fail-stop*.

Seguindo a terminologia da seção 2.2.2, dizemos então que falhas *fail-stop* são falhas toleráveis e que falhas decorrentes de erros de *software* ou de *hardware* são intoleráveis em nosso modelo de falhas.

4.3.2 Falhas em Comunicação de Dados

Em um ambiente distribuído, mesmo assumindo que as falhas em nós sempre são *fail-stop*, diferentes nós podem estar operacionais ou não em um determinado instante. Dizemos que uma *falha parcial do sistema* é a situação onde alguns nós do ambiente distribuído estão operacionais e outros não. Uma *falha total do sistema* ocorre quando todos os nós estão inoperantes.

As falhas parciais em um ambiente distribuído são particularmente difíceis de tratar, pois os nós operacionais normalmente não sabem distinguir se a falha é na estrutura de comunicação ou em um outro nó do ambiente.

Por exemplo, se um nó A envia uma mensagem para outro nó B e não obtém nenhuma resposta de volta, então a rede pode estar defeituosa ou o nó B se encontrar inoperante. Enquanto a falha não for reparada, A não tem meios de saber o que realmente ocorreu. O grande problema desta incerteza é o fato de A não saber se B recebeu a mensagem e iniciou a execução de uma tarefa, fazendo alterações no seu estado interno, ou se a mensagem simplesmente não foi recebida por B.

No caso de um ambiente distribuído, particularmente em um ambiente baseado na arquitetura *shared-nothing*, uma transação pode atualizar dados em diversos servidores de armazenamento. A coordenação destas atualizações múltiplas é feita através de protocolos de terminação que são especialmente sensíveis a ocorrência de falhas parciais.

Os protocolos de terminação (seção 2.6) exigem que os nós participantes de uma transação se comuniquem com o nó coordenador da transação, que tem a responsabilidade de decidir o desfecho da transação baseando-se no voto de todos os nós participantes. Como o término da transação depende da decisão unânime de todos os participantes, a ocorrência de uma partição na estrutura de comunicação pode fazer com que alguns participantes ou o coordenador não tenham informações sobre o término da transação até obter o voto dos outros participantes.

Um participante P não pode decidir unilaterlamente se a transação terminou, confirmando ou não as suas alterações, pois esta decisão pode vir de encontro ao desfecho seguido pelos demais participantes da transação. Neste caso, uma decisão equivocada de P pode gerar inconsistências em seu banco de dados que podem ser propagadas para outras transações, ou gerar informação inconsistente para nós clientes.

4.3.3 Atomicidade na Comunicação de Dados

As dificuldades no tratamento de falhas parciais (seção 4.3.2) em um ambiente distribuído podem ser simplificadas se a estrutura de comunicação do ambiente assegurar que a comunicação com um nó servidor sempre é *atômica*. A comunicação atômica caracteriza-se por garantir que não haja comunicação com apenas uma parte dos nós do ambiente. Ou seja, ou todos os nós conseguem falar com um nó em particular ou nenhuma comunicação com este nó é possível.

Esta propriedade permite garantir que, se um nó servidor S não está disponível, o ambiente pode prosseguir com o processamento de transações nos outros nós sem se preocupar se S realmente está indisponível ou está isolado devido a uma falha de comunicação com S. Como qualquer tipo de comunicação com S é impossível, S e os demais nós têm certeza que nenhuma informação pode ser obtida de S. Isto elimina qualquer possibilidade de S propagar informação inconsistente para seus clientes, caso eventualmente tenha tomado uma decisão equivocada sobre o desfecho de alguma transação.

Quando os outros nós observam que não conseguem se comunicar com S, simplesmente assumem que S falhou. Posteriormente, assim que a comunicação for reestabelecida ou S se tornar operacional, os dados atualizados pelas transações processadas pelos outros nós podem ser aplicados em S normalmente, sem a possibilidade de inconsistências no banco de dados.

Mais precisamente, dizemos que a comunicação em um ambiente distribuído é atômica se as seguintes propriedades são asseguradas:

Sejam N_1, N_2 e N_3 nós de um ambiente distribuído D:

1. Se N_1 se comunica com N_2, então N_2 se comunica com N_1
2. Se N_1 se comunica com N_2 e N_2 se comunica com N_3, então N_1 se comunica com N_3.
3. Se N_1 não se comunica com N_2 e N_2 se comunica com N_3, então N_1 não se comunica com N_3.

Dizemos que os nós N_1 e N_2 encontram-se *disponíveis*, se N_1 consegue se comunicar com N_2.

Estas propriedades garantem que um nó N se comunica com todos os nós disponíveis de D e, portanto, é disponível, ou não se comunica com nenhum outro nó. Dizemos assim que a comunicação de dados em D é *atômica*.

4.3.4 Comunicação Atômica com Elemento Central de Comunicação

Um elemento de comunicação centralizado, por exemplo um concentrador de rede ou um comutador, normalmente é um equipamento que possui uma alta confiabilidade e um elevado MTBF – *Mean Time Between Failures*. Entretanto, esta característica não impede que ocorram falhas de comunicação no ambiente distribuído. A contribuição importante do elemento central para o ambiente distribuído é garantir a atomicidade da comunicação de dados.

Muitas tecnologias atuais de rede local exigem que cada nó se interligue diretamente a um elemento central, que pode ser um concentrador ou um comutador. Desta forma, a estrutura física da rede segue uma topologia em estrela. Se adotarmos um modelo de falhas bastante realístico que simplesmente isole defeitos pouco comuns de *hardware* para o elemento central de comunicação, a topologia em estrela garante que a comunicação entre os nós será atômica.

O modelo de falhas da estrutura de comunicação deve assumir que não haja um defeito que ao mesmo tempo impeça um nó N_1 de se comunicar com N_2 e permita a comunicação com N_3. Como na arquitetura transacional distribuída a estrutura de comunicação possui uma estrutura monolítica, apenas uma falha pouco comum de *hardware* provocaria um defeito desta natureza.

Na presença de uma estrutura central de comunicação C, dizemos que a comunicação entre os nós N_1 e N_2 é atômica se C garante as seguintes propriedades:

1. Se N_1 não consegue se comunicar com C, então N_2 não se comunica com N_1.
2. Se N_1 se comunica com C e N_2 se comunica com C, então N_1 se comunica com N_2.

4.4 Modelo de Replicação

O mecanismo de replicação empregado nesta arquitetura segue o modelo de cópia primária (seção 2.7.2). Neste modelo, apenas um servidor de armazenamento contém a cópia principal de um dado - chamada de *cópia primária* - enquanto que um outro servidor de armazenamento guarda a *cópia secundária* do mesmo dado. Todas as operações de leitura ou atualização em uma transação são realizadas apenas no servidor primário. A cópia secundária, por sua vez, é atualizada automaticamente pelo sistema e somente é manipulada por uma transação no caso de falhas no servidor primário.

A primeira vista, um mecanismo de replicação que não faça distinção entre os acessos permitidos às diversas cópias de um mesmo dado parece ser mais interessante do que a abordagem por cópia primária. Por exemplo, adotando-se um mecanismo *read one, write all* (seção 2.7.1), seria possível ler indistintamente as cópias de um dado, o que a principio resultaria em um melhor uso do espaço em memória estável e permitiria um balanceamento da carga entre os servidores que armazenam cópias de um mesmo dado.

A partir de estudos comparativos entre o desempenho de arquiteturas paralelas para o processamento de transações (seção 3.3.3), verificamos, entretanto, que este tipo de liberdade traria ao ambiente, que é baseado no modelo *shared nothing*, todos os pontos negativos do modelo *shared disks* (seção 3.3.3). Como um determinado dado estaria disponível em mais de um servidor, tal estratégia exigiria o uso de um oneroso mecanismo de controle de concorrência distribuído. Haveria também a necessidade de mecanismos adicionais que assegurassem a coerência dos *buffers* dos servidores que armazenassem cópias dos mesmos dados, o que elevaria o número de mensagens trocadas na rede e implicaria em uma taxa inferior de acerto nos *buffers* destes servidores. O custo destes mecanismos adicionais anulam o ganho potencial do acesso concorrente a cópias do mesmo dado.

Esta verificação é confirmada por estudos comparativos de desempenho de mecanismos de controle de concorrência em diferentes métodos de replicação de dados [9]. Estes estudos não observaram diferenças no desempenho do ambiente distribuído nas opções de modelos de replicação em questão, quando o custo de transmissão de mensagens é compatível com as tecnologias atuais de redes locais.

A escolha do modelo de cópia primária foi reforçada ainda pelas alternativas que este modelo oferece para a propagação das atualizações. Estas alternativas oferecem diferentes níveis de proteção aos dados manipulados por uma transação com custos proporcionais à proteção oferecida. Adicionalmente, as cópias secundárias deste modelo podem ser usadas por uma classe significativa de aplicações que não necessitam de mecanismos de sincronismos sofisticados [44].

Estas características do modelo de cópia primária, em adição à classificação de uma transação segundo os diferentes níveis de confiabilidade exigidos pelos dados que manipula, permite a obtenção de um melhor uso dos recursos do ambiente transacional distribuído. Esta questão é examinada em detalhes na próxima seção.

4.5 Níveis de Confiabilidade de uma Transação

4.5.1 Níveis de Confiabilidade

Dependendo da natureza da aplicação, os requisitos de confiabilidade e durabilidade de uma transação podem variar muito. Algumas aplicações exigem uma manipulação extremamente segura de seus dados, por exemplo aplicações financeiras ou médicas, enquanto que outras aplicações, tais como o cálculo de estatísticas ou previsões, fazem poucas exigências na integridade lógica dos dados. De acordo com as exigências de segurança de uma transação, o mecanismo de replicação de cópias secundárias presta serviços diferenciados às transações do ambiente, fornecendo a segurança adequada a necessidade da transação com economia dos recursos do sistema.

A partir de seus requisitos de confiabilidade e durabilidade, podemos classificar uma transação como *crítica* ou *normal*, de acordo com o modo que as cópias secundárias dos dados alterados pela transação são mantidas atualizadas. Uma transação é crítica quando manipula dados cujos requisitos de durabilidade e confiabilidade exigem que as suas cópias primária e secundárias sejam sempre perfeitamente atualizadas. As transações normais por sua vez operam dados com requisitos menores de segurança e as suas cópias secundárias podem ser atualizadas no momento mais apropriado, em termos de otimização de recursos, para os servidores de armazenamento.

O conceito de *pseudo-transação*, uma transação com exigências mínimas de confiabilidade e durabilidade, também é introduzido neste capítulo. Numa pseudo-transação, o cliente recebe dados diretamente de um ou mais nós servidores com intervenção mínima de um servidor de transação. Este tipo especial de transação permite que cópias secundárias de um dado sejam lidas sem os elevados custos de um controle de concorrência distribuído que seria necessário caso uma transação normal pudesse ter acesso aos dados secundários. O relaxamento das propriedades garantidas por uma pseudo-transação e a possibilidade do uso simplificado de cópias secundárias, torna mais eficiente o uso dos recursos do ambiente distribuído.

4.5.2 Transações Críticas

Muitas aplicações exigem que seus dados tenham grande confiabilidade. Exemplos destas aplicações são as que manipulam informações financeiras ou dados relativos a vida humana. Qualquer perda de informação nestes ambientes pode resultar em imensos prejuizos.

A possibilidade de perda em um ambiente distribuído com replicação de dados é sensivelmente reduzida se os dados primários forem atualizados simultaneamente com as suas réplicas, ou dados secundários. Nesta situação, a transação só termina após ambas as cópias terem sido fisicamente gravadas em disco.

Apesar de adotar um mecanismo conceitualmente simples de replicação de dados, as transações críticas exigem sincronismos adicionais entre os dados primários e secundários no decorrer da transação. Este sincronismo tem um custo de processamento elevado, o que resulta em uma sobrecarga adicional para o processamento das transações críticas, impactando o desempenho global do sistema.

4.5.3 Transações Normais

Nas aplicações cujos dados não possuam requisitos especiais de confiabilidade, o mecanismo de atualização das cópias secundárias pode ser relaxado, de modo a diminuir o seu custo de processamento por transação. Esta atualização pode ser feita em lotes envolvendo diversas transações, amortizando a sobrecarga dos mecanismos de sincronimos entre todas as transações do lote. Adicionalmente, a ausência de sincronismo entre as cópias primárias e secundárias diminui o tempo de processamento da transação, o que minimiza o tempo de retenção dos *locks* mantidos pela transação. Isto reduz a contenção de dados, consequentemente, elevando o desempenho global do sistema. Finalmente, o grande número de gravações em sequência de um lote de replicação torna possível o uso de técnicas de otimização no acesso a disco (por exemplo, otimização do movimento de cabeças de leitura/gravação) e a utilização do sistema de E/S com taxa de transferência plena.

4.5.4 Pseudo-Transações

Diversas classes de aplicações manipulam um grande volume de informações e não possuem grandes exigências de que os dados obtidos estejam instantaneamente atualizados. Aplicações de apoio a decisão, estatística, análise de dados, sumários, entre outras, são exemplos típicos deste tipo de manipulação. Outra característica importante destas aplicações é o fato que apenas realizam operações de consulta, não alterando, portanto, o estado do banco de dados.

Este tipo de aplicação não necessita de todos os mecanismos de proteção embutidos em uma transação. Neste caso, as propriedades asseguradas por uma transação representam um benefício muito inferior ao processamento adicional decorrente da execução de seus mecanismos de proteção. Os mecanismos de proteção necessários a estas classes de aplicação resumem-se a garantir a consistência física dos dados. A consistência lógica é dispensável uma vez que as aplicações não alteram o banco de dados e nem exigem que os dados estejam perfeitamente atualizados.

A consistência física dos dados é assegurada através de uma *pseudo-transação*. Uma pseudo-transação acessa os dados secundários dos servidores de armazenamento com uma interferência mínima dos servidores de transação que por sua vez se limitam a informar os servidores de dados disponíveis no início da pseudo-transação. Os demais acessos são realizados diretamente no servidor de armazenamento que por sua vez usa mecanismos simples baseados em *latches* para evitar conflitos no acesso aos dados.

4.6 Servidores de Armazenamento

Um servidor de armazenamento é responsável pelas operações de leitura e gravação de uma partição do banco de dados. Os servidores de armazenamento são organizados na forma *shared-nothing* que se caracteriza por não haver nenhum tipo de compartilhamento de dados entre os servidores.

Dizemos que um nó de armazenamento é *servidor primário* desta partição do banco de dados, quando os dados desta partição representam a cópia primária segundo o modelo de replicação da arquitetura (seção 4.4). Adicionalmente, um nó de armazenamento é responsável também pela manutenção de réplicas de dados primários de outros servidores. Dizemos ainda que um nó é *servidor secundário* dos dados primários de outros servidores que, por sua vez, constituem cópias secundárias destes dados.

Em uma transação crítica, os servidores de armazenamento atualizam as cópias primárias e secundárias dos dados alterados pela transação antes do seu término. Ou seja, o *commit* da transação só acontece após a gravação física das cópias primária e secundária do dado. Já em uma transação normal, apenas os dados primários são atualizados antes do término da transação. Isto é, o *commit* da transação depende apenas da gravação da cópia primária do dado. A atualização da cópia secundária neste caso é feita de forma assíncrona à transação, ocorrendo em lotes que visam diminuir a carga de trabalho dos nós de armazenamento, amortizando o custo do protocolo de terminação em um grupo de transações.

A ocorrência de uma falha em um servidor secundário antes que os dados atualizados por uma transação normal t tenham sido replicados neste servidor faz com que t seja abortada. Este aborto ocorre invariavelmente após o término (normal) de t e pode ferir a durabilidade da transação se o usuário não for informado da ocorrência do aborto. Como este tipo de falha é uma situação prevista no contexto da arquitetura transacional distribuída, uma aplicação mais sensível (porém não crítica) que use transações normais, pode optar por notificar o término da transação ao usuário apenas quando o ambiente sinalizar que os dados se encontram seguramente replicados.

4.6.1 Particionamento dos Dados

A arquitetura transacional distribuída usa uma estratégia simples para o particionamento dos dados entre os diversos servidores de armazenamento da arquitetura. Os elementos de dados são distribuídos ciclicamente entre os servidores de armazenamento segundo uma estratégia *round-robin*. Esta estratégia permite que os dados sejam distribuídos uniformemente entre os servidores de forma independente da semântica atribuída aos dados.

O particionamento dos dados é controlado pelos servidores de transação que atuam como intermediários da interação entre o cliente e o servidor de armazenamento na localização dos dados. Os pedidos dos clientes são encaminhados para um servidor de transação que designa o servidor de armazenamento responsável por fornecer ou atualizar o dado requisitado. Esta separação de funções permite que outras estratégias de distribuição de dados sejam implementadas em servidores de transações especializados em algum modelo específico de banco de dados (e.g. Relacional, Orientado a Objeto, etc.).

4.6.2 Confiabilidade dos Dados de um Nó de Armazenamento

A manutenção da consistência lógica e física dos dados é responsabilidade exclusiva do nó de armazenamento. Como os dados primários encontram-se estritamente particionados entre os nós e não há operações de escrita atuando diretamente sobre os dados secundários, o mecanismo de controle de concorrência na arquitetura transacional distribuída é local em cada servidor de armazenamento. Este tipo de mecanismo não exije a presença de controles de concorrência distribuídos, tipicamente complexos, que oneram significativamente o custo do processamento de transações em um ambiente com alto grau de distribuição.

A confiabilidade dos dados primários e secundários é mantida através do uso de *latches* e *locks*. Os *locks* são empregados para garantir a consistência lógica dos dados primários enquanto que os *latches* são usados para a manutenção da consistência física dos dados primários e secundários.

A consistência lógica de dados primários é mantida através de um mecanismo tradicional de *locks* baseado no algoritmo de *two-phase locking* estrito (seção 2.4.1.1). Os *locks* são mantidos pelo servidor de armazenamento que contém a cópia primária do dado. Como o banco de dados é particionado entre os servidores de armazenamento, isto é, apenas um nó contém a cópia primária do dado, todas as transações que desejam atualizar um fragmento de dados acessam o mesmo servidor. Isto permite que o mecanismo de *locks* seja local a cada servidor, uma vez que as informações sobre os *locks* só dizem respeito ao nó que possui a cópia primária do dado.

Como os dados secundários não são manipulados diretamente por uma transação, sua consistência lógica é automaticamente garantida pelos mecanismos de replicação, seja no caso de propagação assíncrona nas transações normais ou na propagação síncrona em transações críticas.

Devido ao desacoplamento das operações de atualização de réplicas do processamento de uma transação normal e da liberdade com que as pseudo-transações podem acessar dados secundários, diversas situações de conflito podem surgir, gerando inconsistência na manipulação dos dados. Para que estes conflitos sejam evitados, os servidores de armazenamento adotam um mecanismo simples de *latches*, que visa sincronizar o acesso aos dados secundários. Este mecanismo assegura que duas operações primitivas não acessam o mesmo dado simultaneamente.

Tanto os *locks* quanto os *latches* são mecanismos comuns para o controle de informação compartilhada. Os *latches*, ou semáforos, são estruturas de dados simples mantidas por períodos curtos de tempo, usualmente o tempo de uma operação de leitura ou atualização em um dado. Um *lock*, por ser responsável pela consistência lógica do dado, está necessariamente vinculado a uma transação. Normalmente possui duração bem superior a de um *latch*, usualmente é mantido até o término da transação. Como uma transação pode manter um número elevado de *locks*, que podem perdurar por toda existência da transação, estes obrigatoriamente são gravados em estruturas de dados mais complexas do que as exigidas pelos *latches*. Adicionalmente, os *locks* exigem mecanismos complexos de detecção de *deadlocks* que não são necessários em *latches*.

4.6.3 Gerência de Buffers

Cada servidor de armazenamento possui um mecanismo de gerência de *buffers* que é responsável pela interface entre as cópias de dados nos *buffers* em memória principal e em armazenamento estável.

O mecanismo de gerência de buffers usualmente está intimamente ligado ao mecanismo de recuperação de falhas. Normalmente, a recuperação de falhas impõe restrições às políticas do gerente de buffers que influenciam consideravelmente o desempenho das operações de E/S. Existe, na verdade, um compromisso entre favorecer o desempenho do processamento normal (sem falhas) de transações e o tempo de recuperação de falhas.

Como a arquitetura transacional distribuída tem como premissa otimizar o desempenho da operação do ambiente na ausência de falhas, assume-se que o mecanismo de gerência de buffers adota a política *steal, no force* [25]. Esta política permite que a propagação de elementos dos buffers em memória principal seja totalmente independente do processamento de transações, não exigindo que os elementos alterados permaneçam no *buffer* ou que sejam gravados em armazenamento estável ao término da transação. Com esta liberdade, o gerente de *buffers* pode cumprir melhor o seu papel, voltando-se para aumentar a concorrência, diminuir o volume de operações físicas de E/S e fazer um uso mais eficiente dos *buffers* em memória principal.

A independência das políticas de gerência de buffer em relação aos compromissos de durabilidade de uma transação tem o efeito colateral de exigir um mecanismo de recuperação de falhas mais complexo. Como usualmente as falhas ocorrem pouco frequentemente, uma vez que os equipamentos atuais possuem um alto *MTBF*, este efeito colateral é perfeitamente aceitável.

4.7 Servidores de Transação

Os servidores de transação são responsáveis pelo controle das transações executadas no ambiente transacional distribuído. Este tipo de nó realiza as seguintes tarefas:

1. Coordenação de atualizações em múltiplos nós de armazenamento através de protocolos de terminação;
2. Coordenação de atualizações de cópias secundárias em transações críticas;
3. Armazenamento em *log* das informações de controle de uma transação;
4. Atua como coordenador da transação em nome do nó cliente.
5. Redirecionamento de tarefas para o servidor secundário no caso de falhas do servidor primário

Os servidores de transação atuam como intermediários no relacionamento dos clientes com os servidores de armazenamentos. As requisições dos clientes sempre são enviadas aos servidores de transação que, por sua vez, direcionam o pedido ao servidor de armazenamento adequado. Por estarem conectados a uma estrutura de comunicação única, o tráfego de dados não sofre a mediação dos servidores de transação, fluindo diretamente entre o cliente e os servidores de armazenamento. Apenas as informações de controle são intermediadas pelo servidor de transação. As mensagens que transportam dados quando necessitam do conhecimento do servidor de transação, são simplesmente difundidas através do mecanismo de *broadcast* presente na estrutura de comunicação de dados. Isto evita o atraso do roteamento dos dados através do servidor de transação.

Os servidores de transação tem um importante papel no mecanismo de replicação. Nos casos de falha em um servidor de armazenamento, como o servidor de transação atua como intermediário das requisições dos clientes aos servidores de armazenamento, os pedidos são redirecionados automaticamente ao servidor secundário, transparentemente para o cliente.

De forma semelhante aos dados dos servidores de armazenamento, os registros das transações armazenadas nos servidores de transação possuem um mecanismo de replicação próprio, que visa proteger estes registros de eventuais falhas no servidor de transação. O principal objetivo do mecanismo de replicação é evitar o bloqueio de transações no caso de uma eventual falha em um servidor de transação S_T no momento em que os servidores de armazenamento já se encontram compromissados com o término da transação e ainda não receberam a decisão final de S_T. Nesta situação, como os registros das transações de S_T encontram-se replicados em outro servidor de transações $S_{T'}$, os servidores de armazenamento conseguem respostas sobre o desfecho da transação através de uma simples consulta a $S_{T'}$.

4.8 Clientes

Os nós cliente são aqueles que iniciam uma transação e comandam, com o intermédio do servidor de transação, as operações de leitura e atualização sobre os dados dos nós de armazenamento. As aplicações são executadas nos clientes que requisitam serviços dos servidores de transação, no caso de transações normais ou críticas, ou manipulam diretamente o servidor de armazenamento no caso de pseudo-transações.

Em um ambiente distribuído real, os requisitos de confiabilidade de um nó cliente são inferiores aos dos servidores. Normalmente os servidores (de transação ou de armazenamento) são máquinas em ambientes controlados, no qual existem mecanismos rotineiros que garantem a confiabilidade e disponibilidade dos servidores. De outra forma, os cliente podem ser qualquer tipo de máquina, inclusive situadas em localidades remotas, sobre as quais se tem pouco ou nenhum controle.

Em um ambiente distribuído convencional, os clientes possuem a função de iniciar a transação e, adotando-se protocolos convencionais de terminação de transação, estes também assumem o importante papel de coordenador da transação. Como a ocorrência de falhas no coordenador acarreta grandes impactos no processamento dos nós envolvidos na transação, bem como no funcionamento de outras transações, não é aconselhável que o coordenador da transação seja atribuído ao cliente por sua baixa confiabilidade.

Na arquitetura transacional distribuída, o papel de coordenador de transação é exercido pelos servidores de transação. A presença destes servidores especializados e de uma estrutura central de comunicação, torna dispensável o uso de protocolos de migração de coordenação especializados [48]. Estes protocolos só são necessários quando o processo de migração passa por vários nós intermediários e existem várias alternativas para o nó que receberá o controle da transação. Na arquitetura transacional distribuída, o servidor de transação é o candidato natural ao coordenador, não existindo assim alternativas para a migração. Adicionalmente, a estrutura central de comunicação provê a comunicação direta entre o cliente e o servidor de transação, sem a presença de nós intermediários.

4.9 Servidores de Reserva

A arquitetura transacional distribuída conta com servidores de reserva que entram em atuação no caso de falhas em servidores de armazenamento. Como a estratégia de particionamento dos dados exige que se tenha um número fixo de servidores em atuação, é importante para a disponibilidade do ambiente que hajam servidores prontos para entrar em ação na ocorrência de falhas em um servidor de armazenamento. Tal exigência não ocorre com os servidores de transação. A falha em um servidor de transação, não obstante comprometa as transações que estejam sendo processadas pelo servidor no momento da falha, não afeta a disponibilidade do ambiente como um todo, uma vez que novas transações podem ser designadas livremente para qualquer outro servidor de transações.

Na ocorrência de uma falha em um servidor de armazenamento, o servidor de reserva entra em atuação imediatamente após a falha ter sido detectada e assume temporariamente as funções do servidor que falhou. No momento da entrada em operação do servidor de reserva, ocorre uma inversão de papéis na distribuição de cópias primárias e secundárias entre os servidores de armazenamento, de tal forma que o servidor de reserva atue somente como servidor secundário. A função do servidor de reserva, em primeira instância, é o de registrar as atualizações que estão ocorrendo enquanto o servidor de armazenamento que falhou não se recuperar da falha.

Assim que o servidor de armazenamento estiver novamente consistente, o servidor de reserva envia os dados modificados para o servidor de armazenamento que, após estar totalmente atualizado, reassume as suas funções normalmente. Caso o servidor de armazenamento não possua meios de se recuperar, na ocorrência de uma falha que impeça totalmente a sua operação, o servidor de reserva passa então a receber cópias dos dados dos outros servidores e assume o papel definitivo de servidor de armazenamento quando os seus dados estiverem totalmente atualizados.

4.10 Servidor de Diretório

Ao iniciar uma transação, um nó cliente necessita localizar um nó de transação e direcionar os seus pedidos a este servidor. Para que haja critérios na localização de um servidor de transação por um cliente, faz-se necessário a presença de um servidor de diretório. Este servidor contém informações sobre a carga de trabalho e a disponibilidade dos servidores de transação e decide que servidor será alocado para aquela transação.

Uma estratégia pertinente para a alocação de servidores de transação é basear-se nas relações de afinidade entre aplicações, dados e grupos de usuários (seção 3.4). Esta estratégia pode ter grande influência no desempenho global do ambiente paralelo [59]. Neste trabalho, por simplicidade, optou-se por usar uma estratégia simples de balanceamento de carga baseada no número de transações processadas por um servidor de transação. Quando um nó cliente deseja iniciar uma nova transação, o servidor de diretório aloca o servidor de transações que estiver processando o menor número de transações naquele instante.

A função de servidor de diretório pode ser exercida por um servidor de transação ou por um nó dedicado a esta função. O uso de um nó exclusivamente para o servidor de diretório é relevante para o caso de alocação por relações de afinidade que demandam grandes recursos computacionais para o seu processamento. No caso de estratégias simples, tais quais a de balanceamento de carga empregada neste trabalho, tal distinção não é necessária.

O uso de um servidor dedicado para os serviços de diretório também é relevante nos casos onde os serviços distribuídos são realizados através de um ambiente integrado de computação distribuída que possua serviços estruturados de diretório em sua composição, como por exemplo o DCE [40]. Neste caso, a designação do servidor de transação a ser alocado pelo cliente seria feita nos mesmos moldes dos demais serviços distribuídos do ambiente.

Capítulo 5

Mecanismo de Recuperação e Replicação Transacional Distribuído

5.1 Introdução

Neste capítulo apresentamos o mecanismo de recuperação e replicação da arquitetura transacional distribuída. Este mecanismo tem o objetivo de assegurar as propriedades de atomicidade e durabilidade das transações, provendo recursos adicionais para assegurar a continuidade do processamento de transações na ocorrência de falhas em um nó do ambiente, em particular de um servidor de armazenamento. O mecanismo de recuperação compreende as operações de recuperação local dos servidores de armazenamento e as responsáveis pela manutenção do sincronismo entre as cópias primárias e secundárias dos dados armazenados.

A recuperação local de falhas de cada servidor de armazenamento é baseada no ARIES [34], um mecanismo de recuperação fundamentado no difundido protocolo *write-ahead logging* (seção 2.5.2.2). Uma das características mais interessantes do ARIES é o grau de independência que possui do mecanismo de gerência de *buffers*. Estes último, por sua vez, pode adotar políticas mais adequadas para o desempenho do ambiente, deixando inteiramente para o mecanismo de recuperação a responsabilidade de assegurar a consistência dos dados armazenados. Esta independência é garantida por um conjunto de estruturas de dados que, no contexto da arquitetura transacional distribuída, foi estendida para dar suporte aos mecanismos de sincronismo entre as cópias primárias e secundários dos dados dos servidores de armazenamento. Adicionalmente, o ARIES oferece suporte a *rollbacks* parciais de transações, também incorporado à arquitetura distribuida com suporte complementar a replicação de dados.

A manutenção da consistência entre as cópias primárias e secundárias dos dados dos servidores de armazenamento também é responsabilidade do mecanismo de recuperação e replicação. Este mecanismo se baseia em técnicas de recuperação de desastres, que prevêem o suporte a servidores *backups* remotos, prontos para entrar em operação na ocorrência de falhas do servidor principal [38, 43, 44, 19]. Tais técnicas empregam o modelo de cópia primária de replicação (seção 2.7.2), onde a propagação de réplicas segue o método *1-safe*, que atualiza os dados secundários após o término da transação. O mecanismo de recuperação e replicação, por sua vez, estende estas propostas apresentando um método de propagação de réplicas híbrido que reune as técnicas *1-safe* e *2-safe*. Também em oposição às técnicas de recuperação de desastres, que assumem que o servidor *backup* é um nó remoto, o mecanismo aqui proposto considera que todos os servidores são locais e estão interligados por uma estrutura de comunicação de alto desempenho e confiabilidade. Esta premissa garante um maior grau de acoplamento entre os servidores primários e secundários, permitindo assim um melhor desempenho e simplicidade no mecanismo de recuperação e replicação.

O sincronismo entre as cópias primárias e secundárias é feito de duas formas distintas, conforme o nível de segurança da transação que atualiza os dados. No caso de transações críticas, o sincronismo segue a técnica *2-safe*, sendo embutido na operação de atualização de elemento de dados que *per se* garante que as cópias primária e secundária de suas atualizações se encontram perfeitamente sincronizadas. No caso das transações normais, a atualização dos dados secundárias é feita em termos de lotes de transações por um mecanismo independente, após o término da transação, nos moldes da técnica *1-safe*. Ambas estratégias são descritas neste capítulo.

O mecanismo de recuperação e replicação também tem a função de garantir a disponibilidade dos dados do ambiente distribuído, mesmo na ocorrência de falhas em um servidor de armazenamento. Isto é feito através da coordenação da atuação do servidor de reserva que entra em operação automaticamente na ocorrência de uma falha em um servidor de armazenamento. Este mecanismo, apesar de abortar algumas transações em execução no momento da falha, permite a continuidade no processamento de transações críticas de forma ininterrupta. No caso de transações normais há, contudo, uma breve interrupção no momento em que o servidor de armazenamento, recuperado da falha, reassumir as suas funções habituais.

Mais especificamente, neste capítulo detalhamos o mecanismo de recuperação e replicação, explicitando as estruturas de dados, as ações adicionais embutidas nas operações de leitura e gravação de elemento de dados, de controle sobre transações, de recuperação de falhas e de propagação de atualizações para dados secundários.

5.2 Estruturas de Dados

A arquitetura transacional distribuída emprega diversas estruturas de dados para o controle de transações, replicação de dados e de auxílio na recuperação de falhas. Parte destas estruturas está presente nos servidores de transação, enquanto as demais localizam-se nos servidores de armazenamento. Ambas classes de servidores contam com *logs* para registrar o andamento do processamento de transações. Os *logs* destes servidores apresentam algumas diferenças em seus formatos. Estas diferenças entre os formatos dos *logs*, bem como as demais estruturas de dados da arquitetura transacional distibuída, são descritas a seguir.

5.2.1 Identificador de Transação

Uma transação é identificada univocamente por um *identificador de transação* - IDT. O IDT é consitutído por um par ordenado formado pelo servidor de transações S_T e um índice I_T. O índice I_T é um número sequencial gerado localmente por cada servidor de transação que é incrementado a cada transação que se inicia.

$$\boxed{S_T}\,\boxed{I_T}$$

Figura 5.1: Identificador de uma transação

5.2.2 Elemento de Armazenamento

Os servidores de armazenamento guardam os seus dados na forma de *elementos de armazenamento*. As operações de leitura e gravação dos servidores de armazenamento são realizadas apenas sobre elementos de armazenamento, lendo ou gravando um elemento por operação. Um elemento de armazenamento é portanto a unidade mínima de armazenamento e recuperação do ambiente transacional distribuído.

Cada elemento de armazenamento possui um número sequencial E que o identifica univocamente. Este número sequencial permite também informar em qual dos N servidores de armazenamento A_i se encontra a cópia primária do elemento, bem como o servidor de armazenamento A_j que guarda a cópia secundária.

As cópias primárias e secundárias dos elementos de dados são distribuídas ciclicamente entre os N servidores de armazenamento segundo o seu número E. As cópias primárias são divididas segundo um mecanismo simples de *round-robin* entre os servidores, onde o índice i de servidor A_i de um elemento de armazenamento E é dado pela seguinte fórmula:

$$i = ((E - 1) \bmod N) + 1 \tag{5.1}$$

As cópias secundárias são por sua vez dispersas entre os N servidores de armazenamento de forma cíclica em relação ao servidor A_i que armazena a cópia primária de E. O índice j que identifica o servidor secundário A_j de E é dado pela seguinte fórmula:

$$j = (E + ((E - 1) \div N) \bmod (N - 1)) \bmod N + 1 \tag{5.2}$$

Por exemplo, se considerarmos um ambiente com 4 servidores de armazenamento, a disposição das cópias primárias e secundárias será dada pela tabela 5.1. Note que para o elemento 1, o servidor primário é A_1 e o secundário é o A_2, enquanto que para o elemento 5, cujo servidor primário também é A_1, o servidor secundário é o A_3. Ou seja, há um rodizio entre os servidores secundários para um mesmo servidor primário.

Elemento	Primário	Secundário
1	1	2
2	2	3
3	3	4
4	4	1
5	1	3
6	2	4
7	3	1
8	4	2
9	1	4
10	2	1
11	3	2
12	4	3
13	1	2

Tabela 5.1: Exemplo de distribuição com 4 servidores de armazenamento

Os campos presentes em um elemento de armazenamento relevantes à Arquitetura Transacional Paralela são:

LSN	IDT	Tipo	Conteúdo

Figura 5.2: Elemento de armazenamento

LSN	Índice do registro do log local da última atualização da Página
ID	Identificador da transação responsável pela última atualização
Crítico	Indica se os dados da página são críticos ou não
Conteúdo	Dados armazenados na página

O campo Crítico indica se o elemento de dados foi atualizado por uma transação crítica. A arquitetura transacional distribuída impõe que uma transação crítica acesse apenas elementos de armazenamento atualizados por transações críticas. Como os elementos de armazenamento atualizados por transações normais não possuem garantias de confiabilidade até que suas réplicas sejam propagadas, esta restrição impede que uma transação crítica manipule dados com requisitos inferiores de confiabilidade.

5.2.3 Log Principal

Os servidores de armazenamento e de transação se utilizam de *logs* para registrar o andamento do processamento das transações em memória não volátil. O *log* principal, ou simplesmente *log*, armazena as informações sobre transações relevantes para a recuperação de falhas e as mudanças de estado impostas pelo protocolo de terminação. Adicionalmente, o *log* do servidor de armazenamento contém as alterações incrementais realizadas no banco de dados que também são relevantes para a recuperação de falhas.

Os *logs* dos servidores de armazenamento e de transação compartilham o mesmo formato básico para os seus registros (figura 5.3):

LSN	IDT	Tipo	LSN_Ant	Complemento

Figura 5.3: Registro do Log

Os campos presentes no registro básico do *log* são:

LSN Endereço do primeiro byte do registro no *log*. Na verdade, o LSN não é armazenado no log, ele é citado aqui apenas para facilitar o entedimento do mecanismo de *log*.

IDT Identificador da transação que efetou o registro

Tipo Tipo do registro no *log*

LSN_ANT Endereço do registro anterior da mesma transação. Este campo possui valor zero nos registros de início de transação

Complemento Campo de formato váriavel usado nos tipos de registro que necessitem de informações adicionais.

Este formato básico suporta os diferentes registros usados pelos servidores de transação e armazenamento. O campo Tipo, como o nome indica, informa o tipo do registro do *log*. Como existem grandes diferenças entre os diversos tipos de registros, o campo Complemento é usado pelos tipos de registros que necessitam armazenar informações adicionais às presentes no registro básico do *log*.

Os *log* dos servidores de transação e dos servidores de armazenamento possuem os seguintes tipos de registros:

INÍCIO Marca o início de uma transação.

ATUALIZAÇÃO Registra uma operação de atualização em um elemento de armazenamento. Este tipo de registro é exclusivo do *log* dos servidores de armazenamento.

PREPARAÇÃO Este tipo de registro também só é usado pelos servidores de armazenamento. Ele assinala que o servidor entrou no estado de *preparado* para o término normal de uma transação

COMPENSAÇÃO Este tipo de registro guarda as atualizações desfeitas de uma transação. Este registro ocorre apenas nos servidores de armazenamento

COMPROMISSO Indica que o servidor está *comprometido* com o término normal da transação. Este registro só está presente nos servidores de transação

TÉRMINO Indica o término normal da transação.

ABORTO Informa que a transação foi abortada

SEGURO Registra que os dados atualizados por uma transação foram replicados em um determinado servidor. Este registro é usado apenas pelo *log* dos servidores de armazenamento.

Os registros do tipo INÍCIO e ABORTO tem formato idêntico nos servidores de transação e de armazenamento. Os registros do tipo INÍCIO possuem em seu complemento o campo Crítico, que diz se a transação possui requisitos críticos de confiabilidade, e o campo Cliente que contém o endereço do cliente da transação (figura 5.4). De outro modo, os registros do tipo TÉRMINO e ABORTO não necessitam de nenhuma informação que não esteja presente no registro básico, dispensando assim o uso do campo Complemento.

O registros do tipo COMPROMISSO e TÉRMINO só aparecem nos servidores de transação. De outra forma, os registros dos *logs* do tipo ATUALIZAÇÃO e SEGURO

LSN	IDT	Tipo	LSN_Ant	Crítico	Cliente

Figura 5.4: Registro do tipo INÍCIO

estão presentes somente nos servidores de armazenamento. As diferenças no formato dos registros do tipo COMPROMISSO e os tipos de registro exclusivos de cada classe de servidores são apresentadas nas próximas seções.

5.2.4 Log do Servidor de Transações

O *log* dos servidores de transações registra o andamento das transações gerenciadas pelo próprio servidor. Os registros do tipo COMPROMISSO destes servidores apresentam um complemento nas transações normais (figura 5.5) que indica os servidores de armazenamento envolvidos na transação que está sendo comprometida. Esta informação é usada para controle da propagação de réplicas, indicando quais servidores de armazenamento possuem elementos de armazenamento com réplicas desatualizadas ao término da transação.

Adicionalmente, os registros do *log* dos servidores de transação indicam o término da transação através do registro do tipo TÉRMINO. Nas transações críticas, este registro é gravado logo após os servidores de armazenamento confirmarem que estão compromissados com a transação. De outra forma, nas transações normais, o registro de TÉRMINO é gravado no *log*, assim que as réplicas dos dados atualizados pela transação foram propagadas.

$$\boxed{Serv\_Arm_1} \dots \boxed{Serv\_Arm_n}$$

Figura 5.5: Complemento de Compromisso

O complemento do registro do tipo COMPROMISSO de um servidor de transações é composto pelos seguintes campos:

$A_1 \ldots A_n$ Lista de Servidores de Armazenamento que atualizaram elementos de armazenamento no decorrer da transação

5.2.5 Log do Servidor de Armazenamento

O *log* de um servidor de armazenamento, por registrar informações de uma transação restritas ao contexto do servidor e também as atualizações e operações desfeitas pelo servidor, possui algumas diferenças em sua estrutura quando comparado ao *log* de um servidor de transação. Estas diferenças são examinadas nesta seção.

5.2.5.1 Complemento de Atualização

Nos complemento dos registros do tipo ATUALIZAÇÃO, são armazenadas as imagens anterior e posterior de um elemento de dados (figura 5.6). A imagem anterior é usada para desfazer as atualizações das transações abortadas. A imagem posterior, por sua vez, é empregada para que transações que terminaram normalmente possam ser refeitas no caso de falhas no servidor de armazenamento.

Neste complemento, também aparecem o campo LSN_Elemento. Este campo armazena o LSN da última atualização do mesmo elemento de dados na mesma transação. Este campo é usado para o controle da replicação de elementos na presença de *rollbacks* parciais.

Elemento	Dados_Ant	Dados_Nov	LSN_Elemento

Figura 5.6: Complemento de atualização

Elemento Identificador do elemento atualizado

Dados_Ant Imagem dos dados anterior a atualização

Dados_Nov Imagem dos dados posterior a atualização

LSN_Elemento Ponteiro para o LSN da última atualização do elemento na mesma transação. Assume o valor zero na primeira atualização do elemento na transação

5.2.5.2 Complemento de Compensação

Quando uma transação é abortada, as operações responsáveis por desfazer as alterações realizadas pela transação também precisam ser registradas no *log* do servidor de armazenamento. Isto é feito através dos registros do tipo COMPENSAÇÃO. Este tipo de registro possui um complemento que armazena a identificação do elemento de armazenamento, a imagem do elemento após a alteração desfeita e um ponteiro para o próximo registro a ser desfeito. (figura 5.7). A imagem posterior não é necessária, pois este registro só é usado para se refazer uma transação durante o reinício do sistema após a ocorrência de uma falha.

Elemento	Dados_Nov	LSN_Desfazer

Figura 5.7: Complemento de compensação

Elemento Identificador do elemento atualizado

Dados_Nov Imagem dos dados posterior a operação desfeita

LSN_Desfazer Ponteiro para o próximo LSN a ser desfeito

5.2.6 Log Secundário

Além do *log* principal, os servidores de armazenamento usam um *log secundário* para registrar a propagação das atualizações dos dados secundários em transações normais. Este *log* mantém informações sobre o início de um novo grupo de propagação, as atualizações realizadas e as atualizações desfeitas.

O *log* secundário dos servidores de armazenamento segue o seguinte formato básico (figura 5.8):

LSN	Grupo	Tipo	LSN_Ant	Complemento

Figura 5.8: Registro do Log Secundário

Os campos presentes no registro básico do *log* são:

LSN — Endereço do primeiro byte do registro no log. Na verdade, o LSN não é armazenado no log, ele é citado aqui apenas para facilitar o entedimento do mecanismo de *log* secundário.

Grupo — Identificador da Grupo que efetou o registro

Tipo — Tipo do registro no *log*

LSN_Ant — Endereço do registro anterior do mesmo grupo. Este campo possui valor zero nos registros de início de grupo.

Complemento — Campo de formato váriavel usado nos tipos de registro que necessitem de informações adicionais em seu registro.

Este formato básico suporta os diferentes registros usados pelos servidores de transação. O campo Tipo, como no *log* principal, informa o tipo do registro do *log* secundário. Igualmente, o campo Complemento é usado pelos tipos de registros que necessitam armazenar informações adicionais às presentes no registro básico do *log* secundário.

Os *logs* secundários dos servidores de transação possuem os seguintes tipos de registros:

INÍCIO	Marca o início de um novo grupo de propagação e também o término do grupo de propagação anterior
ATUALIZAÇÃO	Registra uma operação de atualização em um elemento de armazenamento.
COMPENSAÇÃO	Este tipo de registro guarda as atualizações desfeitas de uma transação. Este registro ocorre apenas nos servidores de armazenamento
ABORTO	Informa que o grupo de propagação foi abortado

O registro do tipo INÍCIO marca o término de um grupo e também o início de um novo grupo de propagação de atualização. O registro de ABORTO indica que o grupo atual de propagação foi abortado. Estes dois tipos de registro não possuem COMPLEMENTO. Os registros de ATUALIZAÇÃO e COMPENSAÇÃO, por sua vez, seguem as definições dos registros de mesmo nome do *log* principal dos servidores de armazenamento. Estes registros são descritos, respectivamente, nas seções 5.2.5.1 e 5.2.5.2.

5.2.7 Tabela de Elementos Voláteis

Um elemento de armazenamento é considerado *volátil* se a versão deste elemento armazenada no *buffer* possui atualizações que ainda não estão refletidas na versão estável do mesmo elemento. Os servidores de armazenamento contam com uma tabela de elementos voláteis que guarda informações sobre os elementos cujas versões em memória estável não estão atualizadas. (Tabela 5.9). Quando uma operação de atualização modifica um elemento de armazenamento, esta alteração é feita apenas na versão em *buffer* do elemento. Uma entrada para o elemento modificado é então inserida na T_{EV}, contendo o identificador do elemento e o LSN do registro da atualização do elemento no *log*. O LSN armazenado indica o ponto inicial no *log* que contém modificações para o elemento de armazenamento. Quando a versão em disco de um elemento modificado é atualizada pelo mecanismo de gerência de *buffers*, a entrada do elemento é removida da tabela de elementos voláteis.

Esta tabela é usada na operação de reinício de um servidor de armazenamento que recupera o estado do servidor após a ocorrência de uma falha. Para que a operação de reinício seja mais eficiente, o que diminui o tempo de indisponibilidade do servidor de armazenamento, a tabela de elementos voláteis é salva em armazenamento estável a cada *checkpoint* do servidor de armazenamento.

Elemento	LSN_Alt

Figura 5.9: Tabela de Elementos Voláteis

5.2.8 Tabela de Elementos Não Replicados

As atualizações realizadas por uma transação normal só são refletidas nos servidores secundários após o término da transação. A relação de páginas alteradas por uma transação normal que ainda não foram replicadas é armazenada na *tabela de elementos não replicados* - T_{NR}. Cada item da T_{NR} (Figura 5.10) contém o identificador da transação que alterou o elemento, o identificador do elemento e o LSN do registro do *log* que efetuou a alteração.

A T_{NR} é usada na propagação de réplicas para apontar o registro do *log* que contém a última versão do elemento. Esta informação é importante, pois evita que se façam buscas exaustivas no *log* pela versão mais atual do dado, em meio a múltiplos registros de atualização ou *rollbacks* parciais para o mesmo elemento. Como a T_{NR} aponta para o registro do *log*, o elemento de armazenamento pode ser manipulado normalmente enquanto a réplica não for propagada. Adicionalmente, este mecanismo garante que apenas uma cópia de cada elemento de armazenamento será propagada em cada grupo de propagação, o que diminui sensivelmente o número de acessos a discos e troca de mensagens. Esta otimização é particularmente útil para dados que são atualizados frequentemente, ou *hot-spots*.

IDT	Elemento	Último_LSN

Figura 5.10: Tabela de Elementos não Replicados

Os campos da T_{NR} são:

IDT Identificador da transação que alterou o elemento

Elemento Identificador do elemento alterado

Último_LSN LSN da última atualização do elemento na transação

5.2.9 Tabela de Transações Pendentes

Cada servidor de transações usa uma tabela de transações, T_T, que contém informações das transações pendentes. Uma transação é considerada *pendente* enquanto não tiver terminado e os seus dados estarem replicados no servidor secundário, o que pode ocorrer após o término da transação, no caso de uma transação normal (não crítica).

Os campos da T_T são:

IDT Identificador da transação

IDT	Estado	Cliente	Crítica	Segura	Último_LSN	Lista_Serv_Arm

Figura 5.11: Tabela de Transações Pendentes

Estado	Estado da transação
Cliente	Endereço do cliente da transação
Crítica	Indica se a transação é crítica
Segura	Indica se os dados manipulados pela transação se encontram seguramente replicados
Último_LSN	LSN da última atualização feita pela transação
Serv_Modificado	Lista de servidores de armazenamento que tiveram dados modificados pela transação
Serv_Inseguro	Lista de servidores de armazenamento sem replicacao

5.2.9.1 Estados de uma Transação

O estado de uma transação determina o andamento da transação em relação ao protocolo de terminação. Desta forma, os estados que uma transação pode assumir são:

INICIAL	É o estado inicial de uma transação
PREPARADO	Uma transação entra no estado PREPARADO durante a fase de preparação do protocolo de terminação
ABORTADO	Indica que a transação abortou
NÃO-REPLICADO	Uma transação está no estado NÃO-REPLICADO após o seu término normal enquanto os dados primários não foram replicados nos servidores secundários

REPLICADO É o estado final de uma transação que não abortou

Uma transação crítica nunca entra no estado NÃO-REPLICADO, uma vez que os dados secundários são atualizados de forma síncrona à manipulação dos dados primários correspondentes. Neste caso, não há, portanto, dados não replicados após o término da transação.

Como o protocolo de terminação da arquitetura transacional distribuída é baseado no protocolo *presumed abort* (seção 2.6.3.1), não há necessidade de manter informações sobre uma transação que foi abortada. Este protocolo, na ausência de informações sobre uma transação t, simplesmente assume que t foi abortada, o que permite que transações abortadas simplesmente sejam removidas da tabela de transações pendentes.

5.2.9.2 Segurança de uma Transação

Na ocorrência de uma falha em um servidor de armazenamento, os servidores de transação assumem que o servidor está completamente indisponível. Desta forma, todas as transações normais que possuem atualizações neste servidor que ainda não foram replicadas precisam ser desfeitas. Por consequência, todas as outras transações que leram dados das transações que possuíam atualizações não replicadas também precisam ser desfeitas. A relação de transações normais que precisam ser refeitas nos casos de falhas em um servidor de armazenamento é mantida com o auxílio da T_T.

Uma transação normal t_I é considerada *insegura* na T_T, quando os servidores de armazenamento primários já estão compromissados com o desfecho da transação e as alterações destes servidores ainda não foram propagadas para os servidores secundários. De outra forma, uma transação t_L que tenha lido um dado atualizado por y_I também é dita insegura na T_T. Ou seja, enquanto os dados atualizados por t_I ainda não estão replicados, todas as transações t_L que lêem dados desta transação também são marcadas como inseguras na T_T.

Para gerenciar as transações inseguras, cada entrada na T_T contém o campo Segura que indica se a transação está segura e a lista de servidores de armazenamento sem replicação (campo Serv_Inseguro) que, caso apresentem falha, acarretem o aborto da transação. A cada atualização em um servidor de armazenamento S_A feita por uma transação t_I, S_A é adicionado a lista de servidores inseguros de t_I. De outro modo, se uma transação t_L ler um dado de t_I, t_L é igualmente marcada insegura na T_T. Neste caso, todos elementos da lista de servidores inseguros do registro de t_I são acrescentados a lista de mesmo nome de t_L na T_T.

5.2.10 Tabela de Transações Servidas

Analogamente a tabela de transações pendentes dos servidores de transação, os servidores de armazenamento contam com uma tabela com informações sobre as transações que estão sendo atendidas pelo servidor. A tabela de transações servidas - T_S, na verdade, é uma versão reduzida da tabela de transações pendentes, concentrando as informações de transações no contexto de um único servidor de armazenamento. A única diferença desta tabela em relação a tabela de transações pendentes é a presença do campo LSN_Desfazer, usado para desfazer atualizações de uma transação, que aponta para o próximo registro do *log* a ser desfeito.

IDT	Estado	Crítica	Grupo	Último_LSN	LSN_Desfazer

Figura 5.12: Tabela de Transações Servidas

Os campos da T_S são:

IDT	Identificador da transação
Estado	Estado da transação
Crítica	Indica se a transação é crítica

Grupo Armazena o grupo de replicação da transação. Possui valor zero nas transações críticas

Último_LSN LSN da última atualização feita pela transação

LSN_Desfazer LSN da última operação a ser desfeita

5.2.11 Tabela de Servidores

Os servidores de transação e o servidor de diretório contam com tabelas de servidores, que relacionam os servidores da arquitetura transacional distribuída. A tabela T_{SA} contém informações sobre os servidores de armazenamento e a tabela T_{ST} sobre os servidores de transação. Ambas tabelas compartilham da mesma estrutura (Figura 5.13).

Endereço	Índice	Estado	Carga

Figura 5.13: Tabela de Servidores

Onde os campos de cada item da tabela são:

Endereço Endereço do servidor na estrutura de comunicação

Índice É o índice do servidor

Estado O estado da disponibilidade do servidor

Carga Medida da carga do servidor

O Índice de um servidor na verdade é o identificador do servidor em relação ao tipo de serviço que este presta. Por exemplo, o Índice do servidor de armazenamento A_3 é 3. A presença dos campos Índice e Endereço nesta tabela permite que os servidores sejam acessados independentemente de seu endereço físico na estrutura de comunicações.

Este acesso indireto aos servidores é a base do mecanismo de servidores de reserva, que assumem o papel de um servidor indisponível com uma simples atualização na tabela de servidores.

O campo Carga é uma medida do volume de trabalho de um servidor. Este campo é usado como métrica de balanceamento de carga na alocação de servidores. No caso dos servidores de transação, o campo Carga indica o número de transações que o servidor está processando. Quando um cliente deseja iniciar uma nova transação, o servidor de diretório irá alocar o servidor de transações com o menor valor de Carga, o que significa o servidor que está processando o menor número de transações.

Como na arquitetura transacional distribuída os dados são uniformemente distribuídos entre os servidores de armazenamento, o campo Carga não faz sentido para estes servidores. Isto não impede, contudo, que este campo venha a ser empregado em outras estratégias de alocação de dados para estes servidores.

O campo Estado indica a disponibilidade de um servidor, que pode ser:

DISPONÍVEL	O servidor encontra-se disponível e operacional
INDISPONÍVEL	O servidor encontra-se totalmente indisponível
RECUPERAÇÃO	Se o servidor encontra-se em processo de recuperação de uma falha
BLOQUEADO	Se o servidor bloqueou a execução de transações normais
RESERVA	O servidor encontra-se disponível mas está em *stand-by*

5.2.12 Registro de Checkpoint

A operação de *checkpoint* periodicamente salva informações sobre a tabela de transações servidas T_S e a tabela de elementos voláteis T_{EV} em um *registro de checkpoint* R_{CK} (figura 5.14). Além de T_S e T_{EV}, R_{CK} conta com dois contadores n_{CK} e $n_{CK'}$ que indicam o número seqüencial deste registro.

$$\boxed{n_{CK}}\boxed{T_S}\boxed{T_{EV}}\boxed{n_{CK'}}$$

Figura 5.14: Registro de Checkpoint

O motivo da presença de dois contadores é detectar a presença de falhas no decorrer da atualização do registro de *checkpoint*, atualizando-se o primeiro contador antes de efetuar qualquer atualização neste registro e o outro após todas as alterações. Cada servidor de armazenamento conta ainda com dois registros de *checkpoints* que são atualizados alternadamente a cada operação de *checkpoint*. Esta duplicidade tem o objetivo de garantir que sempre haja um registro de *checkpoint* válido, mesmo na ocorrência de falhas durante a atualização do registro de *checkpoint*.

5.3 Processamento de Transações

Nesta seção descrevemos as principais operações da arquitetura transacional distribuída relativas ao processamento normal de transações. Adicionalmente, descrevemos a operação de *rollback* de uma transação, executada por um servidor sadio, quando detecta a falha em outro nó do ambiente, e a propagação de atualizações em dados replicados. As ações tomadas na recuperação de falhas, entretanto, são apresentadas na seção 5.4. Finalmente, os pseudo-códigos das operações descritas neste cápitulo encontram-se no apêndice A.

5.3.1 Início de uma Transação

Quando um nó cliente C deseja iniciar uma transação, C solicita ao servidor de diretório que aloque um servidor de transação para gerenciar esta nova transação. O servidor de diretório, por sua vez, designa o servidor de transação S_T que, entre os diversos servidores de transação disponíveis, naquele instante estiver gerenciando o menor número de transações. S_T é então notificado que C deseja iniciar a transação, alocando o identificador t para a nova transação na tabela T_T. Por fim, S_T informa t para C.

5.3.2 Leitura de Elemento de Dados

A operação de leitura de um elemento de armazenamento e por uma transação t em um cliente C, tal qual as demais operações da arquitetura transacional distribuída, é realizada com o intermédio do servidor de transações S_T responsável por t.

Inicialmente, S_T determina o servidor de armazenamento S_A que atenderá o pedido de leitura, verificando primeiro o servidor primário S_{AP} de e. Caso S_{AP} não esteja operacional, S_T direciona o pedido de leitura para S_{AS}, o servidor secundário de e. Uma vez contactado, S_A faz um *lock share* em e, enviando uma cópia de e diretamente para C. S_A também informa a S_T se e se encontra seguramente replicado. Caso isto não aconteça e t não seja uma transação crítica, t é marcada INSEGURA e S_A é inserido na lista de servidores inseguros Serv_Inseguro do registro de t na T_T.

A presença de S_A em Serv_Inseguro indica que se ocorrer alguma falha em S_A antes de t estar totalmente segura, t será abortada. Como a segurança de uma transação depende do mecanismo de propagação de réplicas, cuja atuação ocorre após o término da transação, t pode ser abortada mesmo após ao seu término.

Adicionalmente, a operação de leitura verifica se e foi atualizado por uma transação normal e está sendo lido por uma transação crítica. Esta preocupação tem o objetivo de evitar que uma transação crítica acesse um dado inseguro, o que comprometeria a sua segurança. Caso esta violação ocorra, a operação retorna uma situação de erro.

5.3.3 Leitura Simples em Pseudo-Transações

A leitura de um elemento de armazenamento e por uma pseudo-transação é uma operação simples, na qual o servidor de transações simplesmente se encarrega do mapeamento do servidor de armazenamento secundário S_A que guarda o elemento a ser lido. O pedido do cliente C_L é então encaminhado para S_A que, por sua vez, retorna o conteúdo de e para C_L. Como e pode estar sendo atualizado concorrentemente pelo mecanismo de propagação de réplicas, esta operação faz um *latch* em e durante a leitura, evitando a ocorrência de alguma inconsistência física em e.

5.3.4 Atualização de Elementos de Dados

A atualização de um elemento de armazenamento e por uma transação t possui características bem distintas, dependendo se t é uma transação crítica ou não. Se t for uma transação crítica, as cópias primária do servidor S_{AP} e secundária em S_{AS} de e são atualizadas simultaneamente. Caso contrário, apenas a cópia primária de e é atualizada, e é inserido na T_{NR} (ou a entrada de e em T_{NR} é alterada caso e tenha sido atualizado recentemente) e t é marcada como INSEGURA, indicando que t ainda depende do próximo grupo de replicação para que suas alterações estejam seguramente replicadas.

Cada servidor de armazenamento envolvido na atualização – S_{AP} se t for crítica, ou S_{AP} e S_{AS} caso contrário – grava em seu *log* um registro de ATUALIZAÇÃO. Adicionalmente, se esta é a primeira atualização que t faz no servidor, este também registra o INÍCIO de t em seu *log*, indicando que o servidor fará parte da votação para decidir o término de t. Esta distinção permite excluir os servidores de armazenamento que não realizaram atualizações do processamento da terminação de t, o que diminui o número de mensagens trocadas e a carga nestes servidores de armazenamento.

5.3.5 Rollbacks Totais e Parciais

Uma característica importante do ARIES que foi incorporada ao mecanismo de recuperação da arquitetura transacional distribuída consiste no suporte a *rollbacks* parciais. Complementar ao *rollback total*, que desfaz todas atualizações feitas por uma transação, o *rollback parcial* permite que se desfaça apenas as últimas alterações (ou simplesmente a última) feitas por uma transação. Ao contrário do que ocorre em um *rollback* total, após a execução de um *rollback* parcial, a transação pode prosseguir normalmente com a sua execução. O uso de *rollbacks* parciais é particularmente interessante para o suporte a transações aninhadas [47, 39] e violações de restrições de integridade.

Durante a execução de um *rollback*, total ou parcial, todas as atualizações são registradas em *log* através de registros de COMPENSAÇÃO. Estes registros garantem que as ações desfeitas de uma transação não terão que ser novamente desfeitas no caso de uma eventual falha no servidor de armazenamento. Isto ocorre pois os registros de COMPENSAÇÃO também armazenam as alterações realizadas durante o *rollback*, para que possam ser aproveitados em uma eventual recuperação de falhas do servidor de armazenamento.

5.3.6 Terminação de Transação

A operação de terminação de uma transação t é baseada no protocolo de terminação *presumed abort* (seção 2.6.3.1), uma variação do difundido *two-phase commit*. A figura do coordenador é exercida pelo servidor de transação S_T responsável por t e os participantes são todos os servidores de armazenamento que realizaram atualizações em nome de t. Como qualquer descendente do *two-phase commit*, a operação de terminação de t é feita em duas fases distintas.

Na primeira fase da operação de terminação, S_T envia mensagens a todos os servidores que realizaram atualização em t, devidamente registrados na lista Serv_Arm.

Recebendo esta mensagem, cada servidor de armazenamento S_A, caso não deseje abortar a transação, registra sincronamente em seu *log* um registro de PREPARADO e libera todos os *locks share*. A partir deste ponto, S_A abdicou do seu direito de abortar unilateralmente t, estando compromissado com esta decisão.

Quando todos os servidores de armazenamento tiverem manifestado o seu voto, se todos votaram pelo término normal de t, a segunda fase da terminação tem início. S_T então registra sincronamente em seu *log* um registro de COMMIT e informa a todos participantes que t terminou normalmente. Finalmente, registra em seu *log* um registro de TÉRMINO. Cada participante S_A, quando recebe esta mensagem de S_T, libera todos os *locks exclusive* e também escreve em seu *log* um registro de TÉRMINO. Note que este tipo de registro, seja em S_A ou em S_T, não precisa ser escrito sincronamente, uma vez que uma eventual perda deste registro apenas acarretaria em um esforço adicional durante a recuperação do servidor, sem prejuizos para a durabilidade da transação [49].

De outra forma, caso pelo menos um participante vote pelo aborto da transação ou não responda a solicitação de S_T, t é abortada. Cada participante S_A escreve no *log* um registro de ABORTAR, desfaz todas as atualizações feitas por t e, finalmente, registra no *log* um registro de TÉRMINO indicando que todos os traços de t foram eliminados. O procedimento seguido por S_T é semelhante, com exceção de no lugar de desfazer as atualizações, envia mensagens aos participantes comunicando que t foi abortada.

5.3.7 Checkpoints

Os servidores de armazenamento realizam *checkpoints* periódicos que visam reduzir o tempo de recuperação na ocorrência eventual de uma falha. Esta redução se dá principalmente através diminuição da extensão do *log* que necessita ser examinada e do número de atualizações, inclusive as realizadas através de COMPENSAÇÃO, que precisam ser refeitas durante o processo de recuperação de falhas.

Durante um *checkpoint*, os servidores de armazenamento registram em memória não volátil o conteúdo da tabela de transações servidas, T_S e tabela de elementos voláteis, T_{EV}. Esta informação, chamada de registro de checkpoint, é usada como ponto de partida na operação de recuperação de falhas destes servidores.

Para que a execução de um *checkpoint* não interrompa o processamento normal de transações, cada servidor de armazenamento mantém uma cópia auxiliar das tabelas T_S e T_{EV} em memória principal. No início da operação de *checkpoint*, as tabelas auxiliares são copiadas das tabelas originais, o que as libera rapidamente para o processamento normal de transações. Em seguida, as tabelas auxiliares podem ser armazenadas normalmente em memória não volátil.

Como pode ocorrer uma falha no servidor de armazenamento durante a execução de um *checkpoint*, cada servidor de armazenamento possui duas cópias de seu registro de *checkpoint*. A cada *checkpoint*, apenas um dos registros é atualizado alternadamente, cabendo a cópia que não tocada armazenar a versão anterior do registro de *checkpoint*. A proteção contra falhas é complementada com a presença em cada registro de um número sequencial n_{CK} que é incrementado a cada atualização e de campos adicionais para auxiliar na detecção de atualizações parciais [33]. Na recuperação de uma falha, o servidor de armazenamento busca a cópia do registro de *checkpoint* que possui o maior número seqüencial e que não tenha sido parcialmente atualizado.

Cabe ressaltar que em nenhum momento um *checkpoint* força a gravação de elementos de dados sujos em memória não volátil. A transferência de dados entre a memória principal e o armazenamento estável é tarefa exclusiva do mecanismo de gerência de *buffers* (seção 4.6.3). A premissa básica deste mecanismo é agir de forma independente às questões relativas à durabilidade de uma transação, incluindo a recuperação de falhas, voltando-se exclusivamente para a melhoria do desempenho do sistema.

5.3.8 Propagação de Réplicas

O mecanismo de propagação de réplicas é responsável pelo controle da replicação dos elementos de armazenamento atualizados em transações normais. A replicação neste caso ocorre sempre após o término da transação, sendo aplicada em lotes periódicos, chamados de *grupos de replicação*. Cada grupo de replicação reúne diversas transações em uma operação de replicação conjunta, cujo objetivo principal é reduzir o peso da replicação no processamento de uma transação normal. Isto é feito através da divisão dos custos do sincronismo entre as transações de um mesmo lote, e no uso mais eficiente de operações de E/S, melhor otimizadas no contexto mais amplo de um lote de transações.

Cada grupo de replicação é identificado por um número seqüencial g que é incrementado em intervalos de tempo regulares. Este número é de conhecimento de todos os servidores, sendo difundido através de *broadcast* através da rede de comunicação.

Durante o processamento normal de transações, cada servidor de armazenamento S_A guarda informações sobre os elementos atualizados por transações normais no decorrer do grupo corrente de replicação g_i. Quando uma transação normal t termina, S_A atualiza a T_{NR}, tabela de elementos não replicados, com os registros dos elementos alterados por t. No momento que S_A recebe uma mensagem indicando que g_i foi incrementado, marcando o início do grupo de replicação g_{i+1}, S_A inicia a transferência dos elementos atualizados no decorrer de g_i e também começa a registrar as novas alterações não replicadas do grupo g_{i+1}.

A tabela T_{NR}, para cada grupo de replicação g_i, possui uma única entrada para cada elemento de armazenamento e. Quando as alterações do grupo g_i forem propagadas para os servidores secundários, apenas uma cópia de e será replicada, independentemente do número de atualizações que e sofreu no decorrer de g_i. Isto reduz o número de réplicas que necessitam ser alteradas e também o tráfego de mensagens na rede. Este mecanismo é particularmente interessante para o tratamento de elementos de armazenamento com alto grau de atualização, ou *hot spots*.

Por sua vez, quando um servidor de armazenamento secundário S_{AS} recebe uma cópia de elemento e para ser replicada, S_{AS} registra em um *log* a parte, o *log* secundário, o recebimento da cópia de e e faz a atualização desta cópia. As informações do *log* secundário, semelhantemente ao *log* principal, são usadas pelo mecanismo de recuperação de falhas em uma eventual interrupção de um servidor de armazenamento.

Semelhante a uma transação, as atualizações de um grupo de propagação devem ser atômicas em todos os servidores de armazenamento. Isto é, ou estão refletidas em todos os servidores de armazenamento, ou em nenhum deles. Isto é feito através de um protocolo de terminação semelhante ao *two-phase commit* chamado de *distributed group commit* [19]. Neste protocolo, a atomicidade das replicações é garantida em termos de um grupo de replicações e não a cada término de transação. Isto permite amortizar o custo das escritas forçadas em *log* e da troca de mensagens do protocolo de terminação entre todas as transações de um grupo de propagação, economizando assim os recursos do ambiente.

5.4 Recuperação de falhas em Servidores de Armazenamento

Após a ocorrência de uma falha em um servidor de armazenamento S_A, este servidor precisa executar um mecanismo de recuperação que visa garantir a atomicidade e durabilidade das transações que realizaram alterações em S_A. A recuperação dos dados do servidor é feita em duas etapas distintas.

Na primeira etapa, chamada de *recuperação independente*, S_A tenta restaurar o seu estado interno a partir das informações contidas em seu *log* e no registro de *checkpoint*, sem nenhuma troca de informação com os demais servidores da arquitetura. A recuperação independente é semelhante a operação de *restart processing* do ARIES, com a diferença de também prover atuar na recuperação dos dados secundários de

S_A.

Na etapa posterior da recuperação, S_A recorre aos outros servidores de armazenamento para buscar as atualizações que foram realizadas enquanto S_A se recuperava da falha. Como S_A assume o papel exclusivo de servidor secundário até estar totalmente recuperado da falha, nesta fase da recuperação não há distinção de tratamento entre os dados primários e secundários, uma vez que todos os dados de S_A são vistos como secundários pelos outros servidores.

5.4.1 Recuperação Independente

A recuperação independente de um servidor de armazenamento S_A é baseada no algoritmo de *restart processing* do ARIES, com a diferença de possuir uma atuação específica para os dados secundários armazenados em S_A. Este algoritmo possui três fases distintas, *análise*, *refazer atualizações* e *desfazer atualizações*. Nas próximas seções descrevemos as fases na recuperação independente dos dados primários de S_A. A recuperação independente dos dados secundários é descrita um pouco adiante, na seção 5.4.1.4.

5.4.1.1 Fase de Análise

O objetivo da fase de análise da recuperação independente é reconstruir as tabelas de transações servidas, T_S e a tabela de elementos voláteis, T_{EV}. A análise é feita percorrendo-se o *log* de S_A buscando resgatar o histórico das transações que realizaram atualizações em S_A.

Nesta fase, se for encontrado no *log* um registro de ATUALIZAÇÃO ou COMPENSAÇÃO r relativo a um elemento de armazenamento e que não esteja presente na T_{EV}, uma entrada para e é criada nesta tabela, com o campo LSN_Alt contendo o LSN de r. Adicionalmente, T_S é modificada para registrar as mudanças de estado das transações e o LSN do registro mais recente que precisa ser desfeito caso uma transação tenha que abortada.

5.4.1.2 Fase de Refazer Atualizações

A segunda fase da recuperação é responsável por refazer as atualizações que potencialmente poderiam não terem sidos atualizadas pelo gerente de *buffers* até o momento da falha. Como a política de gerência de *buffers* é totalmente independente do processamento de transações, o mecanismo de recuperação necessita refazer todas as atualizações que não tem certeza se foram efetivamente gravadas em memória não volátil. A necessidade da fase de refazer transações é o preço a ser pago na recuperação para otimizar o desempenho do gerente de *buffers* na ausência de falhas.

Durante a fase de refazer atualizações, se um registro de ATUALIZAÇÃO ou COMPENSAÇÃO para um elemento e for encontrado, é verificado se e pertence a T_{EV}. Se e pertencer e o LSN do registro for superior ao campo LSN_Alt de entrada de e na T_{EV}, então e é acessada e, se o seu LSN for inferior ao LSN do registro, e é atualizada. Este mecanismo restringe os acessos a disco apenas aos elementos presentes na T_{EV}, eliminando os custosos acessos desnecessários a disco desta fase.

5.4.1.3 Fase de Desfazer Atualizações

A operação responsável por desfazer as atualizações de transações não concluídas é a última fase da recuperação independente de S_A. Este procedimento percorre a T_S procurando por transações que não estejam no estado de REPLICADO, desfazendo as suas atualizações a partir do campo Último_LSN que aponta para a última atualização feita pela transação.

5.4.1.4 Recuperação Independente de Dados Secundários

A recuperação dos dados secundários de S_A é realizada de forma semelhante a recuperação dos dados primários. Tal qual o outro mecanismo, a recuperação independente dos dados secundários também é feita em três fases. A grande diferença entre os mecanismos de recuperação independente dos dados primários e secundários reside no fato do primeiro mecanismo tratar da recuperação de transações enquanto que o último opera sobre de grupos de propagação. Como existe uma relação de ordem estrita entre os grupos de propagação, o que não ocorre com transações que são por definição totalmente independentes umas das outras, o mecanismo de recuperação de dados secundários é mais simples do que o de dados primários.

A fase de análise deste mecanismo busca no *log* secundário o histórico dos últimos grupos de replicação, procurando saber se um determinado grupo g_i foi corretamente aplicado, abortou ou foi interrompido por uma falha. As fases de refazer e desfazer atualizações são análogas as fases de mesmo nome de recuperação independente de dados primários.

5.4.2 Atualização Remota de Dados

A fase final da recuperação de um servidor de armazenamento S_A consiste na propagação das atualizações feitas nos outros servidores durante o tempo em que S_A esteve indisponível. Isto é feito com intermédio do servidor de reserva S_R que assumiu o lugar de S_A.

Assim que a falha de S_A é detectada, ocorre uma inversão de papel para as cópias secundárias dos dados primários de S_A que passam a serem vistas como cópias primárias. Adicionalmente, o servidor de reserva S_R entra em ação atuando apenas como servidor secundário dos dados que seriam primários em S_A. Desta forma, S_R exerce a função de anotar as atualizações que seriam direcionadas a S_A, repassando-as para o servidor titular assim que este se encontrar disponível.

A transferência de atualizações de S_R para S_A é otimizada com o uso a tabela de elementos não replicados em S_R. A T_{NR} aponta para a última versão de cada elemento armazenado temporariamente em S_R, isto permite que haja apenas uma operação de transferência por elemento de armazenamento, independentemente do número de atualizações que o elemento sofreu.

Em um determinado instante, é necessário que a disposição normal das cópias dos dados de S_A seja reestabelecida, fazendo com que este servidor volte a ser o detentor das cópias primárias de seus dados. Isto é feito bloqueando o acesso das transações normais aos dados de S_A durante um ciclo de propagação completo. Como neste intervalo de tempo, não existem novas atualizações nos dados de S_A, ao final do grupo de propagação as cópias primárias e secundárias dos dados de S_A estarão perfeitamente sincronizadas. Este sincronismo permite que seja feita uma nova inversão na alocação de cópias primárias e secundárias, reestabelecendo-se a alocação normal dos dados entre todos os servidores de armazenamento.

Note que apesar da operação de inversão tornar os dados de S_A indisponíveis para as transações normais durante o período de um grupo de propagação, não há nenhum impacto sobre as transações críticas que continuam tendo acesso normal aos dados do servidor. Como as transações críticas sempre mantém as cópias primárias e secundárias de suas atualizações perfeitamente sincronizadas, não há necessidade de interromper o seu processamento para uma nova inversão das cópias primárias e secundárias.

5.4.3 Falhas Irrecuperáveis

Na ocorrência de uma falha irrecuperável em S_A que impeça totalmente o seu reestabelecimento, S_R assume definitivamente o lugar de S_A. Isto é feito atualizando-se S_{AR} para que este contenha a versão completa dos dados de S_A. Cada servidor de armazenamento que contenha uma fração dos dados de S_A e esteja agindo temporariamente como servidor primário destes dados, vai enviando a última versão de cada elemento desta fração até S_R ficar atualizado.

Quando S_R estiver totalmente atualizado, é feito o procedimento normal de reinversão das cópias primárias e secundárias dos dados que pertenciam a S_A. Tal qual a operação de recuperação de S_A, a segunda inversão das cópias dos dados requer o bloqueio do acesso a estes dados por transações normais durante um ciclo de propagação de réplicas.

5.5 Recuperação de Falhas em Servidores de Transação

A recuperação de falhas em um servidor de transação S_T é uma operação bem mais simples do que a de um servidor de armazenamento. Como S_T não armazena nenhum dado, os mecanismos de refazer e desfazer atualizações não são necessários na recuperação do servidor. A única missão a ser cumprida, portanto, é a recomposição da tabela de transações T_T de S_T através de um mecanismo semelhante a fase de análise (seção 5.4.1.1) da recuperação independente de um servidor de armazenamento. Este mecanismo simplesmente percorre o *log* de S_T reproduzindo as mudanças de estados das transações do servidor que são incrementalmente refletidas na tabela T_T.

Capítulo 6

Considerações Finais

6.1 Resumo

Neste trabalho propôs-se uma arquitetura para o processamento distribuído de transações com mecanismos adicionais que garantem maior confiabilidade no acesso à informações armazenadas. Esta arquitetura é uma variação do modelo *shared nothing* que introduz a divisão das funções de um servidor em *servidores de armazenamento* e *servidores de transação*, responsáveis respectivamente pelo armazenamento de dados e pelo controle da transação. Tal divisão permite isolar o processamento de transações das questões relativas ao armazenamento de dados, dando mais flexibilidade à arquitetura. A arquitetura conta ainda com a presença de uma estrutura central de comunicação de dados que evita a ocorrência de falhas de difícil contorno pela arquitetura distribuída, o que torna os mecanismos de segurança da arquitetura mais simples e eficientes.

O mecanismo de recuperação da arquitetura transacional distribuída é uma extensão do ARIES, um difundido método de recuperação baseado no protocolo *write-ahead logging*, que adiciona o suporte à replicação de dados. O modelo de replicação deste mecanismo, baseado na abordagem por *cópia primária*, prevê suporte diferenciado às aplicações que exijam maior segurança na manipulação de dados. Para estas aplicações, o mecanismo de recuperação garante que todas as cópias de um determinado dado sempre estão em perfeito sincronismo. De outra forma, para as aplicações que não possuam requisitos especiais de segurança, a replicação das atualizações é realizada em lotes que visam economizar os recursos do ambiente. Adicionalmente, o mecanismo de replicação oferece suporte a *pseudo-transações*, uma transação com requisitos mínimos de confiabilidade, que suporta operações de leitura sobre os dados replicados. As pseudo-transações permitem um melhor aproveitamento dos dados replicados, usualmente pouco aproveitados no contexto da abordagem por cópia primária, e oferecem uma opção econômica em termos de recursos para aplicações que toleram a presença de dados com ligeiras inconsistências.

6.2 Contribuições

O presente trabalho apresenta uma série de contribuições no que diz respeito a ambientes distribuídos para o processamento de transações, descritas a seguir.

6.2.1 Servidores de Armazenamento e de Transação

Um servidor na arquitetura *shared nothing* acumula as responsabilidades do gerenciamento do armazenamento e das transações em andamento. Já na arquitetura transacional distribuída, estes dois papéis são designados separadamente aos servidores de armazenamento e aos servidores de transação. Esta separação permite maior flexibilidade na configuração da arquitetura que, desta forma, pode dispor de quantidades distintas de cada tipo de servidor, adequando-se com mais precisão às necessidades de suas aplicações.

Os servidores de transação atuam como autênticos representantes dos clientes, concentrando a responsabilidade do gerenciamento de todas as transações do ambiente. Este servidor também possui a função de abstrair o método de particionamento e replicação dos dados dos servidores de armazenamento. Tal abstração, além de facilitar alterações e reestruturações nos dados dos servidores de armazenamento, é de suma importância em situações de falha em um servidor de armazenamento, pois permite que as requisições dos clientes sejam transparentemente redirecionadas a outros servidores enquanto a falha perdurar. Ou seja, permite que os clientes tolerem a ocorrência de falhas em um servidor de armazenamento, com conseqüências mínimas para as aplicações críticas e indisponibilidade momentânea para as aplicações normais.

6.2.2 Estrutura Central de Comunicação

A presença de uma estrutura central de comunicação, apesar de ser lugar comum em diversas tecnologias de rede local, evita a existência de partições na comunicação entre processadores. Como a estrutura central é o único elo de ligação de um processador, a impossibilidade de comunicação deste processador com a estrutura tem o mesmo significado prático do que uma falha no processador. Isto permite o uso dos mecanismos comuns de recuperação de falha, dispensando assim a exigência de protocolos mais complexos de tratamento de partições em estruturas de comunicação.

6.2.3 Replicação de Dados

A estratégia de replicação de dados em um ambiente distribuído é de suma importância no desempenho do ambiente como um todo. Neste trabalho optou-se por adotar a simplicidade da abordagem por cópia primária, que mostrou-se interessante a partir de observações em comparações de desempenho de arquiteturas de banco de dados paralelos, em detrimento de estratégias que não fazem distinções entre as cópias de um mesmo dado. A escolha da abordagem por cópia primária foi reforçada pelos diferentes métodos de propagação de replicação que esta abordagem oferece, o que permite a adoção de níveis de serviço diferenciados de segurança para uma transação. O baixo aproveitamento das cópias secundárias, um ponto negativo da abordagem por cópia primária, também é reduzido na arquitetura transacional distribuída, permitindo-se o acesso de pseudo-transações a estes dados em particular.

6.3 Possíveis Extensões e Trabalhos Futuros

No decorrer do trabalho, vários aspectos não tiveram um tratamento exaustivo, permanecendo em aberto para futuras pesquisas. Alguns deles já foram apontados, e serão ressaltados novamente nesta seção, em conjunto com outros pontos observados que são relevantes para possíveis extensões deste trabalho.

O primeiro ponto a ser abordado em uma continuação deste trabalho seria a implementação das idéias e pseudo-códigos aqui apresentados. Uma implementação certamente traria a tona diversos pormenores e daria uma visão mais embasada da arquitetura transacional distribuída.

Outro trabalho futuro relevante seria a elaboração de um modelo do desempenho da arquitetura transacional distribuída. Tal estudo possibilitaria uma avaliação precisa do desempenho da arquitetura e, em particular, do seu mecanismo de replicação. Este estudo permitiria a definição de diversos parâmetros da arquitetura, como por exemplo, a relação entre o número de servidores de transação e de armazenamento, duração de um ciclo de propagação, entre outros.

Como extensão à proposta da arquitetura transacional distribuída, é importante o estudo do impacto do uso das relações de afinidade (seção 3.4) no desempenho da arquitetura transacional distribuída, uma vez que tais relações comprovadamente possuem grande influência no desempenho de ambientes distribuídos.

Outros estudos poderiam explorar:

- a presença de múltiplas cópias secundárias, o que permitiria a tolerância a falhas de mais de um servidor de armazenamento, mas, em contra-partida, traria impactos no desempenho do sistema.
- o uso de servidores de transação dedicados a aplicações *batch* intensivas, o que certamente implicaria em técnicas mais apuradas de balanceamento de carga

nestes servidores.

- integração da arquitetura transacional distribuída ao DCE [40], ou outros padrões para ambientes distribuídos, possibilitando o uso dos serviços providos pelos ambientes na arquitetura e, consequentemente, a coexistência com outras aplicações também baseadas no mesmo ambiente distribuído.

- o suporte a transações aninhadas [39, 46], que são particularmente interessantes em um ambiente distribuído.

Referências Bibliográficas

[1] El Abbadi and D. Skeen adn F. Cristian. An Efficient, Fault-Tolerant protocol for Replicated Data Management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Portland, Oregon, janeiro 1985.

[2] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, CA, 1976.

[3] Juan M. Andrade, Mark T. Carges, and M. Randall MacBlane. The Tuxedo System: An Open On-line Transaction Processing Environment. *IEEE Data Engineering*, 17(1), março 1994.

[4] P. Bernstein, W. Emberton, and V. Trehan. DECdta - Digital's Distributed Transaction Processing Architecture. *Digital Technical Journal*, 3(1), 1991.

[5] Philip A. Bernstein and Nathan Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems*, 9(4), dezembro 1984.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] E. Braginsky. The X/Open DTP Effort. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, Asilomar, setembro 1991.

[8] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. In *Distributed Systems*, ACM Press Frontier Series, chapter 3. Addison-Wesley, New York, 1993.

[9] Michael J. Carey and Miron Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems*, 16(4), dezembro 1991.

[10] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3), setembro 1985.

[11] David DeWitt and Jim Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD RECORD*, 19(4):104–112, dezembro 1990.

[12] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of ACM*, 35(6):85–98, junho 1992.

[13] David J. DeWitt. The Gamma Database Machine Project. *IEEE Knowledge and Data Engineering*, 2(1), marcço 1990.

[14] Daniel M. Dias, Balakrishna R. Iyer, John T. Robinson, and Philip S. Yu. Integrated Concurrency-Coherency Controls for Multisystem Data Sharing. *IEEE Transactions on Software Engineering*, 15(4):437–448, abril 1989.

[15] D. L. Eager and K. V. Sevick. Achieving Robustness in Distributed Databases Systems. *ACM Transactions on Database Systems*, 8(3), setembro 1983.

[16] Wolfgang Effelsberg and Theo Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4), dezembro 1984.

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, november 1976.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association of Computing Machinery*, 22(2):374–382, abril 1985.

[19] Hector Garcia-Molina, Chrisos A. Polyzois, and Robert Hagmann. Two Epoch Algorithms for Disaster Recovery. In *Proceedings of the 16th Very Large Data Bases Conference*, Brisbane, Australia, 1990.

[20] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2), junho 1981.

[21] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. Technical Report RJ 1654, IBM Research Division, T. J. Watson Research Center, setembro 1975.

[22] James Gray. Notes on Data Base Operating Systems. In *Operating Systems - An Advance Cource*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[23] Theo Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2), 1984.

[24] Theo Haerder. Handling Hot Spot Data in DB-Sharing Systems. *Information Systems*, 13(2):155–166, 1988.

[25] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, dezembro 1983.

[26] P. Helland. The LU 6.2 protocol boundary: The 'L' stands for 'Lightweight'. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, Asilomar, setembro 1989.

[27] Maurice Herlihy. Concurrency and Availability as Dual Properties of Replicated Atomic Data. *Journal of the Association for Computing Machinery*, 37(2), abril 1990.

[28] Walter H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys*, 13(2), junho 1981.

[29] Akhil Kumar and Arie Segev. Cost and Availability Tradeoffs in Replicated Data Concurrency Control. *ACM Transactions on Database Systems*, 18(1), março 1993.

[30] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1), marcço 1977.

[31] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proceedings of the 16th Very Large Data Bases Conference*, Brisbane, agosto 1990.

[32] C. Mohan. Less Optmism About Optimistic Concurrency Control. In *Proceedings of the 2nd International Workshop on RIDE: Transaction and Query Processing*, Tempe, fevereiro 1992.

[33] C. Mohan. Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Log. Technical Report RJ 9741, IBM Research Division, Almaden Research Center, março 1994.

[34] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), março 1992.

[35] C. Mohan and B. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. In *Proceeding of the 2nd Annual ACM Symposium on Principles on Distributed Computing*, Montreal, agosto 1983.

[36] C. Mohan, B. Lindsay, and R. Obermack. Transaction Management in the R* Distributed Database Management System. *ACM Transaction on Database Systems*, 11(4):378–396, dezembro 1986.

[37] C. Mohan, I. Narang, and S. Silen. Solutions to Hot Spot Problems in a Shared Disks Transaction Environment. In *Proceedings of 4th International Workshop on High Performance Transaction Systems*, Asilomar, CA, setembro 1987.

[38] C. Mohan, Kent Treiber, and Ron Obermarck. Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery. In *Proceedings of the 9th International Conference on Data Engineering*, Viena, abril 1993.

[39] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press series in information systems. Massachusetts Institute of Technology, 1985.

[40] Open Software Foundation. *Introduction to OSF DCE*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[41] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, 1991.

[42] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the Association for Computing Machinery*, 26(4), outubro 1979.

[43] Christos A. Polyzois and Hector Garcia-Molina. Evaluation of Remote Backup Algorithms for Transaction Processing Systems. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 21(2):246–255, junho 1992.

[44] Christos A. Polyzois and Hector Garcia-Molina. Processing of Read-Only Queries at a Remote Backup. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, Dana Point, California, outubro 1994.

[45] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):125–165, junho 1978.

[46] Daniel R. Ries and Gordon Smith. Nested Transactions in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-8(3), maio 1982.

[47] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. Technical Report RJ 6650, IBM Research Division, Almaden Research Center, janeiro 1989.

[48] Kurt Rothermel and Stefan Pappe. Open Commit Protocols Tolerating Commission Failures. *ACM Transactions on Database Systems*, 18(2):289–332, junho 1993.

[49] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proceedings 9th International Conference on Data Engineering*, Vienna, abril 1993.

[50] G. Schlageter. Optimistic Methods for Concurrency Control in Distributed Database Systems. In *Proceedings of the 7th Very Large Data Bases Conference*, Cannes, Francça, 1981.

[51] Mark Sherman. Architecture of the Encina Distributed Transaction Processing Family. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 22(2), maio 1993.

[52] D. Skeen. A Decentralized Termination Protocol. In *Proceedings of the 1st IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, julho 1981.

[53] Dale Skeen and Michael Sonebreaker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, maio 1983.

[54] Michael Stonebreaker. The Case of Shared Nothing. *IEEE Database Engineering*, 9(1), 1986.

[55] Tandem Performance Group. A Benchmark of NonStop SQL on the Debit Credit Transaction. In *Proceedings of the 1988 SIGMOD Conference*, Chicago, junho 1988.

[56] S. S. Thakkar and M. Sweiger. Performance of an OLTP Aplication on Symmetry Multiprocessor System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, maio 1990.

[57] F. Upton IV. OSI Distributed Transaction Processing, An Overview. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, Asilomar, setembro 1991.

[58] William E. Weihl. Specifications of Concurrent and Distributed Systems. In *Distributed Systems*, ACM Press Frontier Series, chapter 3. Addison-Wesley, New York, 1993.

[59] Philip S. Yu and Asit Dan. Performance Analysis of Affinity Clustering on Transaction Processing Coupling Architectures. Technical Report RC 18166, IBM Research Division, T. J. Watson Research Center, julho 1992.

Apêndice A

Pseudo-Código das Operações da Arquitetura Transacional Distribuída

No desenvolvimento do mecanismo de recuperação e replicação da arquitetura transacional distribuída, os pseudo-códigos das principais operações deste mecanismo foram elaborados com o objetivo de proporcionarem uma visão mais detalhada do funcionamento e pormenores desta arquitetura. Estes pseudo-códigos certamente contribuíram para a resolução de diversas questões que surgiram no desenvolvimento da arquitetura, atuando principalmente como um instrumento de verificação das idéias propostas nesta dissertação. Em linhas gerais, também podem ser vistos como uma primeira aproximação de uma futura implementação do mecanismo de recuperação e replicação da arquitetura aqui apresentada.

Neste apêndice descrevemos as convenções usadas no desenvolvimento dos pseudo-códigos, as operações exportadas para os clientes, as operações que são automaticamente executadas pela arquitetura, as consideradas primitivas – que assumimos já se encontrarem disponíveis para a arquitetura – e os pseudo-códigos das operações do mecanismo de recuperação e replicação propriamente ditos. Estes últimos encontram-se, por sua vez, expostos em duas seções distintas, a primeira relativa as principais operações dos servidores de transação, enquanto a última detalha os procedimentos mais relevantes dos servidores de armazenamento.

A.1 Convenções

Todos os procedimentos descritos em pseudo-código neste apêndice obedecem uma convenção que visa identificar o nó da arquitetura transacional distribuída onde tal procedimento é executado. Esta identificação é obtida através da adição de um prefixo a todos os nomes de procedimento, indicando que:

NO o procedimento é compartilhado por qualquer tipo de nó

SA o procedimento é usado pelos servidores de armazenamento

ST o procedimento é usado pelos servidores de transação

Por exemplo, o prefixo *ST* do procedimento *ST_Mapear_Serv_Arm*(t,i) determina que este procedimento é executado por servidores de transação. De outro modo, o procedimento *NO_Enviar_Mensagem(n, msg)*, por possuir prefixo *NO*, é usado por qualquer nó da arquitetura transacional distribuída.

A.2 Operações Exportadas

Esta seção lista as operações da arquitetura transacional distribuída que são disponibilizadas para os nós cliente. Estas operações são executadas nos servidores de transação e assumimos que são acessadas pelos clientes através de um mecanismo de execução remota de procedimentos, como por exemplo o *RPC - Remote Procedure Call* do DCE [40].

- *ST_Iniciar_Transação*(C_L, *Crític*)

Este procedimento, descrito na seção 5.3.1 e com pseudo-código na página 129, inicia uma nova transação t (retorno do procedimento) em nome do cliente C_L, onde o parâmetro $Crític$ indica se a nova transação é crítica.

- *ST_Leitura_Elemento(t, i)*

  Este procedimento, descrito na seção 5.3.2 e com pseudo-código na página 126, faz a leitura de um elemento de armazenamento de número i para a transação t.

- *ST_Leitura_Simples(C_L, i)*

  Este procedimento, descrito na seção 5.3.3 e com pseudo-código na página 127, faz a leitura simples de um elemento de armazenamento de número i, que é enviado diretamente para o cliente C_L que requisitou a leitura. Este procedimento é empregado em pseudo-transações.

- *ST_Atualização_Elementot, i, e)*

  Este procedimento, descrito na seção 5.3.4 e com pseudo-código na página 128, faz a atualização do no novo conteúdo e do elemento de armazenamento número i para a transação t.

- *ST_Abortar_Transação(t)*

  Este procedimento, descrito na seção 5.3.5 e com pseudo-código na página 131, faz o aborto da transação t.

- *ST_Terminar_Transação(t)*

  Este procedimento, descrito na seção 5.3.6 e com pseudo-código na página 130, realiza o término da transação t.

A.3 Operações Automáticas

A arquitetura transacional distribuída possui uma série de procedimentos que são automaticamente executados na ocorrência de falhas e para a propagação de réplicas das atualizações de transações normais.

Os procedimentos iniciados na recuperação de falhas, descrita na seção 5.4, são:

- *SA_Reiniciar*()

  Este procedimento, cujo pseudo-código encontra-se na página 139, é responsável pela recuperação de falhas em um servidor de armazenamento.

- *ST_Falha_Serv_Armazenamento*(S_A)

  Este procedimento, cujo pseudo-código encontra-se na página 133, reune as ações que são tomadas por um servidor de transação na ocorrência de falhas em um servidor de armazenamento S_A.

- *SA_Falha_Serv_Transação*(S_T)

  Este procedimento, cujo pseudo-código encontra-se na página 145, descreve as ações que são tomadas por um servidor de armazenamento na ocorrência de falhas em um servidor de transação S_T.

O procedimento automático relacionado com a propagação de réplicas é:

- *ST_Gerenciar_Replicação*()

  Este procedimento, descrito na seção 5.3.8 e com pseudo-código na página 132 é responsável pelo gerenciamento de toda a atualização de réplicas de transações normais.

A.4 Operações Primitivas

Diversas operações usadas no mecanismo de recuperação e replicação são operações simples ou não são suficientemente relevantes para que mereçam seu detalhamento em pseudo-código. Dizemos então que tais operações são *primitivas* no contexto do mecanismo de recuperação e replicação, sendo descritas informalmente a seguir.

- *ST_Gerar_TID()*

  Gera um novo identificador de transação t (parâmetro de retorno). Este identificador é criado a partir de um contador de transações local a cada servidor de transação que é incrementado a cada novo identificador gerado.

- *SA_Lock(t,i,modo)*

  Realiza um *lock* do tipo *modo* em nome de uma transação t sobre o elemento de armazenamento i.

- *SA_Unlock_Tipo(t,modo)*

  Desfaz todos os *locks* do tipo *modo* de uma transação t. Como este procedimento atua de acordo com o *two-phase locking* estrito (seção 2.4.1.1), só é empregado no término de transações.

- *SA_Unlock_ALL(t)*

  Desfaz todos os *locks* de uma transação t independente de seu tipo. Tal qual o *SA_Unlock_Tipo*, este procedimento só é usado no término de transações, por agir conforme o *two-phase locking* estrito.

- *SA_Latch(t,i,modo)*

  Faz um *latch* do tipo *modo* em nome de uma transação t no elemento de armazenamento i.

- *SA_Unlatch(t,i)*

  Desfaz um *latch* de um elemento de armazenamento i previamente realizado por uma transação t.

- *ST_Registrar(t, tipo, comp)*

  Adiciona em nome da transação t um registro no *log* do servidor de transação do tipo *tipo*. O parâmetro opcional *comp* é usado para especificar o complemento do registro, nos tipos de registros que exigam este campo adicional. Este registro é adicionado de forma assíncrona ao *log* do servidor (seção 2.5.2.2).

- *ST_Registrar_Firme(t, tipo, comp)*

  Semelhante ao procedimento *ST_Registrar* com a diferença de adicionar o registro de forma síncrona ao *log* do servidor de transação, gravando em disco também todos os registros anteriores do *log* ainda em memória volátil.

- *SA_Registrar(t, tipo, comp)*

  Procedimento dos servidores de armazenamento dual ao *ST_Registrar* dos servidores de transação.

- *SA_Registrar_Firme(t, tipo, comp)*

  Procedimento dos servidores de armazenamento dual ao *ST_Registrar_Firme* dos servidores de transação.

- *SA_Registrar_Secundário(t, tipo, comp)*

  Procedimento análogo à *SA_Registrar* com a exceção de manipular o *log* secundário do servidor de armazenamento.

- *SA_Registrar_Secundário_Firme(t, tipo, comp)*

  Procedimento análogo à *SA_Registrar_Firme* com a exceção de manipular o *log* secundário do servidor de armazenamento.

- *SA_Início_Log()*

  Retorna o *lsn* do primeiro registro do *log* principal do servidor de armazenamento.

- *SA_Próximo_Registro_Log(r)*

  Retorna o *lsn* registro posterior a r do *log* principal do servidor de aramzenamento.

- *SA_Buscar_Elemento(i)*

  Busca o elemento de armazenamento e (parâmetro de retorno) cujo número é i. Esta operação é implementada pelo gerente de *buffers* (seção 4.6.3) que efetivamente se encarrega de recuperar o elemento de armazenamento, seja da memória secundária ou de um *buffer* em memória principal.

- *SA_Escrever_Elemento(i, e)*

  Operação responsável por atualizar a versão estável do elemento de armazenamento de número i com o novo conteúdo e. Esta operação é implementada pelo gerente de *buffers* e não necessariamente atualiza o elemento de armazenamento no momento em que foi executada. Tal atualização ocorre no momento mais propício em termos de desempenho para o gerente de *buffers*.

- *SA_Check_Point()*

  Realiza um *checkpoint* em um servidor de armazenamento.

- *SA_Recuperar_TS()*

  Recupera a tabela de transações servidas, T_S, do registro de *checkpoint*.

- *SA_Recuperar_TEV()*

  Recupera a tabela de elementos voláteis, T_{EV}, do registro de *checkpoint*.

- *NO_Enviar_Mensagem(n, msg)*

  Envia uma mensagem msg através da estrutura de comunicação para o nó n.

- *NO_Aguardar_Mensagem(n)*

  Aguarda o recebimento de uma mensagem de um nó n.

A.5 Operações do Servidores de Transações

Nesta seção apresentamos o pseudo-código das principais operações realizadas pelos servidores de transações.

$ST\_Mapear\_Serv\_Simples(i)$

Parâmetros

i = Índice do Elemento de Armazenamento a ser acessado

Variáveis locais

S_{AP} = Servidor de Armazenamento Primário
S_{AS} = Servidor de Armazenamento Secundário
I_{AP} = Índice do Servidor de Armazenamento Primário
I_{AS} = Índice do Servidor de Armazenamento Secundário

Variáveis globais

T_{SA} = Tabela de Servidores de Armazenamento

Algoritmo

$I_{AP} \leftarrow ((i-1) \bmod N_{SA}) + 1$
se $T_{SA}[I_{AP}]$.Estado = DISPONÍVEL **então**
  $I_{AS} \leftarrow (I_{AP} + ((I_{AP}-1) \div N_{SA}) \bmod (N_{SA}-1)) \bmod N_{SA} + 1$
  $S_{AS} \leftarrow T_{SA}[I_{AS}]$.Endereço
  retornar S_{AS}
senão
  retornar erro
fim-se

Figura A.1: Algoritmo de Mapeamento do Servidor para Leitura Simples

$ST\_Mapear\_Serv\_Arm(t,i)$

Parâmetros

t = Transação que deseja acessar o dado
i = Índice do Elemento de Armazenamento a ser acessado

Variáveis locais

S_A = Servidor de Armazenamento
I_{AP} = Índice do Servidor de Armazenamento Primário
I_{AS} = Índice do Servidor de Armazenamento Secundário

Variáveis globais

T_{SA} = Tabela de Servidores de Armazenamento
T_T = Tabela de Transações

Algoritmo

$I_{AP} \leftarrow ((i-1) \bmod N_{SA}) + 1$
se $T_{SA}[I_A]$.Estado = DISPONÍVEL **então**
  $S_A \leftarrow T_{SA}[I_{SA}]$.Endereço
  retornar S_A
senão
  se $T_T[t]$.Crítica = **verdadeiro então**
    $I_{AS} \leftarrow (I_{AP} + ((I_{AP} - 1) \div N_{SA}) \bmod (N_{SA} - 1)) \bmod N_{SA} + 1$
    $S_A \leftarrow T_{SA}[I_{SA}]$.Endereço
    retornar S_A
  fim-se
fim-se
retornar erro

Figura A.2: Algoritmo de Mapeamento do Servidor de Armazenamento

$ST\_Mapear\_Serv\_Arm\_Sec(i)$

Parâmetros

t = Transação que deseja acessar o dado
i = Índice do Elemento de Armazenamento a ser acessado
S_{AP}= Servidor de Armazenamento Primário

Variáveis locais

S_{AS}= Servidor de Armazenamento Secundário
I_{AP} = Índice do Servidor de Armazenamento Primário
I_{AS} = Índice do Servidor de Armazenamento Secundário

Variáveis globais

T_{SA}= Tabela de Servidores de Armazenamento
T_T = Tabela de Transações

Algoritmo

$I_{AP} \leftarrow ((i-1) \bmod N_{SA}) + 1$
se $T_{SA}[I_{AP}]$.Estado = DISPONÍVEL **então**
    $I_{AS} \leftarrow (I_{AP} + ((I_{AP} - 1) \div N_{SA}) \bmod (N_{SA} - 1)) \bmod N_{SA} + 1$
    $S_{AS} \leftarrow T_{SA}[I_{AS}]$.Endereço
    retornar S_{AS}
senão
    se $T_{SA}[I_{AP}]$.Estado = RECUPERAÇÃO **então**
        $S_{AS} \leftarrow T_{SA}[I_{AP}]$.Endereço
    fim-se
fim-se
retornar erro

Figura A.3: Algoritmo de Mapeamento do Servidor de Armazenamento Secundário

ST_Leitura_Elemento$(T,\ i)$

Parâmetros

t = Transação
i = Índice do Elemento de Armazenamento a ser lido

Variáveis locais

e = Elemento de Armazenamento
S_A = Servidor de Armazenamento
rc = Código de Retorno

Variáveis globais

T_T = Tabela de Transações

Algoritmo

S_A = ST_Mapear_Serv_Arm(t, i)
se não há servidor disponível **então**
  erro
fim-se
rc = SA_Obter_Elemento(S_A, t, i)
se rc = **erro então**
  erro
fim-se
se $T_T[t]$.Crítica = **falso então**
  se rc = INSEGURO **então**
    $T_T[t]$.Serv_Inseguro $\leftarrow$ $T_T[t]$.Serv_Inseguro $\cup$ $\{S_A\}$
  fim-se
fim-se
retornar E
fim-se

Figura A.4: Algoritmo de Leitura de Elementos de Dados

$ST\_Leitura\_Simples(C_L,i)$

Parâmetros

C_L = Cliente que solicitou a leitura
i = Índice do Elemento de Armazenamento a ser lido

Variáveis locais

e = Elemento de Armazenamento
S_A = Servidor de Armazenamento
rc = Código de Retorno

Variáveis globais

T_T = Tabela de Transações

Algoritmo

S_A = ST_Mapear_Serv_Simples(i)
se não há servidor disponível **então**
  erro
fim-se
rc = SA_Obter_Elemento_Simples(S_A, C_L, i)
se rc = **erro então**
  erro
fim-se
retornar E

Figura A.5: Algoritmo de Leitura Simples de Elementos de Dados

$ST\_Atualização\_Elemento(t,\ i,\ e)$

Parâmetros
t = Transação que deseja atualizar o dado
i = Índice do Elemento de Armazenamento a ser atualizado
e = Elemento de Armazenamento a ser atualizado

Variáveis locais
S_{AP}= Servidor de Armazenamento Primário
S_{AS}= Servidor de Armazenamento Secundário

Variáveis globais
T_T = Tabela de Transações

Algoritmo
$S_{AP} \leftarrow$ ST_Mapear_Serv_Arm(t, i)
se não há servidor disponível **então**
    erro
fim-se
se T_T.Serv_Modificado = $\emptyset$ **então**
    ST_Registrar(t, INÍCIO, ($Critic$, T_T.Cliente))
fim-se
se $T_T[t]$.Crítica = **verdadeiro então**
    $S_{AP} \leftarrow$ ST_Mapear_Serv_Arm_Sec(t, i)
    se não há servidor disponível **então**
        erro
    fim-se
    SA_Atualizar_Elemento(S_{AP}, t, i, e, $Crítico$)
    se erro na atualização **então**
        erro
    fim-se
    $T_T[t]$.Serv_Modificado $\leftarrow$ $T_T[t]$.Serv_Modificado $\cup$ $\{S_{AP}, S_{AS}\}$
senão
    SA_Atualizar_Elemento(S_{AP}, t, i, e, $Crítico$)
    se erro na atualização **então**
        erro
    fim-se
    SA_Atualizar_Elemento(S_{AS}, t, i, e, $Crítico$)
    se erro na atualização **então**
        erro
    fim-se
    $T_T[t]$.Serv_Modificado $\leftarrow$ $T_T[t]$.Serv_Modificado $\cup$ $\{S_{AP}\}$
    $T_T[t]$.Serv_Inseguro $\leftarrow$ $T_T[t]$.Serv_Inseguro $\cup$ $\{S_{AP}\}$
fim-se

Figura A.6: Algoritmo de Alteração de Elementos de Dados

$ST\_Iniciar\_Transação(C_L, Crític)$

Parâmetros

C_L = Cliente que iniciou a transação
$Crit$= Indica se a transação é crítica

Variáveis locais

t = Transação a ser iniciada

Variáveis globais

T_T = Tabela de Transações
T_{ST} = Tabela de Servidores de Transação
S_A = Servidor de Transação que executa a operação

Algoritmo

$t \leftarrow$ ST_Gerar_TID()
$T_T \leftarrow T_T \cup (t,$ NÃO-PREPARADO, C_L, $Crític$, **verdadeiro** , 0, $\emptyset$, $\emptyset$)
$T_{ST}[S_A]$.Carga $\leftarrow T_{ST}[S_A]$.Carga + 1
retornar t

Figura A.7: Algoritmo de Início de Transação

ST_Terminar_Transação(t)

Parâmetros

t = Transação a ser Terminada

Variáveis locais

S_A = Servidor de armazenamento
L_V = Lista de Votos

Variáveis globais

T_T = Tabela de Transações
T_{ST} = Tabela de Servidores de Transação
S_A = Servidor de Transação que executa a operação

Algoritmo

```
se T_T[t].Serv_Modificado ≠ ∅ então
    para todo S_A ∈ T_T[t].Serv_Modificado faça
        NO_Enviar_Mensagem(S_A, (t, PREPARAR))
    fim-para
    L_V ← ST_Esperar_Votos(t, T_T[t].Serv_Modificado)
    se timeout então
        ST_Abortar_Transação(t)
    fim-se
    se ABORTAR ∈ L_V então
        ST_Abortar_Transação(t)
    senão
        ST_Registrar_Firme(t, COMMIT, (T_T[t].Serv_Modificado))
        para todo S_A ∈ T_T[t].Serv_Modificado faça
            NO_Enviar_Mensagem(S_A, (t, TERMINAR))
        fim-para
        ST_Registrar(t, TÉRMINO)
    fim-se
    se T_T[t].Crítica = falso então
        T_T[t].Estado ← NÃO-REPLICADO
    senão
        T_T ← T_T - T_T[t]
        T_ST[S_A].Carga ← T_ST[S_A].Carga - 1
    fim-se
senão
    T_T ← T_T - T_T[t]
    T_ST[S_A].Carga ← T_ST[S_A].Carga - 1
fim-se
```

Figura A.8: Algoritmo de Término de Transação

ST_Abortar_Transação(t)

Parâmetros
t = Transação a ser abortada

Variáveis locais
S_A Servidor de armazenamento

Variáveis globais
T_T = Tabela de Transações
T_{ST} = Tabela de Servidores de Transação
S_A = Servidor de Transação que executa a operação

Algoritmo
ST_Registrar(t, ABORTAR
para todo $S_A \in T_T[t]$.Serv_Modificado **faça**
NO_Enviar_Mensagem(S_A, (t, ABORTAR))
fim-para
ST_Registrar(t, TÉRMINO)
$T_T \leftarrow T_T$ - $T_T[t]$
$T_{ST}[S_A]$.Carga $\leftarrow T_{ST}[S_A]$.Carga - 1

Figura A.9: Algoritmo de Aborto de Transação

ST_Gerenciar_Replicação()

Variáveis locais
S_A = Servidor de Armazenamento

Variáveis globais
G = Grupo de Replicação Atual
L_V = Lista de Votos

Algoritmo
sempre faça
$G \leftarrow G + 1$
para todo S_A **faça**
NO_Enviar_Mensagem(S_A, (NOVO_GRUPO, G))
fim-para
$L_V \leftarrow$ ST_Esperar_Votos($S_1, \ldots, S_N$)
se timeout **então**
ST_Abortar_Grupo(G)
fim-se
se ABORTAR $\in L_V$ **então**
ST_Abortar_Grupo(G)
fim-se
ST_Esperar_Periodo_Grupo()
fim-sempre

Figura A.10: Algoritmo de Gerenciar Replicação

ST_Abortar_Grupo(G)

Parâmetros
G = Grupo de replicação a ser abortado

Variáveis locais
S_A = Servidor de armazenamento

Variáveis globais
T_T = Tabela de Transações

Algoritmo
ST_Registrar(t, ABORTAR
para todo S_A **faça**
NO_Enviar_Mensagem(S_A, (G, ABORTAR))
fim-para

Figura A.11: Algoritmo de Aborto de Grupo

$ST\_Falha\_Serv\_Armazenamento(S_A)$

Variáveis locais
r = Registro do log

Variáveis globais
S_A = Servidor de Armazenamento que falhou
T_T = Tabela de transações pendentes
T_{SA} = Tabela de Servidores de Armazenamento

Algoritmo
para todo $t \in T_T$ **faça**
  se $S_A \in T_T[t]$.Lista_Serv_Arm **então**
    ST_Abortar_Transação(t)
  fim-se
  $T_{SA}[S_A]$.Estado $\leftarrow$ INDISPONÍVEL
fim-para

Figura A.12: Algoritmo de recuperar falha de servidor de armazenamento

A.6 Operações dos Servidores de Armazenamento

Esta seção descreve o pseudo-código das operações realizadas pelos servidores de armazenamento.

$SA\_Obter\_Elemento(S_A,\ Crit,\ i,\ C_L)$

Parâmetros
S_A = Servidor de Armazenamento
$Crit$= Indica se a transação que deseja o dado é crítica
i = Índice do Elemento de Armazenamento a ser lido
C_L = Cliente que requisitou o dado

Variáveis locais
e = Elemento de Armazenamento
S_A = Servidor de Armazenamento

Variáveis globais
T_S = Tabela de Transações Servidas

Algoritmo
```
SA_Lock(i, t, Share)
e = SA_Buscar_Elemento(i)
se erro então
    erro
fim-se
se Crit = verdadeiro e e.Crítica = falso então
    erro
fim-se
SA_Enviar_Mensagem(C_L, e)
se T_S[e.TID].Segura = falso então
    retornar INSEGURA
fim-se
retornar OK
```

Figura A.13: Algoritmo de Obtenção de Elementos de Dados

SA_Obter_Elemento_Simples(S_A, C_L, i)

Parâmetros

S_A = Servidor de Armazenamento
C_L = Cliente que requisitou o dado
i = Índice do Elemento de Armazenamento a ser lido

Variáveis locais

e = Elemento de Armazenamento
S_A = Servidor de Armazenamento

Variáveis globais

T_S = Tabela de Transações Servidas

Algoritmo

SA_Latch(i, Share)
e = SA_Buscar_Elemento(i)
se erro **então**
    erro
fim-se
SA_Unlatch(i)
NO_Enviar_Mensagem(C_L, e)
retornar OK

Figura A.14: Algoritmo de Obtenção Simples de Elementos de Dados

$SA\_Atualizar\_Elemento(S_A,\ i,\ c,\ Crít)$

Parâmetros

S_A = Servidor de armazenamento
i = Índice do elemento de armazenamento a ser atualizado
c = Conteudo do elemento a ser atualizado
$Crit$= Indica se a transação que deseja o dado é crítica

Variáveis locais

e_a = Valor anterior do elemento de Armazenamento
S_A = Servidor de Armazenamento
lsn = Registro do log

Variáveis globais

T_S = Tabela de transações servidas
T_{NR}Tabela de elementos não replicados

Algoritmo

```
SA_Lock(i, t, Exclusive)
se erro então
    erro
fim-se
e_a ← SA_Buscar_Elemento(i)
se t ∉ T_S então
    SA_Registrar(t, INÍCIO, (Crític))
    T_S ← T_S ∪ {(t, NÃO_PREPARADO, Crít, falso )}
fim-se
se Crít = verdadeiro então
    lsn ← SA_Registrar(t, Atualização, (i, e_a, e, 0))
senão
    se T_NR[i] ∉ T_NR então
        T_NR ← T_NR ∪ {(t, i, 0)}
    fim-se
    lsn ← SA_Registrar_Firme(t, Atualização, (i, e_a, e, T_NR[i].Último_LSN))
    T_NR[i].Último_LSN ← lsn
fim-se
e.Conteúdo ← c
e.TID ← t
e.Crític ← Crít
e.LSN ← lsn
SA_Escrever_Elemento(i, e)
se erro então
    erro
fim-se
fim-se
retornar OK
```

Figura A.15: Algoritmo de Atualização de Elementos de Dados

SA_Terminar_Transação(S_A, t)

Parâmetros

S_A = Servidor de armazenamento
t = Transação a ser desfeita

Variáveis locais

msg= Mensagem do Coordenador da Transação

Variáveis globais

T_S = Tabela de transações servidas

Algoritmo

```
SA_Registrar_Firme(t, PREPARADO)
T_S[t].Estado ← PREPARADO
NO_Enviar_Mensagem(t.S_T, (t,VOTO-SIM))
SA_Unlock_Tipo(t,Share)
msg ← NO_Aguardar_Mensagem(t.S_T)
se timeout ou msg = ABORTAR então
    SA_Unlock_ALL(t)
    SA_Abortar_Transação(t)
senão
    SA_Unlock_Tipo(t,Exclusive)
    SA_Registrar(t, TÉRMINO)
    se T_S[t].Crític = verdadeiro então
        T_S[t].Estado ← TÉRMINO
    senão
        T_S[t].Estado ← NÃO-REPLICADO
    fim-se
fim-se
```

Figura A.16: Algoritmo de Término de uma transação

SA_Desfazer_Atualizações$(t,\ fim)$

Parâmetros

t = Transação a ser desfeita
fim= Registro final do log a ser desfeito

Variáveis locais

lsn = Ponteiro para registro do log
l = Ponteiro para registro do log
r = Registro do log
e = Elemento de armazenamento a ser desfeito
e_a = Valor anterior do elemento de Armazenamento

Variáveis globais

T_S = Tabela de transações servidas

Algoritmo

```
lsn ← T_S.[t].Último_LSN
enquanto lsn ≠ fim faça
    r ← SA_Buscar_Registro(lsn)
    se r.Tipo = ATUALIZAÇÃO então
        SA_Latch(r.Elemento, Exclusive)
        l = SA_Registrar(t, COMPENSAÇÃO, (r.Página, r.ÚltimoLSN, r.Dados_Ant))
        e.LSN ← l
        e.IDT ← t
        e.Tipo ← T_S[t].Tipo
        e.Conteúdo ← r.Dados_Ant
        SA_Escrever_Elemento(r.Elemento, e)
        SA_Unlatch(r.Elemento)
        lsn ← r.Último_LSN
    senão
        se r.Tipo = COMPENSAÇÃO então
            lsn ← r.LSN_Desfazer
        senão
            lsn ← r.Último_LSN
        fim-se
    fim-se
fim-enquanto
```

Figura A.17: Algoritmo de Desfazer Atualizações

SA_Abortar_Transação(t)

Variáveis globais

T_S = Tabela de transações servidas

Algoritmo

SA_Registrar(t, ABORTO)
SA_Desfazer_Atualizações(t, 0)
$T_S \leftarrow T_S - T_S[t]$
SA_Registrar(t, TÉRMINO)

Figura A.18: Algoritmo de Aborto de uma Transação

SA_Reiniciar()

Parâmetros

Variáveis globais

T_S = Tabela de transações servidas
T_{EV}= Tabela de elementos voláteis
T_{SA}= Tabela de servidores de armazenamento
S_A = Servidor de armazenamento que executa o procedimento

Algoritmo

$T_{SA}[S_A]$.Estado $\leftarrow$ RECUPERAÇÃO
$T_S \leftarrow$ SA_Recuperar_TS()
$T_{EV} \leftarrow$ SA_Recuperar_TEV()
SA_Analisar_Reinicio()
SA_Refazer_Transações()
SA_Desfazer_Transações()
$T_S \leftarrow \emptyset$
SA_Check_Point()

Figura A.19: Algoritmo de Reinicio de um Servidor de Armazenamento

```
SA_Analisar_Reinicio()
    Variáveis locais
        r    = Registro do log
    Variáveis globais
        T_S  = Tabela de transações servidas
        T_EV = Tabela de elementos voláteis
    Algoritmo
        r ← SA_Início_Log()
        enquanto r ≠ ∅ faça
            selecione r.Tipo entre
                caso INÍCIO faça
                    T_S ← T_S ∪ (r.IDT, INICIAL, r.Crític, ∅ , ∅ )
                    T_S[r.TID].Último_LSN ← r.LSN
                fim-caso
                caso ATUALIZAÇÃO faça
                    T_S[r.TID].Último_LSN ← r.LSN
                    T_S[r.TID].LSN_Desfazer ← r.LSN
                    se r.Elemento ∉ T_EV então
                        T_EV ← T_EV ∪ (r.Elemento, r.LSN)
                    fim-se
                fim-caso
                caso COMPENSAÇÃO faça
                    T_S[r.IDT].Último_LSN ← r.LSN
                    T_S[r.IDT].LSN_Desfazer ← r.LSN_Desfazer
                    se r.Elemento ∉ T_EV então
                        T_EV ← T_EV ∪ (r.Elemento, r.LSN)
                    fim-se
                fim-caso
                caso PREPARAÇÃO faça
                    T_S[r.IDT].Estado ← PREPARADO
                    T_S[r.IDT].Último_LSN ← r.LSN
                fim-caso
                caso TÉRMINO faça
                    se T_S[r.IDT].Crític = verdadeiro então
                        T_S ← T_S - T_S[r.IDT]
                    senão
                        T_S[r.IDT].Estado ← NÃO-REPLICADO
                        T_S[r.IDT].Último_LSN ← r.LSN
                    fim-se
                fim-caso
                caso ABORTO faça
                    T_S[r.IDT].Estado ← ABORTADO
                    T_S[r.IDT].Último_LSN ← r.LSN
                fim-caso
                caso SEGURO faça
                    T_S ← T_S - T_S[r.IDT]
                fim-caso
            fim-selecione
            r ← SA_Próximo_Registro_Log(r)
        fim-enquanto
```

Figura A.20: Algoritmo de Analise de Reinício

SA_Refazer_Transações()

Variáveis locais
r = Registro do log
e = Elemento de armazenamento a ser refeito

Variáveis globais
T_S = Tabela de transações servidas
T_{EV} = Tabela de elementos voláteis

Algoritmo

```
r ← SA_Início_Log()
enquanto r ≠ ∅ faça
    se r.Tipo = ATUALIZAÇÃO ou r.Tipo = COMPENSAÇÃO então
        se r.Elemento ∈ T_EV e r.LSN ≤ T_EV[r.Elemento].LSN_Alt então
            e = SA_Buscar_Elemento(r.Elemento)
            se e.LSN < r.LSN então
                e.LSN ← r.LSN
                e.Conteúdo ← r.Dados_Nov
                SA_Escrever_Elemento(r.Elemento, e)
            senão
                T_EV[r.Elemento].LSN_Alt ← r.LSN
            fim-se
        fim-se
    fim-se
    r ← SA_Próximo_Registro_Log(r)
fim-enquanto
```

Figura A.21: Algoritmo de Refazer Transações

$SA\_Desfazer\_Transações()$

Variáveis locais
r = Registro do log

Variáveis globais
T_S = Tabela de transações servidas

Algoritmo
para todo $t \in T_S$ **faça**
  se $T_S[t]$.Estado = ABORTADO **então**
    SA_Desfazer_Atualizações(t, 0)
    SA_Registrar(t, TÉRMINO)
  senão
    se $T_S[t]$.Crític = **falso** **então**
      SA_Registrar(t, ABORTO)
      SA_Desfazer_Atualizações(t, 0)
      SA_Registrar(t, TÉRMINO)
    fim-se
  fim-se
fim-para

Figura A.22: Algoritmo de Desfazer Transações

$SA\_Propagar\_Atualizações()$

Variáveis locais

S_{AS} = Servidor Secundário
i = Índice de um elemento de armazenamento
g = Grupo de propagação
lsn = lsn
r = Registro no log

Variáveis globais

T_{NR} = Tabela de elementos não replicados
S_{AP} = Servidor que armazena o dado
S_{TR} = Servidor de transação que gerencia o processo de réplica
T_{SA} = Tabela de servidores de armazenamento
S_A = Servidor de armazenamento que executa o procedimento

Algoritmo

```
sempre faça
    G ← SA_Esperar_Mensagem(NOVO_GRUPO)
    se T_SA[S_A].Estado = RECUPERAÇÃO então
        T_SA[S_A].Estado ← BLOQUEADO
    senão
        se T_SA[S_A].Estado = BLOQUEADO então
            T_SA[S_A].Estado ← DISPONÍVEL
        fim-se
    fim-se
    para todo (g, i, lsn) ∈ T_NR tal que g = (G − 1) faça
        r ← SA_Buscar_Registro(lsn)
        S_AS ← NO_Mapear_Serv_Sec(i)
        NO_Enviar_Mensagem(S_AS, (PROPAGAR, g, i r.Dado_Nov))
        se erro então
            NO_Enviar_Mensagem(S_TR, (ABORTAR_GRUPO, g)) erro
        fim-se
    fim-para
    NO_Enviar_Mensagem(S_TR, (FIM_GRUPO, g))
fim-sempre
```

Figura A.23: Algoritmo para Propagar Réplicas

SA_Receber_Propagação()

Variáveis locais

S_{AS}= Servidor Secundário
i = Índice de um elemento de armazenamento
g = Grupo de propagação
lsn = lsn
d = Dados a serem propagados

Variáveis globais

T_{NR}= Tabela de elementos não replicados
S_{AP}= Servidor que armazena o dado
S_{TR}= Servidor de transação que gerencia o processo de réplica

Algoritmo

```
sempre faça
    msg ← NO_Esperar_Mensagem(NOVO_GRUPO,PROPAGAR,ABORTAR_GRUPO)
    selecione msg.Tipo entre
        caso NOVO_GRUPO faça
            G ← G + 1
        fim-caso
        caso PROPAGAR faça
            (g, i, d) ← msg.Corpo
            e_a ← SA_Buscar_Elemento(i)
            lsn ← SA_Registrar_Secundário(t, Atualização, (i, e_a, e))
            e.Conteúdo ← d
            e.TID ← g
            e.Crític ← falso
            e.LSN ← lsn
            SA_Latch(i, Exclusive)
            SA_Escrever_Elemento(i, e)
            se erro então
                erro
            fim-se
            SA_Unlatch(i)
        fim-caso
        caso ABORTAR_GRUPO faça
            SA_Abortar_Grupo(G)
        fim-caso
    fim-selecione
fim-sempre
```

Figura A.24: Algoritmo de Receber Propagações

$SA\_Falha\_Serv\_Transação(S_T)$

Parâmetros

S_T = Servidor de Transação

Variáveis locais

r = Registro do log

Variáveis globais

S_T = Servidor de Transações

T_S = Tabela de transações servidas

Algoritmo

```
para todo t ∈ T_S faça
    se t.S_T = S_T então
        SA_Abortar_Transação(t)
    fim-se
fim-para
```

Figura A.25: Algoritmo de recuperar falha de servidor de transação

PROPOSTA DE ARQUITETURA TRANSACIONAL DISTRIBUÍDA

Dissertação de Mestrado apresentada por **Lorenzo Francesco Giovanni Gino Maria Ridolfi**, em 28 de setembro de 1995 ao Departamento de Informática da PUC-Rio, e aprovada pela Comissão Julgadora formada pelos seguintes professores:

Prof. Marco Antônio Casanova (Co-orientador)
Centro Científico Rio — IBM Brasil

Prof. Luiz Fernando Gomes Soares (Co-orientador)
Departamento de Informática — PUC-Rio

Prof. Sergio Lifschitz
Departamento de Informática— PUC-Rio

Visto e permitida a impressão.
Rio de Janeiro, 28 de setembro de 1995.

Coordenador de Programas de Pós-Graduação do
CENTRO TÉCNICO CIENTÍFICO