Universidade Federal do Ceará

Mestrado em Ciência da Computação

Teste do Grafo de Serialização Temporal: Uma Estratégia para o Controle de Concorrência em Ambientes de Broadcast

José Maria da Silva Monteiro Filho

Dissertação de Mestrado

Fortaleza-CE

26 de outubro de 2001

Teste do Grafo de Serialização Temporal:
Uma Estratégia para o Controle de
Concorrência em Ambientes de Broadcast

José Maria da Silva Monteiro Filho

Dissertação de Mestrado

Teste do Grafo de Serialização Temporal: Uma Estratégia para o Controle de Concorrência em Ambientes de Broadcast

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por José Maria da Silva Monteiro Filho e aprovada pela Banca Examinadora.

Fortaleza, 26 de outubro de 2001.

Prof. Dr. Ângelo Roncalli Alencar Brayner
(Orientador)

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da UFC, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Mestrado em Ciência da Computação

Universidade Federal do Ceará

Teste do Grafo de Serialização Temporal: Uma Estratégia para o Controle de Concorrência em Ambientes de Broadcast

José Maria da Silva Monteiro Filho

Outubro de 2001

Banca Examinadora:

- Prof. Dr. Ângelo Roncalli Alencar Brayner
  Universidade de Fortaleza - UNIFOR
- Prof. Dr. Sérgio Lifschitz
  Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio
- Profa. Dra. Vânia Maria Ponte Vidal
  Universidade Federal do Ceará - UFC

Resumo

A disseminação de dados baseada em difusão (*broadcast*) está se transformando no principal modo de transferência de informações em ambientes de computação móvel e comunicação sem fio [23, 1]. Ambientes com suporte a mobilidade e com disseminação de dados do tipo broadcast apresentam diversas restrições, como por exemplo, um link com baixa largura de banda para a transmissão de dados dos clientes para os servidores (uplink). Além disso, os clientes móveis apresentam recursos limitados de memória e de processamento, além de apresentarem baixa autonomia com relação ao fornecimento de energia. As aplicações nestes ambientes necessitam ler dados atuais e consistentes. Contudo, os mecanismos tradicionais para controle de concorrência baseados em seriabilidade têm se mostrado impróprios para ambientes de broadcast. Esse fato é uma conseqüência das restrições na capacidade de comunicação dos clientes em ambientes com suporte a mobilidade. Este trabalho propõe um novo protocolo para controle de concorrência que utiliza a serializabilidade como critério de corretude. O protocolo proposto garante que os clientes não necessitam contatar o servidor para executar suas operações (para solicitar bloqueios, por exemplo). Os clientes móveis escutam o canal de broadcast apenas para retirar os itens de seu interesse, podendo desconectar-se por qualquer período de tempo. Além disso, não precisam armazenar e nem gerenciar estruturas complexas, como as propostas em [32] e em [31], para realizar o controle de concorrência.

Abstract

A broadcast environment is defined as a mobile computing environment in which data are delivered to mobile clients by means of a broadcast-based mechanism. Several restrictions arise in a broadcast environment. For example, the communication from clients to servers is realized through a low bandwidth. Another difficulty stems from the fact that mobile clients have scarce computational resources, such as memory. Additionally, battery power is a very limited resource. Applications running in a broadcast environment need to access data managed by database servers (database systems which run on server machines). Of course, those applications have to see the most recent consistent database state. For that reason, in such a scenario, database servers should synchronize operations for ensuring data consistency and currency of data. However, conventional serializability-based concurrency control protocols are unsuitable for synchronizing transactions in broadcast environments. The major goal of this work is to present a new serializability-based protocol to synchronize transactions in data intensive applications. The proposed protocol saves battery power, since it ensures that mobile clients do not have to contact servers (for requiring locks, for example) to access data. Thus, mobile clients do not need to listen to the broadcast continuously, they listen to the broadcast channel to retrieve data they need. That means, the proposed protocol supports client disconnections. Furthermore, the proposed protocol saves the use of mobile-clients computational resources. That is because it does require that clients have to manage complex structures as those proposed in [Shan99] and in [Pit99] for controlling concurrency in broadcast environments.

Sumário

Resumo **vi**

Abstract **vii**

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 4

1.3 Organização 5

2 Ambientes de Computação Móvel com Disseminação de Dados através de Broadcast **6**

2.1 Introdução 6

2.2 Taxonomia e Arquitetura Referência para um Ambiente de Computação Móvel 7

2.3 Características de um Ambiente de Computação Móvel 9

2.4 Ambientes de Broadcast 19

2.4.1 Arquitetura 21

2.4.2 Estrutura do Broadcast 22

3 Processamento de Transações em um Ambiente de Broadcast **25**

3.1 Introdução 25

3.2 O Modelo Clássico de Processamento de Transações 26

3.2.1 O Problema da Concorrência em Sistemas de Bancos de Dados 26

3.2.2 Transações 28
3.2.3 Execução Correta de Transações Concorrentes 30
3.2.4 Confiabilidade de Schedules 35
3.2.5 Protocolos de Controle de Concorrência 38
3.3 O Modelo de Transações em Ambientes de Computação Móvel 41
3.4 Controle de Concorrência em Ambientes de Broadcast 44

4 Trabalhos Correlatos 46
4.1 Introdução 46
4.2 Considerando Transações Somente de Leitura 46
4.3 Relatório de Invalidação 48
4.4 Invalidação Baseada em Versão 49
4.5 Múltiplas Versões 50
4.6 Teste do Grafo de Serialização (SGT) 51
4.6.1 Implementando o método SGT 52
4.7 Consistência de Atualização 53
4.7.1 Critério de Corretude 54
4.7.2 Implementando a Consistência de Atualização (F_MATRIX) . 54
4.8 Quadro Comparativo 57
4.9 Serializabilidade Semântica 60
4.9.1 O Modelo 61

5 Teste do Grafo de Serialização Temporal 69
5.1 Introdução 69
5.2 Teste do Grafo de Serialização Temporal 70
5.2.1 Cenário Exemplo 71
5.2.2 Protocolo TGST Básico 73
5.2.3 Protocolo TGST Estendido 75
5.2.4 Considerações Sobre a Comunicação das Leituras 80
5.2.5 Pontos de Falha na Comunicação 81

5.2.6 Um Método Coletor de Lixo 82
5.2.7 Considerações Sobre o Protocolo TGST 83
5.3 Atualização nos Clientes Móveis 83
5.4 Teste do Grafo de Serialização Temporal-Semântico 89
5.4.1 O Protocolo TGSTSe 90

6 Conclusões **92**

Bibliografia **94**

Lista de Figuras

2.1 Arquitetura para um Ambiente de Computação Móvel. 8
2.2 Disseminação de dados baseada em broadcast 21
2.3 Uma Possível Estrutura para a Organização do Broadcast 24

3.1 Classificação dos Protocolos de Controle de Concorrência. 39
3.2 Schedules Locais S_1 e S_2 43
3.3 Schedule Global SG 43

4.1 Comparação entre os principais mecanismos para controle de concorrência em ambientes de broadcast. 59
4.2 Organização de uma empresa seguradora. 61
4.3 Schedule S. 67
4.4 Grafo de serialização semântica para o schedule S. 67

5.1 Schedule SG. 72
5.2 Grafo de serialização do Schedule SG. 72
5.3 Schedule SG': Schedule SG com informações atrasadas. 73
5.4 Grafo de serialização do Schedule SG com informações atrasadas. . . 73
5.5 Schedule SG'. 76
5.6 Grafo de serialização correto do Schedule SG'. 76
5.7 Schedule SG''. 77
5.8 Grafo de serialização do Schedule SG''. 77
5.9 Grafo de serialização correto do Schedule SG''. 77

5.10 Schedule SG_U. 85
5.11 Grafo de Serialização Temporal para o Schedule SG_U. 86
5.12 Schedule SG_C. 86
5.13 Grafo de Serialização Temporal para o Schedule SG_C. 87
5.14 Schedule SG'_U. Schedule SG_U com informações atrasadas. 88
5.15 Grafo de Serialização para o Schedule SG'_U. 88
5.16 Schedule SG'_C. Schedule SG_C com informações atrasadas. 88
5.17 Grafo de Serialização para o Schedule SG'_C. 89

Teste do Grafo de Serialização Temporal: Uma Estratégia para o Controle de Concorrência em Ambientes de Broadcast

José Maria da Silva Monteiro Filho

26 de outubro de 2001

Capítulo 1

Introdução

1.1 Motivação

A integração dos computadores portáteis com as recentes tecnologias de comunicação celular, redes de comunicação sem fio e serviços via satélite está proporcionando o surgimento de um novo paradigma em computação, denominado de computação móvel. Este paradigma fornece o suporte para um ambiente computacional no qual os usuários, de posse de computadores portáteis, têm acesso a informações e recursos compartilhados independente de onde estejam localizados e de sua mudança de localização (mobilidade). Portanto, o termo "móvel" implica na capacidade que um computador possui de mover-se enquanto mantém uma conexão com uma infra-estrutura fixa de comunicação. Neste contexto, um operador da bolsa de valores, utilizando um computador portátil (móvel), pode participar dos pregões do mercado financeiro ao mesmo tempo em que realiza uma viagem de negócios.

Vale ressaltar que o principal objetivo da computação móvel é prover aos usuários uma plataforma computacional com as seguintes características: um conjunto de serviços similares aos existentes num sistema distribuído tradicional e que, adicionalmente, permita a mobilidade.

Neste cenário, um dispositivo portátil (*Handheld*, *PDA*, *Laptop*,...) é capaz de

movimentar-se livremente enquanto interage com uma rede de computadores fixos, através de um *link* (conexão) sem fio, podendo, por exemplo, executar consultas em banco de dados localizado na rede fixa. Assim, um computador móvel (cliente) pode requisitar serviços disponibilizados por computadores fixos (servidores). A comunicação entre computadores móveis e fixos é realizada através das estações de suporte a mobilidade (*MSS - Mobile Support Station*). Cada *MSS* é responsável por uma determinada região geográfica, chamada de célula. Desta forma, a principal atribuição de uma *MSS* é viabilizar a comunicação dos computadores móveis fisicamente localizados em sua célula com os computadores da rede fixa.

Este novo ambiente computacional apresenta características bastantes particulares. Um computador móvel pode passar longos períodos desconectado devido às limitações de energia da bateria e da própria mobilidade da máquina, pois pode mover-se para uma área não coberta pelo sistema de comunicação. Isto faz com que as transações móveis que acessam dados dos servidores sejam de longa duração. Além disso, as conexões sem fio apresentam uma limitada largura de banda para a comunicação entre as unidades móveis e as *MSS's*, o que restringe a capacidade dos clientes móveis no que diz respeito ao envio de solicitações aos servidores.

A tecnologia de computação móvel proporciona o suporte necessário para a execução de sofisticadas aplicações, muitas das quais envolverão dados onde a consistência deverá ser mantida apesar das atualizações, que poderão ocorrer até mesmo nos clientes móveis. Porém, devido às suas características, torna-se de fundamental importância reexaminar os problemas referentes ao processamento de transações em sistemas de computação móvel. Os mecanismos tradicionais para controle de concorrência baseados em serializabilidade, como por exemplo, bloqueio, *timestamp* e consistência de *cache*, têm se mostrado apropriados para aplicações convencionais de bancos de dados nas quais as transações são tipicamente de curta duração. Entretanto, estes mecanismos requerem uma excessiva comunicação entre clientes e servidores e rejeitam certas execuções corretas, o que torna tais mecanismos impróprios para ambientes de computação móvel [32].

Recentemente, o modo de disseminação de dados baseado em difusão (push-based) vem ganhando grande destaque e já está se transformando no principal modo de transferência de informações em ambientes de computação móvel e comunicação sem fio [23, 1]. Nesta abordagem o servidor repetidamente difunde (*broadcast*) os itens de dados (objetos do banco de dados) para uma população de clientes sem que haja uma requisição específica. Os clientes, por sua vez, monitoram o canal de *broadcast* e retiram os itens de dados que necessitam. Vale ressaltar que tradicionalmente os dados são enviados dos servidores para os clientes sob demanda (abordagem *pull-based*). Os ambientes de computação móvel com disseminação de informações através de *broadcast* apresentam diversas vantagens. O servidor na rede fixa não fica sobrecarregado com pedidos de requisições e não envia várias mensagens individuais que teriam que ser transmitidas em sistemas *pull-based*. Além disso, os dados podem ser acessados concorrentemente por qualquer número de clientes sem nenhuma degradação de performance. Aplicações que apresentam como características um grande número de clientes móveis, um pequeno número de servidores e um banco de dados relativamente pequeno podem se beneficiar do modo de disseminação de dados baseado em difusão. Como exemplo podemos citar as aplicações de comércio eletrônico, tais como leilões e propostas eletrônicas, sistemas de controle de tráfego e automação industrial [32].

As aplicações em ambientes de *broadcast* necessitam ler dados atuais e consistentes. Entretanto, os mecanismos tradicionais para controle de concorrência têm se mostrado impróprios para estes ambientes. Assim, torna-se de fundamental importância a concepção de novos mecanismos que garantam, de forma eficiente, a consistência dos dados em ambientes de *broadcast*.

1.2 Objetivos

Com o objetivo de solucionar eficientemente o problema do controle de concorrência em ambientes de *broadcast*, várias propostas têm sido apresentadas. Porém, a maioria dessas abordagens requer que estruturas complexas sejam enviadas aos clientes móveis. Assim, os clientes têm que permanecer mais tempo em seu estado ativo para gerenciar essas estruturas, além de terem que armazena-las localmente. Propostas como o relatório de invalidação [31] e consistência de atualização [32] apresentam essas desvantagens. Por exemplo, em [32] é proposto que uma matriz n x n seja enviada aos clientes móveis, onde n representa o número de objetos do banco de dados.

Neste trabalho, identificamos e investigamos as principais propostas para o controle de concorrência em ambientes de broadcast. Além disso, propomos um novo mecanismo, denominado teste do grafo de serialização temporal (TGST) [9], o qual explora informações temporais referentes ao momento em que um objeto do banco de dados foi lido ou atualizado.

O protocolo TGST evita o envio de estruturas complexas para os clientes móveis, o que reduz o tráfego de comunicação entre o servidor e os clientes, e minimiza o tempo em que o cliente necessita escutar o canal de broadcast. Além disso, nesta abordagem, os clientes só precisam armazenar os valores dos objetos de seu real interesse. Dessa forma, o protocolo proposto pode garantir uma economia no custo de utilização dos canais de comunicação, na limitada capacidade de energia dos computadores portáteis e nos seus escassos recursos de memória.

1.3 Organização

A dissertação está organizada da seguinte forma: o capítulo 2 discute os ambientes de computação móvel com disseminação de informações baseada em *broadcast*; o capítulo 3 faz uma introdução ao problema do controle de concorrência em ambientes de *broadcast*; o capítulo 4 descreve o estado da arte, fazendo uma comparação entre as diversas soluções propostas; o capítulo 5 apresenta novos mecanismos para o controle de concorrência em ambientes de *broadcast*; o capítulo 6 conclui esta dissertação e aponta trabalhos futuros.

Capítulo 2

Ambientes de Computação Móvel com Disseminação de Dados através de Broadcast

2.1 Introdução

Devido aos recentes avanços da computação e comunicação sem fio podemos afirmar que a informática está entrando em uma nova revolução, tão profunda quanto as que ocorreram com o surgimento dos computadores pessoais e das redes de computadores [25]. A massificação dos dispositivos computacionais móveis e sua integração com as redes de comunicação sem fio irão possibilitar o acesso à informação em qualquer lugar e a qualquer momento. Este novo cenário recebe a denominação de computação móvel. A computação móvel irá alterar profundamente a maneira como as pessoas trabalham, estudam e usam seu tempo, além de mudar a forma como as empresas oferecem seus produtos e serviços e interagem com seus clientes.

Neste capítulo descreveremos e analisaremos as principais propriedades e características de um ambiente de computação móvel. Especialmente aqueles que utilizam difusão como forma de disseminação de informações. Este capítulo está

organizado como descrito a seguir. Na seção 2.2 apresentamos a arquitetura referência para um ambiente de computação móvel. A seção 2.3 discute as características, as limitações e os problemas referentes aos ambientes de computação móvel. Na seção 2.4, analisaremos os ambientes de computação móvel baseados em *broadcast*.

2.2 Taxonomia e Arquitetura Referência para um Ambiente de Computação Móvel

Os avanços na tecnologia de comunicação celular, redes de comunicação sem fio e serviços via satélite estão proporcionando o surgimento de um novo paradigma em computação, chamado de computação móvel. Neste novo ambiente, os usuários, de posse de computadores portáteis, têm acesso a informações e recursos compartilhados independente de onde estes usuários estejam localizados e de sua mudança de localização (mobilidade) [25].

O termo "móvel" implica na capacidade que um computador possui de mover-se enquanto mantém uma conexão com uma infra-estrutura fixa de comunicação. Assim, de maneira semelhante aos ambientes de computação distribuída, um dispositivo computacional tem acesso a serviços, recursos e dados distribuídos; e, adicionalmente, pode mover-se livremente. Desta forma, a computação móvel amplia o conceito tradicional de computação distribuída. Um modelo abstrato de uma arquitetura para um ambiente de computação móvel é mostrada na figura 2.1. Esta arquitetura consiste de um conjunto de computadores móveis (MH - Mobile Host) e uma rede de alta velocidade que interconecta fisicamente computadores fixos (FH - Fixed Host) e estações de suporte à mobilidade (MSS - Mobile Support Station). Um computador móvel é um dispositivo de computação inteligente que pode movimentar-se livremente ao mesmo tempo em que mantém sua conexão com a rede fixa através de um *link* (conexão) sem fio. As estações de suporte à mobili-

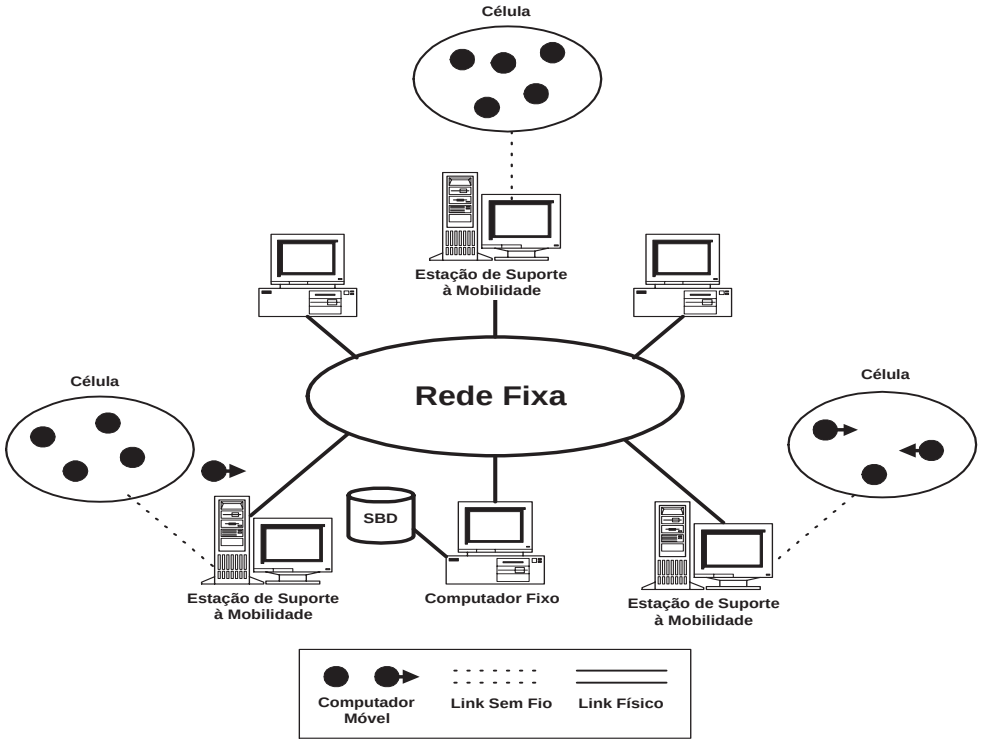

Figura 2.1: Arquitetura para um Ambiente de Computação Móvel.

dade fornecem uma interface sem fio que permite aos computadores móveis interagir com a rede fixa, tendo acesso a informações e recursos compartilhados. Cada MSS é responsável por uma determinada região geográfica, chamada de célula, tendo como uma de suas tarefas o endereçamento dos computadores móveis localizados nesta região. Uma MSS somente pode se comunicar com as unidades móveis que estiverem fisicamente localizada em sua célula. Um computador fixo é um computador de propósito geral, que pode disponibilizar serviços para os computadores móveis, mas que não é capaz de comunicar-se diretamente com estes [25].

Em uma plataforma de computação móvel, o computador móvel exerce o papel de cliente que requisita serviços disponibilizados por servidores (computadores fixos) localizados na rede fixa. Assim, podemos nos referir aos computadores móveis como clientes, ou usuários. Clientes móveis e estações de suporte à mobilidade se comunicam através de canais de comunicação sem fio. Este canal, em geral, consiste de dois *links*; o *uplink*, utilizado para mover dados dos clientes para as MSS's, e o

downlink, usado para transportar dados das MSS's para os clientes.

2.3 Características de um Ambiente de Computação Móvel

Nesta seção iremos apresentar e discutir as principais características de uma plataforma computacional baseada no paradigma de computação móvel. Analisaremos ainda os principais problemas introduzidos pela tecnologia de computação móvel.

Como afirmado anteriormente, um ambiente de computação móvel apresenta características bastantes particulares [22]. Dentre elas destacamos:

(i) **Limitação de Energia nos Computadores Móveis.**

Os computadores móveis dependem de baterias para funcionar. Atualmente, as baterias disponíveis no mercado são relativamente pesadas e só conseguem armazenar energia para algumas horas de uso. Este problema é visto como o maior empecilho no uso de computadores móveis. Infelizmente, a tecnologia de construção de baterias não tem acompanhado o crescimento de outros segmentos da informática e a evolução prevista não muda esse cenário [34]. Logo, o gerenciamento de energia é um problema importante e deve ser tratado tanto pelo *hardware* quanto pelo *software*. Portanto, energia é um recurso limitado em computadores móveis e o seu consumo deve ser minimizado. A fim de conservar energia e estender o tempo de vida da bateria, o cliente entra no modo *doze* (*stand by mode*), no qual o cliente não está ativamente escutando o canal de comunicação com o servidor. Os clientes gastam uma quantidade de energia significativamente menor no modo *doze* que no modo ativo, logo uma das principais metas em computação móvel é minimizar o tempo que o cliente deve gastar no modo ativo para recuperar os itens de dados do seu interesse.

(ii) **Longa Desconexão dos Clientes.**

Um computador móvel pode passar longos períodos desconectado devido às limitações de energia da bateria e da própria mobilidade da máquina, pois pode mover-se para uma área não coberta pelo sistema de comunicação.

(iii) **Transações de Longa Duração.**

A eventual e freqüente desconexão dos clientes pode fazer com que as transações móveis que acessam dados dos servidores sejam de longa duração.

(iv) **Deve ser Escalonável.**

O número de clientes nas aplicações em ambientes móveis tende a crescer rapidamente. Desta forma, uma plataforma de computação móvel deve estar apta a suportar um grande número de clientes.

(v) **Comunicação Assíncrona entre Clientes e MSS's.**

Em uma rede de comunicação sem fio o custo para manter uma conexão *statefull* chega a ser proibitivo. Por isso, nestes ambientes, as estações de suporte a mobilidade têm uma largura de banda relativamente alta para disseminar informações (*downlink*), enquanto os clientes não podem transmitir dados ou se o fazem é sobre um *link* com baixa largura de banda (*uplink*).

A combinação da comunicação sem fio com a mobilidade dos computadores introduziu novos problemas nas áreas de redes de computadores, sistemas de informação, sistemas de bancos de dados e sistemas operacionais. De uma perspectiva da tecnologia de bancos de dados, podemos identificar os seguintes problemas introduzidos pelo paradigma de computação móvel [3, 21, 23, 25]:

- **Problemas Referentes a Mobilidade**

  Em um ambiente de computação móvel, os usuários podem estar em constante deslocamento, ao contrário das tradicionais arquiteturas cliente/servidor onde tanto clientes quanto servidores possuem uma posição fixa na rede. Esta característica introduz novos problemas no gerenciamento dos dados. A seguir analisaremos os principais problemas introduzidos pela propriedade da mobilidade.

  (i) Localização de Usuários

  Em um ambiente de computação móvel, a localização do usuário pode ser considerada como um item de dado cujo valor é atualizado a cada movimento deste usuário. Assim, a localização passa a ser uma porção de dados freqüentemente "mutável". Estabelecer uma conexão requer o conhecimento da localização da parte com a qual queremos estabelecer a conexão. Contudo, localizar um determinado usuário é o mesmo que ler o item de dado referente à sua localização. Esta leitura pode envolver uma consulta sobre um banco de dados como também uma busca extensiva através da rede. A tarefa de atualizar a localização de um usuário consiste numa operação de escrita sobre a "variável" que representa a localização do usuário, podendo envolver a atualização de itens de dados em um banco de dados local ou remoto. A localização de um usuário pode também ser usada por consultas complexas, como por exemplo, "encontre o número de policiais em um estádio" ou "encontre o médico mais próximo de um acidente". Assim, a localização dos usuários pode ser tratada como uma porção de dados que pode ser atualizada e consultada. Desta forma, o gerenciamento da localização é um problema de gerenciamento de dados.

(ii) Bancos de Dados Móveis

O armazenamento de dados nos computadores móveis implica que o acesso aos dados terá um custo adicional na comunicação, pois será necessário localizar o computador móvel, e um consumo maior dos limitados recursos de energia por parte dos clientes. Além disso, os dados podem não estar disponíveis devido às freqüentes desconexões dos clientes móveis.

(iii) Consultas Sobre Informações Dependentes da Localização

A computação móvel está criando a necessidade de novas e sofisticadas consultas, como por exemplo, "encontre o número de policiais em um estádio" ou "encontre o médico mais próximo de um acidente". Para responder tais perguntas podemos basicamente utilizar duas abordagens: na primeira abordagem o processamento das consultas baseiam-se somente nas informações do banco de dados (ou seja, informações armazenadas nos servidores de localização). Uma vez que os valores dos itens no banco de dados podem ser imprecisos, devido à constante movimentação dos objetos envolvidos no sistema, a resposta da consulta também será imprecisa e passível de erros. Por outro lado, o esforço gasto com comunicação para a obtenção da localização dos objetos relevantes para esta consulta é nulo. Na segunda abordagem, o processamento de consultas envia mensagens adicionais para encontrar a exata localização dos objetos que são relevantes para a consulta. Aqui, existe um gasto adicional com a comunicação, porém, podemos obter uma resposta mais precisa. Obviamente, devem ser trocadas o menor número de mensagens possível, uma vez que o custo de comunicação é sempre substancial. Desta forma, estamos diante de um impasse entre custo de comunicação e a precisão necessária para responder uma consulta. O desafio atual dos pesquisadores consiste em obter máxima precisão com um custo de comunicação fixo.

- **Problemas Referentes a Desconexão.**

  A principal diferença entre desconexão e falha é sua natureza eletiva: A desconexão pode ser tratada como uma falha planejada, a qual pode ser antecipada. Dessa forma, o sistema de computação móvel deve estar preparado para reagir a uma situação de desconexão. Portanto, podem existir vários graus de desconexão variando de uma conexão *stateful* até uma conexão fraca ou parcial, isto é, o computador móvel está conectado ao resto da rede via um canal de rádio com baixa largura de banda. Dois problemas críticos introduzidos pela desconexão dos clientes móveis são discutidos a seguir:

  (i) *HandOff*

  Como já mencionamos anteriormente, em um ambiente de computação móvel cada célula é gerenciada por uma estação de suporte a mobilidade (MSS). Assim, os dispositivos móveis localizados em uma determinada célula interagem com a rede fixa através de uma conexão sem fio com a MSS responsável por esta célula. Quando um computador móvel movimenta-se entre duas diferentes células ocorre uma desconexão seguida de uma reconexão deste dispositivo com a MSS responsável pela nova célula. Este processo é chamado de *HandOff*. Desta forma, novos problemas surgem quando o usuário móvel tem que se mover durante a execução de uma transação e continuar sua execução em uma nova célula ou quando o usuário tem que se desconectar no meio de uma transação.

  (ii) Consistência de *Cache*

  O *caching* de itens de dados freqüentemente acessados é uma técnica importante que visa reduzir a utilização da limitada largura de banda dos canais de comunicação sem fio. Os dados armazenados em *cache* podem ajudar a aumentar o tempo de resposta das consultas e dar suporte para que operações de atualização e consulta possam ser executadas quando o usuário está desconectado ou fracamente conectado ao restante

do sistema de comunicação. Entretanto, as estratégias para a consistência de *cache* são severamente limitadas por dois motivos: a desconexão e a mobilidade dos clientes. Isto decorre do fato de que um servidor pode não ser conhecedor da localização atual dos clientes e nem do *status* das conexões dos clientes. Os algoritmos existentes para consistência de *cache* podem ser divididos em duas categorias:

Abordagens *Callback*: Os servidores enviam mensagens de invalidação diretamente aos clientes que têm itens de dados armazenados em *cache* para que estes sejam atualizados.

Abordagem de Detecção: Os clientes enviam consultas aos servidores para validar os dados em *cache*.

Uma vez que os clientes móveis podem estar desconectados a fim de conservar a energia das baterias e estão freqüentemente se movendo, a abordagem *Callback* não é facilmente implementada am ambientes móveis. Por outro lado, a limitada largura de banda das redes sem fio são um obstáculo quando um grande número de clientes deseja consultar o servidor para validar os dados em *cache*. Além disso, em ambientes sem fio, os clientes móveis usualmente têm uma pequena capacidade de transmissão por causa das limitações de bateria enquanto os servidores possuem um *link* com alta largura de banda para *broadcast*. Ambas as abordagens não são aplicáveis em ambientes de computação móvel. Caso os dados armazenados em *cache* também estejam sendo atualizados pelo usuário, esquemas de controle de concorrência são necessários.

- **Problemas Quanto aos Modos de Acesso a Dados**

  O desenvolvimento de novos métodos de acesso aos dados é motivado pelo avanço nos meios de transmissão sem fio e pelas limitações existentes nos clientes móveis, como por exemplo, restrições no fornecimento de energia, longos períodos de desconexão e a impossibilidade de se manter uma conexão *statefull* com os servidores. Devido a estas características, o acesso aos dados ganha grande importância e torna-se parte fundamental em ambientes com suporte a mobilidade. A seguir discutiremos as principais questões referentes aos mecanismos de acesso aos dados em sistemas de computação móvel.

  (i) Segurança

  A segurança é sem dúvidas o problema mais polêmico em ambientes de computação móvel. Atualmente, ele está sendo ignorado o que é típico em uma área de pesquisa recente. A segurança abrange questões como autenticação de usuários e criptografia dos dados que estão sendo transmitidos por um canal de comunicação sem fio.

  (ii) Meios de Comunicação sem Fio

  Os meios de comunicação sem fio irão prover novos e poderosos métodos para a disseminação de informações para um grande número de usuários. Novos métodos de acesso e novos paradigmas para a organização dos dados terão de ser desenvolvidos a fim de prover a disseminação e a recuperação das informações. Recentemente, o modo de disseminação de dados baseado em *broadcast* vem ganhando grande destaque e já está se transformando no principal modo de transferência de informações em ambientes de computação móvel e comunicação sem fio. Estes ambientes apresentam diversas vantagens. O servidor na rede fixa não fica sobrecarregado com pedidos de requisições. Além disso, os dados podem ser acessados concorrentemente por qualquer número de clientes sem nenhuma degradação de performance.

(iii) Gerenciamento Eficiente de Energia

Os computadores móveis dependem de baterias para funcionar. Atualmente, as baterias disponíveis só conseguem armazenar energia para algumas horas de uso e a evolução prevista não muda esse cenário. Portanto, a energia é um recurso limitado em computadores móveis e o seu consumo deve ser minimizado. Desta forma, novos protocolos e algoritmos para o acesso aos dados devem ser desenvolvidos tendo o consumo de energia como uma das métricas na composição dos custos de acesso aos dados.

- **Problemas Referentes a Escalabilidade**

A escalabilidade refere-se ao número de usuários do sistema, que em ambientes de computação móvel tende a ser extremamente grande. Algumas previsões apontam que em um futuro próximo teremos dezenas de milhões de dispositivos portáteis que poderão mover-se através de uma rede de comunicação mundial. Esta escala grandiosa afeta diversos outros problemas em computação móvel, como por exemplo, o gerenciamento da localização de usuários. O crescimento no número de usuários requer a utilização de células cada vez menores, este fato deve-se as limitações na largura de banda disponível para a comunicação entre clientes móveis e servidores. Consequentemente, teremos um aumento no número de *HandOffs*, o que complica ainda mais a tarefa de localizar um determinado usuário móvel. Além disso, a utilização maciça dos serviços resulta em grande heterogeneidade. Isto proporciona novos desafios, como por exemplo a necessidade de prover uma maneira uniforme de acesso aos serviços e bancos de dados.

- **Problemas Referentes aos Recursos dos Computadores Móveis**

  Os computadores móveis apresentam diversas limitações, como por exemplo, uma pequena capacidade de armazenamento de energia, dispositivos de interface bastante restritos (uma tela de pequenas dimensões, ausência de teclado e *mouse*, etc.), além de atualmente apresentarem restrições em sua capacidade de memória e processamento. A partir de uma perspectiva da tecnologia de bancos de dados as restrições dos recursos computacionais dos clientes móveis têm um impacto significativo nas seguintes áreas:

  (i) Otimização de Consultas

  Uma das principais questões quanto a forma com que as consultas serão executadas em um ambiente de computação móvel diz respeito aos otimizadores de consulta. Uma vez que os computadores móveis funcionam através da utilização de baterias e que estas apresentam uma capacidade bastante limitada de armazenamento de energia, novas técnicas de otimização de consulta terão de ser desenvolvidas com a finalidade de economizar os limitados recursos de energia dos computadores móveis. Assim, a seleção de planos de consulta deve basear-se no consumo de energia. Então, o critério usual de maximizar o número de consultas executadas por segundo deve ser substituído por minimizar o consumo de energia por consulta. Obviamente, isto deve ser feito sem degradar o tempo de resposta. Desta forma, os otimizadores de consulta deverão buscar novas métricas, onde os custos de transmissão via rede terão uma maior importância.

  (ii) Projeto de Interface

  Tradicionalmente as linguagens de consulta a bancos de dados, especialmente SQL, não podem ser facilmente utilizadas em dispositivos de interface baseados em caneta. O estado da arte no reconhecimento de

escrita é inadequado para a entrada de textos de grande tamanho, como uma consulta em SQL, por exemplo. Por outro lado, os computadores portáteis possuem uma tela de tamanho bastante limitado e em geral não apresentam teclado e nem mouse. Desta forma, uma interface gráfica na qual as consultas sejam expressas através da ação de apontar uma caneta sobre um elemento gráfico torna-se de extrema necessidade para o desenvolvimento de um ambiente de computação móvel.

(iii) Processamento de Transações

Os ambientes de computação móvel serão usados para executar sofisticadas aplicações que envolverão dados onde a consistência deverá ser mantida apesar das atualizações. Porém, devido a limitada largura de banda que os clientes possuem para comunicação com os servidores, às freqüentes desconexões dos clientes móveis e às suas restrições quanto a capacidade de armazenamento de energia, processamento e armazenamento de dados, torna-se de fundamental importância reexaminar os problemas referentes ao processamento de transações em sistemas de computação móvel. Os mecanismos tradicionais para controle de concorrência baseados em serializabilidade, como por exemplo, bloqueio, *timestamp* e consistência de *cache*, têm se mostrado impróprios para ambientes de computação móvel. Assim, novos mecanismos de controle de concorrência que sejam eficientes em ambientes de computação móvel e novos critérios de corretude devem ser concebidos.

2.4 Ambientes de Broadcast

A computação móvel está proporcionando o surgimento de novas e sofisticadas aplicações em banco de dados. Muitas delas apresentam como características um grande número de clientes móveis, um pequeno número de servidores e um banco de dados relativamente pequeno. Como exemplo podemos citar as aplicações de comércio eletrônico, tais como leilões e propostas eletrônicas, sistemas de controle de tráfego e automação industrial [32]. Enquanto tradicionalmente os dados são enviados dos servidores para os clientes sob demanda (*pull-based*), estas aplicações se beneficiam do modo de disseminação de dados baseado em difusão (*push-based*). Nele, o servidor repetidamente difunde (*broadcast*) os itens de dados para uma população de clientes sem que haja uma requisição específica por partes dos clientes. Cada período do *broadcast* é chamado de *broadcast cycle* ou *bcycle*, enquanto que o conteúdo do *broadcast* é chamado de *bcast*. Os clientes monitoram o canal de *broadcast* e retiram os itens de dados que necessitam [31]. Enquanto estes itens são enviados por *broadcast*, as transações que estão sendo executadas no servidor podem atualizar os valores dos itens de dados. Desta forma, distinguimos dois tipos de transações: as transações do servidor e as transações dos clientes, as quais chamaremos de transações móveis.

Os ambientes de *broadcast* apresentam diversas vantagens. O servidor na rede fixa não fica sobrecarregado com pedidos de requisições e não envia várias mensagens individuais que teriam que ser transmitidas em sistemas *pull-based*. Além disso, os dados podem ser acessados concorrentemente por qualquer número de clientes sem nenhuma degradação de performance. Entretanto, o acesso aos dados é estritamente seqüencial, ou seja, os clientes necessitam esperar que os dados de seu interesse apareçam no canal. Isto aumenta a latência de acesso à informação, a qual é proporcional à quantidade de informações a serem transmitidas em um *bcast*.

O conceito de disseminação de dados por *broadcast* não é novo. Trabalhos anteriores incluem o projeto *datacycle* e o sistema de informações da comunidade de

Boston (BCIS). No projeto *datacycle*, um banco de dados circula sobre uma rede de alta velocidade (140 Mbps). Os usuários consultam o banco de dados filtrando as informações relevantes através de um *hardware* especial (*special massively parallel transceiver*). Herman [18] discute um suporte transacional na arquitetura *datacycle*. Ele utiliza um mecanismo de controle de concorrência baseado em serializabilidade, os quais já mostramos serem muito dispendiosos, restritivos e desnecessários em tais ambientes. No sistema BCIS, notícias e informações são enviadas por *broadcast* , através de um canal de FM, para clientes com computadores pessoais equipados com receptores de rádio.

Em [2], os autores discutem a relação entre dados correntes (atuais) e problemas de performance quando alguns itens de dados enviados por *broadcast* são atualizados por processos executados no servidor. Entretanto, as atualizações não estão associadas a uma semântica transacional. As atualizações são realizadas somente por processos executados no servidor, enquanto que os processos nos clientes são somente de leitura.

A disseminação de dados baseada em *broadcast* está se transformando no principal modo de transferência de informações em ambientes de computação móvel e comunicação sem fio [23, 1]. Muitos destes sistemas têm sido propostos [33, 26] e já existem alguns produtos comerciais para disseminação de informações em redes de comunicação sem fio como *AirMedia* (*www.airmedia.com*) que envia regularmente notícias (manchetes e resumos) da CNN para usuários de computadores móveis, e o *DirectPC* (*www.directpc.com*) que coleta informações em servidores *Web*, envia para uma rede de satélites e, em seguida, difunde as mensagens para computadores pessoais em velocidades de até 400 kbps.

2.4.1 Arquitetura

Os componentes da arquitetura básica para um ambiente de disseminação de dados baseado em *broadcast* são apresentados na figura 2.2. O banco de dados consiste em uma coleção de itens de dados inter-relacionados. O servidor de banco de dados (SGBD) é responsável por armazenar e gerenciar as informações de um banco de dados. O servidor de *broadcast* periodicamente difunde os itens de dados. Os clientes representam computadores móveis nos quais estão sendo executadas aplicações que realizam operações de leitura e escrita sobre os itens de dados.

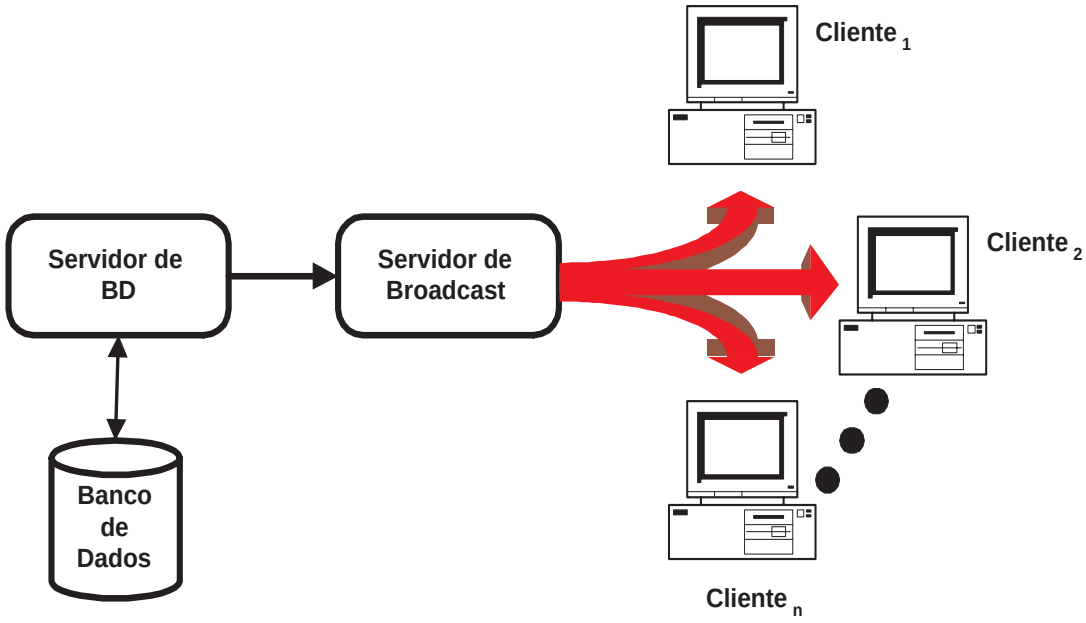

Figura 2.2: Disseminação de dados baseada em broadcast

Recentemente, o modo de disseminação de dados baseado em difusão tem recebido uma considerável atenção na área de computação móvel por causa do suporte físico para o *broadcast* existente nas redes celulares e via satélites.

As aplicações, em ambientes de *broadcast*, necessitam ler dados atuais e consistentes apesar das atualizações que podem ocorrer no servidor ou até mesmo nos clientes móveis. Portanto, o servidor de banco de dados (SGBD que reside no servidor) deve garantir que as seguintes restrições sejam satisfeitas:

(1) Consistência Mútua: Os dados mantidos no servidor e os dados lidos pelos clientes devem ser mutuamente consistentes.

(2) Dados Correntes: Os dados lidos pelos clientes devem ser atuais.

Contudo, garantir as restrições do tipo *consistência mútua* e *dados correntes* torna-se uma tarefa complexa quando consideramos as seguintes propriedades de ambientes de *broadcast*. Primeiramente, o canal de comunicação que um cliente possui para se comunicar com o servidor apresenta uma largura de banda limitada. Isso limita a capacidade dos clientes móveis no que diz respeito ao envio de solicitações aos servidores, como, por exemplo, solicitações de bloqueio. Em segundo lugar, os clientes móveis dependem de baterias para funcionar. Atualmente, estas baterias só conseguem armazenar energia para algumas horas de uso. Dessa forma, torna-se fundamental minimizar o tempo em que clientes gastam no seu modo ativo, realizando, por exemplo, funções relativas ao controle de concorrência. Adicionalmente, um cliente móvel pode passar longos períodos desconectado devido às limitações de energia da bateria e à própria mobilidade do cliente. Esses longos tempos de desconexão podem transformar transações executadas nos clientes móveis e que acessam dados dos servidores em transações de longa duração. Assim, garantir a consistência dos dados de maneira eficiente é um problema de pesquisa desafiador [32].

2.4.2 Estrutura do Broadcast

A partir do momento que um cliente necessita de um item de dado, o computador móvel deve ficar escutando o meio de comunicação até receber a informação desejada. Este é um processo que consome energia e só pode ser executado com o computador móvel no modo ativo. Além disso, é comum que os clientes móveis queiram acessar somente alguns itens específicos de dados transmitidos via difusão. Logo, é importante organizar os dados transmitidos via difusão para que os clientes

escutem o meio de comunicação apenas pelo período de tempo necessário para recuperar os dados de seu interesse. Dessa forma, será minimizado o consumo de energia, pois o cliente não terá que permanecer ativo durante todo o processo de difusão.

Assim, os clientes não necessitam escutar continuamente o canal de *broadcast*. Em vez disso, eles podem sintonizar apenas para ler determinados itens. Entretanto, para que esta seletividade seja possível os clientes devem ter, a priori, conhecimento sobre a estrutura do *broadcast*, a fim de determinar quando um item do seu interesse aparecerá no canal. Uma outra alternativa consiste em que o *broadcast* seja auto-descritivo, ou seja, de alguma forma as informações sobre a estrutura do *broadcast* é enviada junto com os dados. Neste caso, o cliente primeiramente lê as informações sobre o *broadcast* e em seguida usa estas informações nas leituras dos itens de seu interesse. Algumas técnicas para enviar informações de índices junto com os dados também podem ser utilizadas [25].

Uma estrutura para a organização do *broadcast* é mostrada na figura 2.3 [13]. Nesta estrutura, cada *bcast* é constituído por segmentos de índices e por blocos de dados. Os segmentos de índices descrevem a organização e a ordem das informações transmitidas. A utilização de índices é de extrema importância nas situações em que um cliente está interessado em parte dos dados transmitidos, permitindo um acesso seletivo aos itens desejados. Isto possibilita que os clientes economizem energia, ficando no modo *standby* a maior parte do tempo e entrando no estado ativo apenas para recuperar os itens de seu real interesse. Os blocos de dados são formados por unidades lógicas chamadas *buckets*. Em outras palavras, um bloco de dados é formado por vários *buckets*. O conceito de *bucket* é análogo ao de bloco em sistemas de arquivos. Cada *bucket* tem um cabeçalho que inclui informações úteis, como por exemplo, o número do ciclo corrente (atual) ou os identificadores dos itens atualizados durante o último ciclo. O conteúdo exato do cabeçalho pode variar, dependendo da implementação do *broadcast*. As informações no cabeçalho geralmente incluem a posição do *bucket* no *bcast*, um *offset* para o início do *bcast* e um *offset* para o início do próximo *bcast*. O *offset* para o início do próximo *bcast* pode ser usado pelo cliente

para determinar o início do próximo *bcast* quando o tamanho do *bcast* não é fixo. Cada *bucket* contém vários itens de dados. Os itens correspondem, por exemplo, a tuplas no banco de dados. Os usuários acessam os dados através do valor de um dos atributo do registro, a chave de busca.

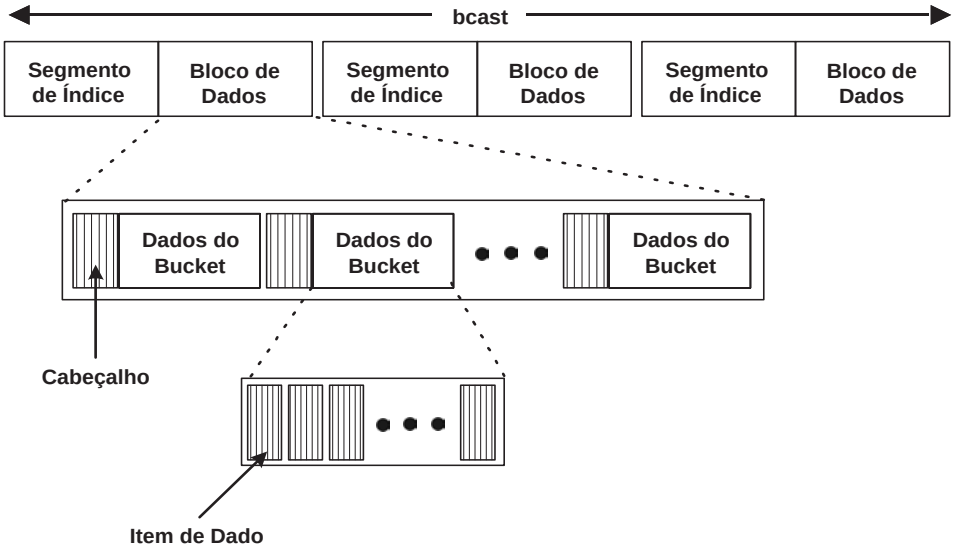

Figura 2.3: Uma Possível Estrutura para a Organização do Broadcast

Capítulo 3

Processamento de Transações em um Ambiente de Broadcast

3.1 Introdução

Desde os anos setenta o problema do controle de concorrência em ambientes multiusuário tem sido amplamente explorado. Em 1976, Eswaran et al. [14] propuseram um modelo que introduziu o conceito de transação. Este modelo, conhecido como modelo clássico para o controle de concorrência, é baseado no princípio de que a execução de uma transação em um ambiente multiusuário deve ser atômica, ou seja, sem interferência ou entrelaçamento com operações de outras transações.

O modelo clássico de transações tem se consolidado como uma solução padrão para o problema do controle de concorrência em sistemas de bancos de dados [5]. Este modelo tem se mostrado apropriado para processamento de transações em aplicações convencionais de bancos de dados nas quais as transações são de curta duração e o armazenamento dos dados é centralizado.

Contudo, a natureza e os requisitos do processamento de transações em ambientes de *broadcast* diferem das aplicações convencionais. Desta forma, os protocolos tradicionais para controle de concorrência têm se mostrado impróprios para ambi-

entes de *broadcast*. Esse fato é uma conseqüência das restrições na capacidade de comunicação dos clientes em ambientes com suporte a mobilidade.

O restante deste capítulo está organizado da seguinte forma: Na seção 3.2 o modelo transacional clássico é discutido e o critério de corretude convencional, denominado de serializabilidade, é descrito. Alguns dos protocolos para controle de concorrência baseados em serializabilidade são examinados. Vantagens e desvantagens deste modelo são apontadas e discutidas. Na seção 3.3, caracterizaremos o processamento de transações em ambientes de *broadcast*. Na seção **??**, resumiremos o capítulo e discutiremos os problemas adicionados no processamento de transações pelos ambientes de *broadcast*.

3.2 O Modelo Clássico de Processamento de Transações

3.2.1 O Problema da Concorrência em Sistemas de Bancos de Dados

Um banco de dados consiste de uma coleção de objetos que representam entidades do mundo real. Cada objeto do banco de dados tem um nome e um valor. Por exemplo, um objeto chamado conta-A com o valor de 5.000 pode representar uma conta de um determinado banco. O conjunto de valores de todos os objetos armazenados em um banco de dados num dado instante de tempo é chamado estado do banco de dados [5].

Portanto, o estado de um banco de dados representa uma visão instantânea do mundo, refletindo apenas seus aspectos estáticos. Entretanto, as mudanças que ocorrem em um mundo real devem ser refletidas no banco de dados. Essas mudanças são capturadas pelo conceito de transição de estados. Uma transição de estado representa o salto de um determinado estado para outro. Estas transições são realizadas pelos programas aplicativos que contêm operações de leitura e escrita

sobre os objetos do banco de dados.

O mundo real impõe certas restrições sobre as suas entidades. Por exemplo, uma conta bancária não pode ter saldo negativo. Logo, os bancos de dados devem capturar tais restrições, chamadas restrições de consistência. Se os objetos do banco de dados, em um dado instante, satisfazem todas as restrições de consistência dizemos que o estado do banco de dados é consistente.

Na prática, as aplicações são executadas concorrentemente, isto é, através do entrelaçamento de operações de diferentes programas, o que pode levar a mudanças inconsistentes no banco de dados. Consequentemente, a execução concorrente de programas aplicativos deve ser monitorada e controlada. Esta funcionalidade é chamada controle de concorrência.

Entretanto, do ponto de vista do controle de concorrência, nem todas as operações de um programa são relevantes, somente as operações sobre o banco de dados necessitam ser consideradas. Com base nesta observação, Eswaran entre outros [14] propôs um modelo que garante a correta execução de um conjunto de operações concorrentes sobre um banco de dados. Este modelo, chamado modelo transacional clássico, introduz o conceito de transação. Uma transação é uma abstração que representa uma seqüência de operações sobre o banco de dados (leitura e escrita) resultante da execução de um programa aplicativo.

A execução concorrente de um conjunto de transações é realizada através do entrelaçamento das operações que compõem as várias transações. Como já foi mencionado, alguns entrelaçamentos podem produzir estados inconsistentes. Logo, torna-se necessário definir quando a execução concorrente de transações está corretamente entrelaçada, ou seja, resulta em estados consistentes. Daqui em diante chamaremos uma execução corretamente entrelaçada de execução correta.

Um estado consistente do banco de dados representa uma visão coerente do mundo real. Com base nesta observação, podemos ter o seguinte critério de corretude para a execução concorrente de múltiplas transações: uma execução de transações concorrentes é correta se ela produz um estado consistente do banco de

dados. A primeira vista, este critério parece bastante razoável. Infelizmente, nas aplicações reais, nem todas as restrições de consistência são conhecidas. Isto torna inviável a identificação de um estado consistente do banco de dados. Em geral, a única informação disponível sobre as restrições de consistência é que uma aplicação preserva as restrições de consistência.

No modelo convencional para o controle de concorrência em bancos de dados, a noção de execução correta baseia-se na seguinte premissa: a execução de uma transação é correta, se a transação é executada por completo e de forma isolada das outras transações. Desta forma, se o estado do BD estava consistente antes do início da transação ele será consistente após o término da transação. Desde que a execução de uma transação simples seja correta, é fácil mostrar, usando indução sobre o número de transações, que qualquer execução serial de um conjunto de transações é correta. Consequentemente, se uma execução concorrente de um conjunto de transações é equivalente à execução serial das mesmas transações, então ela também é correta. Este critério de corretude é chamado de serializabilidade.

Desta forma, a serializabilidade proporciona a ilusão que a execução concorrente de múltiplas transações acontece de forma serial, uma após a outra. Por esta razão, dizemos que a serializabilidade fornece a ilusão que a execução de uma transação consiste em uma ação atômica. Uma transação é considerada uma unidade de execução atômica [5].

Nas próximas seções, definiremos mais precisamente os conceitos do modelo convencional. Noções de transações e *schedules* são discutidas e examinadas.

3.2.2 Transações

Como vimos, o modelo convencional para o controle de concorrência introduz a noção de transação. Transações são usadas para representar as operações sobre o banco de dados executadas pelas aplicações. Afim de preservar a semântica dos programas aplicativos, as transações também indicam a ordem na qual as operações

sobre o banco de dados devem ser executadas.

Ao iniciar sua execução dizemos que a transação encontra-se no estado *ativo*. Quando alcança sua última instrução, ela entra no estado *parcialmente executado*. Caso suas alterações possam ser armazenadas de forma permanente no banco de dados a transação passará ao estado *executado* ou *committed*. Caso contrário suas alterações devem ser desconsideradas ou desfeitas, e a transação entra no estado *abortado* ou *aborted*. Entretanto, uma transação pode ser abortada mesmo antes de encontar-se no estado *parcialmente executado* caso sua execução normal não possa mais prosseguir.

A notação $r_i(x)$ e $w_i(x)$ será utilizada para representar operações de leitura e escrita executadas por uma transação T_i sobre um objeto x. $OP(T_i)$ representa o conjunto de todas as operações executadas pela transação T_i. Além das operações de leitura e escrita uma transação T_i deverá conter após todas as operações de leitura e escrita uma operação *commit* (c_i) ou *abort* (a_i), mas não ambas. A operação *commit* indica que a transação encontra-se no estado *parcialmente executado* e que deseja passar para o estado *executado* ou *committed*. Já a operação *abort* indica que a transação deseja entrar no estado *aborted*.

Formalmente podemos definir uma transação T_i como uma ordem com relação de ordem $<_i$ onde:

(1) $OP(T_i) \subseteq \{r_i(x), w_i(x) \mid \text{x é um item de dado}\} \cup \{ a_i, c_i\}$

(2) $a_i \in OP(T_i)$ se e somente se $c_i \notin OP(T_i)$

(3) Seja $t \in \{c_i, a_i\}$, para qualquer operação p $\in OP(T_i)$, $p <_i t$ e

(4) Se $r_i(x), w_i(x) \in OP(T_i)$, então $r_i(x) <_i w_i(x)$ ou $w_i(x) <_i r_i(x)$

A relação de ordem $<_i$ representa a precedência entre duas operações de uma transação T_i. Por exemplo, $r_i(x) <_i w_i(x)$ indica que a operação $r_i(x)$ ocorreu antes da operação $w_i(x)$.

3.2.3 Execução Correta de Transações Concorrentes

O principal objetivo de um modelo de controle de concorrência é maximizar a concorrência entre as transações ao mesmo tempo em que garante a consistência do banco de dados. A execução concorrente de um conjunto de transações é realizada através do entrelaçamento das operações que compõem as várias transações. A execução entrelaçada de várias transações é modelada por uma estrutura denominada *schedule* ou história. Uma história ou *schedule* indica a ordem na qual as operações de um conjunto de transações são executadas umas com relação às outras, ou seja, ela representa uma execução concorrente. Desta forma, a ordem das operações em uma transação deve ser preservada. Isto é, se uma operação p_i precede q_i, na transação T_i ($p_i < q_i$), então a execução de p_i deve acontecer antes de q_i em qualquer *schedule* contendo T_i. A execução de um *schedule* representa o mapeamento de um estado do banco de dados para outro.

O conjunto das operações executadas no *schedule* S é denotado por OP(S). A notação p $<_S$ q indica que a operação p foi executada antes da operação q no *schedule* S. Por exemplo, $r_i(x) <_S r_j(y)$ representa o fato de que a execução da operação $r_i(x)$, pertencente à transação T_i, precede $r_j(y)$, pertencente a T_j, no *schedule* S, onde S é executado sobre um conjunto de transações T e $T_i, T_j \in T$.

Seja S um *schedule* sobre o conjunto $G \cup L$ de transações, onde G e L são conjuntos disjuntos de transações. A projeção do *schedule* S sobre o conjunto G é um *schedule* S' para o qual as seguintes condições devem ser satisfeitas:

(1) S' contem somente as operações das transações pertencentes ao conjunto G.

(2) $\forall\, p, q \in OP(S') \Rightarrow p, q \in OP(S)$

(3) $\forall\, p, q \in OP(S'),\ p <_{s'} q \Rightarrow p <_s q$

Um *schedule* S é serial se, para cada par de transações T_i, $T_j \in S$, todas as operações de T_i aparecem antes de todas as operações de T_j ou vice-versa. Assim,

podemos denotar um *schedule* serial sobre $T_1, T_2, T_3, ..., T_n$ como $T_{i1}, T_{i2}, T_{i3}, ..., T_{in}$ onde $i_1, i_2, ..., i_n$ é uma permutação de 1,2, ... , n. Visto que a execução de uma transação simples, de forma completa e isolada das demais transações, preserva a consistência do banco de dados, é fácil mostrar, usando indução sobre o número de transações, que qualquer execução serial (*schedule* serial) de um conjunto de transações é correta, ou seja, preserva a consistência do banco de dados.

No modelo clássico de transações uma execução concorrente de um conjunto de transações é correta se ela produz o mesmo resultado que alguma execução serial das mesmas transações. Portanto, se a execução de um *schedule* S sobre um conjunto de transações T, é equivalente a execução de algum schedule serial sobre T, então S é correto. Este critério de corretude é chamado de **serializabilidade**.

Um *schedule* S é denominado **serializável** se ele for equivalente a algum *schedule* serial S_s. Logo, um *schedule* serializável é correto. Desta forma, reduzimos o problema de identificar *schedules* corretos ao problema de definir equivalência entre *schedules*.

Existem três diferentes noções de equivalência que podem ser encontradas na literatura. A classe mais genérica de equivalência entre *schedules* é conhecida como equivalência por estado final. Dois *schedules* executados sobre o mesmo conjunto de transações T são ditos equivalentes por estado final se:

(i) possuem as mesmas operações pertencentes às transações em T, e

(ii) produzem o mesmo estado do banco de dados, caso sejam executados a partir do mesmo estado inicial.

A idéia da equivalência por estado final é identificar *schedules* que realizam a mesma transição de estados quando são executados a partir do mesmo estado inicial. Dois *schedules* equivalentes por estado final produzem o mesmo efeito sobre o banco de dados.

Uma outra maneira de definir equivalência entre *schedules* é através da noção de equivalência por visão. Dois *schedules* S e S' sobre o mesmo conjunto de transações

T são considerados equivalentes por visão se as seguintes condições forem satisfeitas:

(i) envolvem as mesmas operações pertencentes às transações em T,

(ii) para toda operação $r_i(x)$ de uma transação $T_i \in T$, o valor lido por r_i deve ser o mesmo nos dois *schedules*, e

(iii) se $w_i(x)$ é a última operação sobre um objeto x em S, então $w_i(x)$ é também a última operação de escrita sobre x em S'.

Neste caso, as operações de leitura de *schedules* equivalentes por visão têm a mesma visão do banco de dados. Como o resultado de uma operação escrita é uma função de todos os valores lidos anteriormente pela transação [5], a condição (iii) garante que a última operação de escrita nos dois *schedules* produzem o mesmo efeito sobre o banco de dados. Consequentemente, *schedules* equivalentes por visão sempre produzem o mesmo efeito sobre o banco de dados.

Comparando as duas classes de equivalência entre *schedules* descritas acima, podemos observar que a segunda condição da definição de equivalência por visão representa uma restrição sobre a classe de *schedules* equivalentes por estado final. Esta condição requer que *schedules* equivalentes por estado final tenham a mesma visão do banco de dados. Logo, podemos afirmar que a equivalência por visão é um caso especial de equivalência por estado final. Desta forma, a classe de *schedules* equivalentes por visão é um subconjunto da classe de *schedules* equivalentes por estado final, isto é, dois *schedules* equivalentes por visão também são equivalentes por estado final.

Antes de definir a terceira e mais restritiva classe de equivalência entre *schedules*, necessitamos introduzir o conceito de conflito entre duas operações. Duas operações de diferentes transações conflitam (ou estão em conflito) se e somente se elas acessam o mesmo objeto do banco de dados e pelo menos uma delas é uma operação de escrita.

De acordo com a definição de conflito, uma operação $r_i(x)$ sempre conflita com $w_j(x)$, e $w_i(x)$ conflita com $r_j(x)$ e $w_j(x)$, onde $i \neq j$. Por outro lado, uma operação

$w_i(x)$ não conflita com $w_j(y)$, pois operam sobre objetos diferentes (x e y), e $r_i(x)$ não conflita com $r_j(x)$, pois nenhuma das duas operações é de escrita.

Agora, iremos caracterizar a terceira noção de equivalência entre *schedules* chamada de equivalência por conflito. Dois *schedules* S e S' sobre um conjunto de transações T são ditos equivalentes por conflito se as seguintes condições são satisfeitas:

(i) envolvem as mesmas operações pertencentes às transações em T, e

(ii) para toda operação $p_i(x) \in OP(T_i)$ que conflita com uma operação $q_j \in OP(T_j)$, com $i \neq j$, se $p_i <_s q_j$, então $p_i <_{s'} q_j$.

Informalmente, dois *schedules* são equivalentes por conflito se as operações que conflitam estão na mesma ordem nos dois *schedules*. Do ponto de vista prático, a noção de equivalência por conflito é a mais importante das três, como será demonstrado mais adiante.

Desta forma, podemos distinguir três conceitos de *schedules* corretos:

(i) Serializabilidade por estado final (FSR): dizemos que um *schedule* S é serializável por estado final se ele é equivalente por estado final a algum *schedule* serial.

(ii) Serializabilidade por visão (VSR): um *schedule* S é dito serializável por visão caso seja equivalente por visão a algum *schedule* serial.

(iii) Serializabilidade por conflito (CSR): um *schedule* S será considerado serializável por conflito se for equivalente por conflito a algum *schedule* serial.

Podemos observar que todos os três conceitos de *schedules* corretos podem ser definidos no modelo convencional. Isto não significa que um determinado conceito é "mais correto" que os outros. De fato, cada noção de corretude representa um grau

de concorrência diferente, pois pode permitir um maior ou menor entrelaçamento entre as operações. Assim, quanto maior for o entrelaçamento permitido menos restritivo será o critério de corretude. Em [27], é mostrado que $FSR \supset VSR \supset CSR$. A serializabilidade por estado final é o critério menos restritivo e a serializabilidade por conflito é a mais restritiva.

A primeira vista, pode-nos parecer mais eficiente e racional adotar o critério menos restritivo, chamado serializabilidade por estado final. Isto porque este critério permite um maior entrelaçamento e, conseqüentemente, uma maior concorrência entre as transações. Infelizmente, a noção de equivalência por estado final é baseada nos conceitos de estado inicial e final de um banco de dados, os quais não são capturados pelo conceito de *schedule*. Este problema pode ser resolvido pela serializabilidade por visão. Entretanto, o problema de verificar se um *schedule* é serializável por visão não pode ser solucionado em tempo polinomial, isto é, este é um problema NP-Difícil [4]. Felizmente, podemos verificar se um *schedule* é serializável por conflito em tempo polinomial, isto é, existe um método eficiente para verificar se um *schedule* pertence à classe CSR. Por este motivo, quase todos os trabalhos práticos na área de controle de concorrência implementam a serializabilidade por conflito como critério de corretude.

A estratégia utilizada para determinar se um *schedule* S é ou não serializável por conflito consiste em verificar se um grafo direcionado, chamado grafo de serialização, possui ou não ciclos. A seguir apresentamos uma definição formal para o grafo de serialização.

Seja S um *schedule* global sobre um conjunto de transações $T = \{T_1, T_2, ..., T_n\}$. O grafo de serialização para S, representado por GS(S), é definido como o grafo direcionado GS(S) = (N,A) no qual cada nó em N corresponde a uma transação em T. O conjunto A contém as arestas na forma $T_i \rightarrow T_j$, se e somente se $T_i, T_j \in N$ e existem duas operações $p \in OP(T_i)$, $q \in OP(T_j)$, onde p conflita com q e $p <_S q$. A noção de grafo de serialização proporciona um método eficiente para identificar *schedules* serializáveis por conflito. Um *schedule* global S é serializável por conflito

se e somente se o grafo de serialização para S é acíclico (complexidade ($O(n^2)$) [4].

Até agora, temos analisado a corretude de *schedules* completos, os quais são representados através da execução completa das transações. Entretanto, em um ambiente dinâmico é importante considerar a possibilidade da execução incompleta de um determinado *schedule*. A execução incompleta de um *schedule* é representada através do conceito de prefixo. Assim, L é um prefixo do schedule S se as seguintes condições são satisfeitas:

(i) $OP(L) \subseteq OP(S)$

(ii) $\forall p, q \in OP(L), p <_L q \Leftrightarrow p <_S q$

(ii) $\forall q \in OP(L), \forall p \in OP(S), p <_S q \Rightarrow p \in OP(L)$

Em ambientes dinâmicos é fundamental que o critério de corretude seja "*prefix-closed*", isto é, se um schedule S é correto então qualquer prefixo de S também é correto. Em [4] encontramos o seguinte teorema : A serializabilidade por conflito é "*prefix-closed*".

Esta característica permite otimizar o método para identificar *schedules* serializáveis por conflito, pois se um prefixo L de um *schedule* S não for serializável por conflito podemos afirmar que S também não será serializável por conflito.

Torna-se importante destacar que nem todos os critérios de corretude são "*prefix-closed*". Por exemplo, a serializabilidade por estado final e a serializabilidade por visão não são "*prefix-closed*".

3.2.4 Confiabilidade de Schedules

Até agora, temos discutido o modelo clássico para a execução de transações concorrentes sem considerar a presença de falhas. Na prática, as transações são executadas em ambientes onde podem ocorrer falhas. Logo, um mecanismo eficiente para o processamento de transações deve suportar a existência de falhas.

Outra contribuição importante do modelo clássico de processamento de transações é o conceito de schedule confiável. Em [16] nós podemos encontrar uma discussão inicial sobre este assunto. Neste trabalho a confiabilidade esta relacionada ao grau de consistência. A seguir definiremos formalmente a noção de *schedule* confiável.

Dizemos que um *schedule* é confiável se as seguintes condições são satisfeitas:

(i) O *abort* de uma determinada transação T_i não afeta a semântica das transações que já executaram sua operação de *commit*, e

(ii) Quando uma determinada transação T_i aborta, os valores dos objetos atualizados anteriormente por T_i devem ser restaurados como se esta transação nunca tivesse sido executada. Os efeitos de uma transação abortada devem ser completamente desfeitos.

Torna-se importante destacar que os conceitos de serializabilidade e confiabilidade de *schedules* são propriedades ortogonais. Existem *schedules* serializáveis que não são confiáveis e vice-versa. Contudo, do ponto de vista prático os *schedules* devem ser corretos (serializáveis) e confiáveis.

A seguir iremos caracterizar melhor os *schedules* confiáveis. De fato, existem três diferentes níveis de confiabilidade de *schedules*. O nível mais genérico é chamado de recuperação de *schedules*. Um *schedule* S é dito recuperável se a condição abaixo é satisfeita:

Se $w_i(x) <_S r_j(x)$, então a operação *commit* da transação T_i precede a operação de *commit* da transação T_j, isto é, $c_i <_S c_j$.

Schedules recuperáveis garantem que, uma vez que uma transação T_i executa sua operação de *commit*, seus efeitos nunca serão desfeitos. Entretanto, mesmo em *schedules* recuperáveis, o *abort* de uma transação T_i pode provocar efeitos colaterais indesejáveis. Por exemplo, se $w_i(x) <_S r_j(x)$, e a transação T_i aborta após a execução da operação $r_j(x)$ e antes de T_j executar sua operação de *commit*, o valor lido

por $r_j(x)$ não é válido. Isto implica que a transação T_j também deve ser abortada. Este fenômeno é chamado de *abort* em cascata. A fim de evitar a ocorrência destes fenômenos nós necessitamos de um nível de confiabilidade mais forte. Um *schedule* é dito que evita *aborts* em cascata se:

Sempre que $w_i(x) <_S r_j(x)$, então $c_i <_S r_j(x)$ ou $a_i <_S r_j(x)$

Algumas vezes, esta propriedade ainda não é suficientemente forte para garantir um nível de confiabilidade desejável. O *schedule* S mostrado a seguir ilustra este fato.

$S = w_i(x)\, w_j(x)\, c_j\, a_i$

Como T_i foi abortada, o valor do objeto x deve ser restaurado para o valor anterior à execução da transação T_i. Infelizmente, isto não é possível, pois a transação T_j já executou sua operação de *commit* e sobrescreveu o valor escrito por T_i. Este problema pode ser evitado fazendo com que a operação $w_j(x)$ espere até que a transação T_i execute sua operação de *commit* ou de *abort*, só então a operação $w_i(x)$ poderá ser executada. Esta condição é garantida pelo nível mais forte de confiabilidade, que é chamado "*strictness*". Um *schedule* é chamado estrito ("*strict*") se:

Sempre que $w_i(x) <_S p_j$ então $c_i <_S p_j$ ou $a_i <_S p_j$, onde $p_j \in \{r(x), w(x)\}$

Observe que um schedule estrito ("*strict*") evita *aborts* em cascata e é recuperável. Isto deve-se ao fato de que a classe de *schedules* estritos ("*strict*") é uma subclasse dos *schedules* que evitam *aborts* em cascata, o qual por sua vez é uma subclasse dos *schedules* recuperáveis. Uma prova formal deste fato pode ser encontrada em [17] ou em [4].

3.2.5 Protocolos de Controle de Concorrência

Os modelos de processamento de transações são implementados através dos protocolos de controle de concorrência. Estes protocolos são responsáveis por produzir *schedules* em conformidade com o critério de corretude adotado. Pelo termo "produzir schedules", entendemos que um protocolo de controle de concorrência recebe como entrada as operações pertencentes a um determinado conjunto de transações e produz como saída uma seqüência corretamente entrelaçada destas operações. Por exemplo, o protocolo 2PL (*Two Phase Locking*) é um protocolo de controle de concorrência que implementa o modelo clássico de processamento de transações, logo, utiliza serializabilidade como critério de corretude. Assim, o protocolo 2PL produz schedules serializáveis.

Os protocolos de controle de concorrência constituem o núcleo dos escalonadores (*schedulers*). Estes protocolos podem ser classificados como agressivos ou conservativos. Um protocolo agressivo tenta escalonar as operações imediatamente. Desta forma, um protocolo agressivo evita adiar a execução das operações de uma transação. Entretanto, esta estratégia pode levar a situações onde não é possível produzir uma execução correta de todas as transações ativas. Neste caso, um protocolo agressivo rejeita operações de uma ou mais transações, levando estas transações ao estado *aborted*. Assim, os protocolos agressivos podem gerar índices de *aborts* inaceitáveis.

Adicionalmente, os protocolos agressivos podem ser classificados em duas sub-classes: pessimistas e otimistas. Um *scheduler* que implementa um protocolo pessimista deve decidir se aceita ou rejeita uma operação tão logo ele receba esta operação.

Por outro lado, um *scheduler* que utiliza um protocolo otimista escalona as operações que recebe de forma imediata. Após um determinado período de tempo, o *scheduler* verifica se o *schedule* que está sendo gerado é correto ou não. Caso o *schedule* seja correto, o *scheduler* continua escalonando operações. Caso contrário,

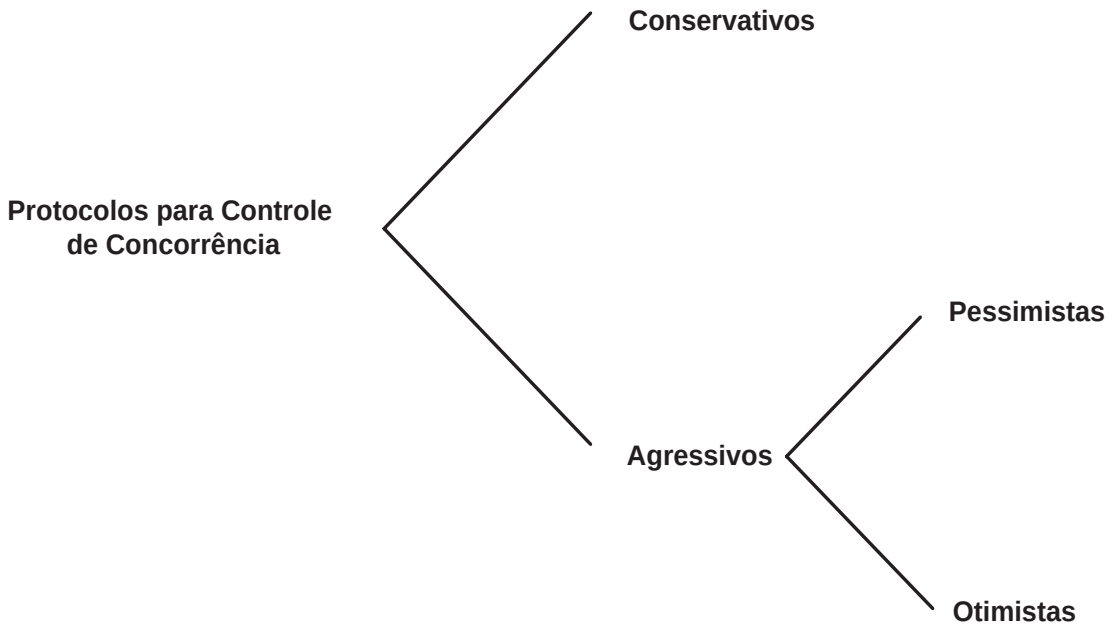

Figura 3.1: Classificação dos Protocolos de Controle de Concorrência.

o *scheduler* deve abortar uma ou mais transações. Um exemplo clássico para esta classe de protocolos são os certificadores [19].

Por sua vez, um protocolo conservativo tende a adiar a execução de determinadas operações com a finalidade de escaloná-las corretamente. Desta forma, estes protocolos, apesar de não induzir o *abort* de transações, podem levar as aplicações a tempos de espera inaceitáveis, principalmente se estas forem transações de longa duração [5]. A figura 3.1 mostra uma visão geral desta classificação [4].

A seguir, descreveremos três mecanismos utilizados para controle de concorrência. O primeiro, intitulado protocolo baseado em bloqueio, utiliza a idéia de *lock* (bloqueio) e pode ser classificado como um protocolo conservativo. O segundo mecanismo, baseia-se no conceito de *timestamp* (marcador de tempo) e é denominado ordenação por *timestamp* (*Timestamp Ordering - TO*). Este protocolo pode ser caracterizado como agressivo. Finalmente, discutiremos os mecanismos de consistência de cache.

Nos protocolos baseados em bloqueio, uma transação T_i somente pode acessar um item de dado caso tenha anteriormente obtido um bloqueio sobre este item.

Assim, a fim de acessar um determinado item, a transação T_i precisa primeiramente bloqueá-lo. Caso o item de dado esteja bloqueado por uma outra transação, então T_i deverá esperar até que o bloqueio seja liberado. Desta forma, o *scheduler* garante que somente uma transação pode acessar o item de dado por vez. Logo, os protocolos baseados em bloqueio produzem *schedules* serializáveis [4].

O mecanismo de ordenação por *timestamp* associa a cada transação um determinado marcador de tempo ($ts(T_i)$). Esta tarefa é realizada antes da execução da primeira operação de cada transação. O marcador de tempo indica a ordem em que as transações foram iniciadas. Assim, se a transação T_i iniciou sua execução antes da transação T_j teremos que $ts(T_i) < ts(T_j)$. O *scheduler* ordena as operações das transações observando a seguinte regra: Se $P_i(x)$ e $q_j(x)$ são operações que conflitam, então o *scheduler* irá processar $p_i(x)$ antes de $q_j(x)$ se e somente se $ts(T_i) < ts(T_j)$. Assim, o protocolo de ordenação por *timestamp* produz *schedules* serializáveis.

Com a finalidade de melhorar o desempenho no acesso às informações foram propostos os mecanismos de *caching*. Nesta abordagem, os clientes podem manter, em uma memória *cache*, cópias dos dados mais prováveis de serem acessados, diminuindo assim o número de requisições enviadas ao servidor, bem como o tempo que o cliente espera para acessar os itens de dados. Entretanto, a introdução de *caches* nos clientes proporcionou a forte necessidade de técnicas para garantir que semântica transacional não seja violada como resultado da criação e destruição das cópias de dados. Tais técnicas são chamadas de algoritmos de consistência de *cache*.

Nos tradicionais SGBD's cliente/servidor, é relativamente fácil garantir a consistência dos dados armazenados em *cache* utilizando uma combinação de bloqueios e mensagens de invalidação, uma vez que a localização e a conexão dos clientes não mudam. Nestes ambientes, os algoritmos para consistência de *cache* podem ser divididos em duas categorias:

- **Avoidance-based Approach:** Servidores enviam mensagens de invalidação diretamente aos clientes que têm dados em *cache* para que estes sejam atualizados.

- **Detection-based Approach:** Clientes enviam consultas para os servidores a fim de validar a consistência dos dados armazenados em *cache*.

3.3 O Modelo de Transações em Ambientes de Computação Móvel

Nesta seção, iremos definir um modelo básico para o processamento de transações em ambientes de computação móvel. Inicialmente, nós precisamos definir um ambiente de computação móvel do ponto de vista do processamento de transações. Um ambiente de computação móvel consiste de uma infra-estrutura fixa (rede de computadores fixos) na qual reside um sistema de bancos de dados (SGBD), adicionada de um conjunto de usuários, de posse de computadores portáteis, que podem acessar (consultar e atualizar) as informações armazenadas no SGBD independentemente de sua mobilidade, ou seja, estes usuários podem acessar o banco de dados ao mesmo tempo em que se movimentam livremente. Neste ambiente, tanto os usuários fixos quanto os usuários móveis interagem com o SGBD através de transações. Assim, duas classes de transações são suportadas em um ambiente de computação móvel:

- **Transações Móveis.** São as transações executadas nos computadores móveis, ou seja, estas transações resultam das aplicações executadas nos clientes móveis.

- **Transações do Servidor.** São as transações executadas nos computadores da rede fixa, ou seja, estas transações resultam das aplicações executadas nos clientes fixos.

Definição 1 (Sistema de Computação Móvel) *Um sistema de computação móvel consiste de:*

(i) Um conjunto $CF = \{CF_1, CF_2, ..., CF_m\}$ *de computadores fixos.*

(ii) Um conjunto $CM = \{CM_1, CM_2, ..., CM_n\}$ *de computadores móveis.*

(iii) Um SGBD localizado em um ou mais computadores fixos.

(iv) Um conjunto $M = \{M_1, M_2, ..., M_t\}$ *de transações móveis onde cada* M_k *representa o conjunto de transações móveis executadas no computador móvel* CM_k*.*

(v) Um conjunto $F = \{F_1, F_2, ..., F_l\}$ *de transações do servidor onde cada* F_k *representa o conjunto de transações do servidor executadas no computador fixo* CF_k*.*

Definição 2 (Schedule Local) *Um schedule local* S_k *modela o entrelaçamento das operações pertencentes às transações executadas a partir de um computador móvel* CM_k*. Neste caso, o schedule* S_k *é definido sobre o conjunto de transações* M_k*.*

Definição 3 (Schedule Global) *Um schedule global SG modela o entrelaçamento das operações pertencentes às transações executadas tanto a partir dos computadores móveis quanto a partir dos computadores fixos. Neste caso, o schedule global SG é definido sobre o conjunto de transações* $T = M \cup F$*.*

Para ilustrar as definições acima, vamos tomar como exemplo um sistema de computação móvel formado por um computador fixo, CF_1, e dois computadores móveis, CM_1 e CM_2. Além de um sistema de banco de dados (SBD_1) localizado no computador CF_1. Considere o seguinte conjunto de transações, as quais possuem operações de leitura e escrita sobre os itens do banco de dados SBD_1.

$T_1 : r_1(a)\, r_1(d)$

$T_2 : r_2(b)\, r_2(c)$

$T_3 : r_3(a)\, r_3(c)$

$T_4 : r_4(c)\, r_4(e)$

$T_5 : w_5(b)\, w_5(e)$

As transações T_1 e T_2 são executadas no cliente móvel CM_1, enquanto T_3 e T_4 no cliente móvel CM_2. Por outro lado, a transação T_5 é executada no computador fixo CF_1.

A figura 3.2 mostra dois possíveis *schedules* locais, S_1 e S_2, para as operações pertencentes às transações executadas a partir dos computadores móveis CM_1 e CM_2, respectivamente. Já a figura 3.3 ilustra um possível *schedule* global SG.

$$S_1: r_1(a)\ r_2(b)\ r_2(c)\ r_1(d)$$

$$S_2: r_3(a)\ r_4(c)\ r_3(c)\ r_4(e)$$

Figura 3.2: Schedules Locais S_1 e S_2

$$SG: r_1(a)\ r_3(a)\ r_2(b)\ w_5(b)\ r_4(c)$$
$$r_3(c)\ r_2(c)\ w_5(e)\ r_1(d)\ \ r_4(e)$$

Figura 3.3: Schedule Global SG

3.4 Controle de Concorrência em Ambientes de Broadcast

O modelo clássico de processamento de transações tem se mostrado apropriado para aplicações convencionais de bancos de dados nas quais as transações são tipicamente de curta duração. Entretanto, os mecanismos de controle de concorrência baseados em serializabilidade requerem (a) uma excessiva comunicação entre clientes e servidores, para obter bloqueios, por exemplo, e (b) rejeita certas execuções corretas. A primeira alternativa é cara para ambientes de *broadcast* por causa da limitada largura de banda que os clientes possuem para se comunicar com os servidores. A segunda alternativa leva a *aborts* desnecessários, o que resulta numa inaceitável perda de trabalho pois as transações em ambientes de *broadcast* são de longa duração. A seguir, mostraremos que as três principais técnicas utilizadas, em ambientes distribuídos e cliente/servidor, pela maioria dos mecanismos para controle de concorrência baseados em serializabilidade são impróprias para ambientes de *broadcast*:

Bloqueio: Os clientes precisam obter bloqueios para cada item a ser lido. Isto sobrecarrega o servidor com requisições de bloqueio, pois o número de clientes tende a ser grande. Além disso, os clientes precisam contactar o servidor a cada leitura, porém, o uplink possui uma limitada largura de banda.

Mecanismos de Consistência de *Cache*: O servidor precisa conhecer os itens armazenados pelos clientes, pois mudanças nos itens devem ser propagadas/invalidadas para os clientes. Neste caso, o servidor terá que manter a lista dos dados armazenados em um grande número de clientes. Além disso os clientes devem informar ao servidor toda vez que um item é lido. Manter versões antigas nos clientes comprometem a atualidade dos itens lidos.

Timestamp: requer comunicação entre clientes e servidores a cada leitura.

Portanto, as aplicações em ambientes de *broadcast* necessitam de um novo mecanismo de controle de concorrência baseado em serializabilidade que seja eficiente nestes ambientes ou de novos critérios de corretude.

Capítulo 4

Trabalhos Correlatos

4.1 Introdução

Com o objetivo de solucionar eficientemente a questão do controle de concorrência em ambientes de *broadcast*, várias abordagens têm sido apresentadas. Neste capítulo descreveremos as principais propostas existentes para a solução deste complexo problema.

Os principais mecanismos para o controle de concorrência em ambientes de *broadcast* consideram que as transações nos clientes são somente de leitura, ou seja, todas as atualizações são executadas no servidor e disseminadas a partir deste para a população de clientes. Esta consideração é razoável, pois a maioria das transações em sistemas de difusão são somente de leitura. Além disso, mesmo que as transações de atualização fossem permitidas nos clientes, seria mais eficiente processá-las com algoritmos especiais.

4.2 Considerando Transações Somente de Leitura

Em um ambiente de computação móvel com disseminação de dados por *broadcast* o servidor periodicamente difunde o conteúdo do banco de dados para a população de clientes móveis. Um banco de dados consiste de um conjunto finito de itens de

dados. Enquanto os itens de dados são enviados por *broadcast*, as transações que estão sendo executadas no servidor podem atualizar os valores dos itens que foram enviados em *broadcast*. Vamos assumir que os valores dos itens de dados enviados em *broadcast* durante cada ciclo correspondem ao estado do banco de dados até o início do *broadcast*, ou seja, correspondem aos valores produzidos por todas as transações que executaram operações de *commit* até o início do ciclo. Iremos denominar essas transações de *"committed"*. Desta forma, garantimos que o conteúdo do *broadcast* a cada ciclo é consistente.

Portanto, uma transação somente de leitura que lê todos os seus dados em um único ciclo pode ser executada sem problemas, pois todos os valores lidos estão consistentes.

Um critério de corretude que pode ser usado no caso de transações somente leitura é que cada transação deve ler dados que correspondem a um mesmo estado do banco de dados, ou seja, a um único *bcast*.

Visto que o conjunto de itens a serem lidos por uma transação não é conhecido a priori e o acesso aos dados é seqüencial, as transações podem ler itens de dados de diferentes *bcasts*, ou seja, valores de diferentes estados do banco de dados. Esta é a principal desvantagem desta abordagem. Consideremos, por exemplo, a transação T que corresponde ao seguinte programa:

if $a > 0$ then read b else read c

Suponha que os itens "b" e "c" precedem o item "a" no *broadcast*. Logo, a transação cliente tem que primeiramente ler o item "a" e esperar o próximo ciclo para ler os itens "b" e "c". Caso as transações clientes leiam itens de dados de diferentes ciclos, não é garantido que os valores lidos sejam consistentes.

Esta abordagem é utilizada como critério de corretude por várias das propostas discutidas a seguir.

4.3 Relatório de Invalidação

Neste mecanismo, proposto em [31], cada *bcast* é precedido por um relatório de invalidação na forma de uma lista que inclui todos os itens de dados que foram atualizados no servidor, por transações *committed*, durante o ciclo de *broadcast* anterior. Para cada transação somente leitura ativa R, o cliente guarda um conjunto *Read_Set(R)* de todos os itens de dados lidos por R. Vamos definir *readset* de uma transacao T, denotado por *Read_Set(T)*, o conjunto dos itens lidos pela transação T. Em particular, *Read_Set(T)* é um conjunto de pares ordenados na forma (item lido, valor lido). O *readset* de cada transação somente leitura deve ser um subconjunto de um estado consistente do banco de dados. No início de cada *bcast*, o cliente sintoniza e lê o relatório de invalidação. A transação somente leitura R é abortada se um item x $\in$ *Read_Set(R)* foi atualizado, isto é, se x aparece no relatório de invalidação.

[31] apresenta o seguinte teorema: O relatório de invalidação produz transações somente leitura corretas.

Uma transação somente leitura R lê os valores mais recentes, isto é, os valores produzidos por todas as transações *committed* até o início do ciclo onde R executa o *commit*. Desta forma, todos os valores lidos pela transação R correspondem a um único *bcast*, ou seja, a um único estado do banco de dados.

Esta abordagem requer que cada cliente leia todos os relatórios de invalidação, que são enviados antes de cada ciclo. Desta forma, o cliente não pode desconectar-se por muito tempo. Para tentar resolver este problema, os relatórios de invalidação podem ser enviados em uma freqüência menor. Neste caso, antes que uma transação somente de leitura execute o *commit* ela deve ler o próximo relatório de invalidação que aparecer no canal de *broadcast*.

Além de ser uma abordagem bastante simples, o cliente não necessita contactar o servidor e as informações de controle transmitidas são de tamanho relativamente pequeno. Por outro lado, ela descarta várias histórias serializáveis por conflito. Além disso, o cliente não pode desconectar-se por muito tempo, pois precisa ler todos os

relatórios de invalidação, independentemente de sua freqüência, o que torna esta proposta inviável.

4.4 Invalidação Baseada em Versão

Neste método, descrito em [31], um marcador de tempo ou um número de versão é enviado junto com cada item de dado. Este número de versão corresponde ao número do ciclo onde o item foi atualizado pela última vez. Para cada transação somente de leitura ativa R, o cliente guarda um conjunto *Read_Set(R)* de pares ordenados formados pelos itens de dados lidos até o momento e seus respectivos números de versão. Por exemplo, o par ordenado (x, C_i) indica que o item de dado x foi atualizado pela última vez durante o ciclo C_i. Seja C_0 o número do ciclo no qual uma transação R executa sua primeira operação de leitura. A cada nova leitura, o seguinte teste deve ser realizado: se um item $x \in Read\_Set(R)$ foi atualizado, isto é, se $C_x > C_0$ então a transação R deve ser abortada.

O seguinte teorema é demonstrado em [31]: A invalidação baseada em versão produz transações somente leitura corretas.

Uma transação somente leitura R lê os valores produzidos por todas as transações *committed* até a execução da primeira operação de leitura pela transação R . Desta forma, todos os valores lidos pela transação R correspondem a um único *bcast*, ou seja, a um único estado do banco de dados.

Como no relatório de invalidação, o cliente não necessita contactar o servidor. Contudo, o cliente não necessita ler todos o *bcasts*, podendo desconectar-se por qualquer período de tempo. Como desvantagens desta proposta podemos citar: descarta várias histórias serializáveis por conflito; além disso, a cada nova leitura deve-se ler também o número de versão de todos os itens lidos anteriormente.

4.5 Múltiplas Versões

A idéia básica desta proposta, apresentada em [31], consiste em manter temporariamente versões anteriores dos itens de dados a fim de diminuir o número de *aborts* nas transações somente leitura. Em uma abordagem particular, *S-Multiversion*, para cada item de dado são armazenados os seus S valores anteriores, ou seja, os valores do item durante os S *bcycles* anteriores, onde S é o máximo *span* entre todas as transações somente leitura. Vamos definir *span* de uma transação T, *span(T)*, como sendo o número máximo de ciclos de *broadcast* onde a transação T lê itens de dados. Então, para uma transação T qualquer, se *span*(T) = 1, T é correta. Para implementar este esquema, o servidor, além de difundir o último valor *committed* para cada item de dado, mantém e difunde múltiplas versões para cada item de dado. Pelo menos um valor, a versão corrente, é difundido para cada item de dado. A cada $bcycle_k$, o servidor descarta as k-S versões do *bcast*.

Seja C_0 o número do ciclo de *broadcast* durante o qual a transação cliente R executada sua primeira operação de leitura. Durante C_0, a transação R lê o valor mais atual de cada item de dado. Nas leituras posteriores, R lê o valor com o maior número de versão C_n, tal que $C_n \leq C_0$.

Em [31] encontramos a demonstração do seguinte teorema: *S-Multiversion* produz transações somente leitura corretas.

Desta forma, uma transação somente leitura R lê os valores produzidos por todas as transações *committed* até o início do *bcycle* C_0. Assim, todos os valores lidos pela transação R correspondem a um único *bcast*, ou seja, a um único estado do banco de dados.

Basicamente, existem três meios para o armazenamento das versões anteriores. Um meio de armazenamento em potencial é o ar, neste caso, as versões anteriores são difundidas junto com os valores atuais dos itens de dados. A segunda possibilidade é manter as versões anteriores em um *cache* no cliente. Neste caso, é possível termos um coletor de lixo para as versões antigas, desde que existam informações sobre as

transações ativas no cliente. Por último, podemos manter parte do banco de dados nos clientes na forma de visões materializadas.

O uso de múltiplas versões proporciona um aumento no grau de concorrência. Além disso o cliente não necessita contactar o servidor para executar as operações de leitura. Por outro lado, esta abordagem descarta várias histórias serializáveis por conflito, provoca um *overhead* na leitura e manutenção das múltiplas versões.

Caso as múltiplas versões sejam armazenadas em *caches* nos clientes, relatórios de invalidação são enviados, a cada *bcycle*, para garantir a leitura de dados correntes. Desta forma, o cliente necessita escutar o canal de *broadcast* a cada *bcycle*. Para tentar resolver este problema, os relatórios de invalidação podem ser enviados em uma freqüência menor. Neste caso, antes que uma transação somente de leitura execute o *commit* ela deve ler o próximo relatório de invalidação que aparecer no canal de *broadcast*.

4.6 Teste do Grafo de Serialização (SGT)

Tanto os métodos baseados em invalidação quanto o de múltiplas versões garantem que as transações somente leitura lêem valores consistentes, isto é, valores produzidos por uma execução serializável, pois os valores lidos por estas transações correspondem a um único *bcast*, ou seja, a um único estado do banco de dados. No caso do método do relatório de invalidação, os valores lidos correspondem ao *bcast* do final da transação, enquanto no método de múltiplas versões os valores lidos correspondem ao *bcast* do início da transação. Entretanto, é suficiente que as transações leiam valores que correspondam a qualquer estado consistente, não necessariamente um estado que tenha sido enviado por *broadcast*. Neste fato baseia-se o método do teste do grafo de serialização, proposto em [31].

O grafo de serialização de uma história H, denotado por SG(H), é um grafo direcionado onde os nós são as transações *committed* em H e as arestas são da forma $T_i \rightarrow T_j$ $(i \neq j)$ tal que uma operação de T_i precede e conflita com uma operação de

T_j em H. De acordo com o teorema da serializabilidade, uma história H é serializável se e somente se SG(H) é acíclico.

O método SGT trabalha da seguinte forma. Cada cliente mantém localmente uma cópia do grafo de serialização. O grafo de serialização no servidor inclui somente as transações *committed* no servidor, enquanto, o grafo mantido nos clientes incluei as transações *committed* no servidor e as transações somente leitura ativas neste cliente. A cada ciclo, o servidor difunde qualquer atualização no grafo de serialização. Após receber as atualizações, o cliente integra estas atualizações em sua cópia local do grafo. Uma operação de leitura no cliente é executada somente se ela não cria um ciclo no grafo de serialização local. O grafo de serialização no servidor não é necessariamente usado para o controle de concorrência no servidor, um método mais prático, como o de bloqueio em duas fases pode ser utilizado.

4.6.1 Implementando o método SGT

Neste método, o servidor difunde, no início de cada *bcast*, as seguintes informações de controle:

- As alterações no grafo de serialização

  Em particular, o servidor difunde para cada transação T_i que foi *committed* durante o ciclo anterior, uma lista das transações com a qual ela conflita, isto é, as transações com a qual T_i está diretamente conectada por uma aresta.

- Um relatório de invalidação

  Este relatório inclui todos os itens de dados escritos durante o *bcycle* anterior juntamente com o identificador da transação que primeiramente escreveu cada item.

Estas informações de controle são utilizadas pelos clientes na montagem e atualização do grafo de serialização. Assim, cada cliente sintoniza no início do *broadcast* para obter as informações de controle. Em seguida, o cliente atualiza

a sua cópia local do grafo de serialização da seguinte forma: No início de cada $bcycle_{i+1}$, o cliente adiciona arestas para todas as transações somente leitura ativas da seguinte forma: Seja R uma transação ativa e $RS_i(R)$ o conjunto dos itens lidos por R até o $bcycle_i$. Para cada item x presente no relatório de invalidação tal que $x \in RS(R)$, o cliente adiciona uma aresta $R \rightarrow Tf$, onde Tf é a primeira transação que escreveu em x durante o *bcycle* i. Desta forma R conflita com todas as transações que escreveram em x durante o $bcycle_i$.

O teorema a seguir é apresentado em [31]: O método SGT produz transações somente leitura corretas.

Nesta abordagem, o cliente não necessita contactar o servidor para executar as operações de leitura. Além disso, algumas histórias que não seriam aceitas na serialização tradicional são consideradas corretas. Por outro lado, o cliente não pode desconectar-se por muito tempo, pois precisa ler as informações de controle enviadas no início de cada *bcycle*. Para ler estas informações, o cliente necessita escutar o canal de *broadcast* durante um período de tempo maior, o que se torna problemático devido às limitações no fornecimento de energia pelas baterias dos computadores portáteis. O cliente também precisa manter e testar o grafo de serialização. Entretanto, os computadores portáteis têm, em geral, uma pequena capacidade de memória.

4.7 Consistência de Atualização

Esta abordagem, proposta em [32], utiliza como critério de corretude uma adaptação do critério proposto no contexto do controle de concorrência em sistemas de múltiplas versões, chamado consistência de atualização. Nele uma execução é dita consistente em atualização se e somente se as duas condições a seguir forem satisfeitas:

(1) Todas as transações de atualização são serializáveis;

(2) Cada transação somente de leitura R é serializável com relação

ao conjunto das transações que, direta ou indiretamente, atualizaram valores que foram lidos pela transação R;

Determinar se uma história é consistente em atualização é um problema NP-Completo [32]. Por isso, este critério foi adaptado e um algoritmo de verificação, aproximativo e de tempo polinomial, APPROX, foi desenvolvido para determinar de forma eficiente histórias legais.[32]

4.7.1 Critério de Corretude

Seja T uma transação que executa em uma história H.

Seja $LIVE_h(T)$ o conjunto mínimo fechado onde:

(a) $T \in LIVE_h(T)$

(b) Se $T' \in LIVE_h(T)$, para toda transação T'' tal que T' lê um valor de um item de dado escrito por T'', $T'' \in LIVE_h(T)$

H_{UPDATE} é a projeção da história H que inclui apenas as operações de todas as transações que executam alguma operação de escrita.

Uma determinada história S é correta se e somente:

(1) H_{UPDATE} é serializável por conflito;

(2) Para toda transação somente de leitura R em H, $S_H(R)$ é acíclico. Onde $S_H(R)$ é o grafo de serialização consistindo somente das transações em $LIVE_h(R)$

4.7.2 Implementando a Consistência de Atualização (F_MATRIX)

Uma implementação do algoritmo APPROX, chamada F_MATRIX, foi apresentada em [32]. A seguir, discutiremos seu funcionamento.

- **Funcionalidades no Servidor**
  - Durante cada ciclo, o servidor difunde o último valor *committed* dos itens de dados. Assim, o servidor mantém duas versões de cada item: o último valor *committed* e o último valor escrito.
  - Garante a serializabilidade por conflito de todas as transações de atualização.
  - Transmite uma matriz de controle, durante cada ciclo, que será usada pelos clientes para determinar se as transações somente leitura lêem valores consistentes.
- **Funcionalidades nos Clientes**
  - **Operação de leitura:** Antes de uma operação de leitura ser executada sobre um item de dado, durante um determinado ciclo do *broadcast*, as informações de controle transmitidas durante este ciclo são consultadas para determinar se a operação de leitura pode proceder. Caso a operação de leitura não possa proceder a transação é abortada.
  - **Operação de escrita:** Quando um item é escrito, a operação de escrita é executada em uma cópia local do item no cliente. Nenhuma checagem é feita.
  - **Commit:** Se a transação não executa nenhuma operação de escrita ela pode concluir. Caso contrário, a lista de todos os itens escritos e seus valores, além das operações de leitura e os ciclos onde elas ocorreram, são enviados para o servidor. O servidor então checa para ver se a transação de atualização pode executar o *commit* e comunica o resultado ao cliente [24]
  - **Abort:** Se a transação não executou nenhuma operação de escrita basta que ela seja interrompida Caso contrário, todas as cópias dos itens de dados escritos devem ser descartadas e a transação interrompida.

- **Natureza das Informações de Controle**

  As informações de controle são enviadas na forma de uma matriz C, de ordem n X n, onde n é o número de itens do banco de dados.

  Considerando que os itens de dados possuem identificadores eles podem ser representados da seguinte forma; $O_{b1}....O_{bn}$.

  Cada entrada da matriz de controle C, C(i,j), terá como valor o número de um determinado ciclo do *broadcast*.

  Seja H a história das transações de atualização *committed* no servidor e T_j a última transação *committed* que escreve em O_{bj}.

  A matriz de controle C é montada da seguinte forma:

  Vamos considerar que uma transação hipotética T_0 escreve todos os itens no ciclo 0. Então,

  $C(i,j) = max_{\ T' \in LIVE_H(T_j) \wedge T' \ escreve \ em \ O_{bi}} \ (C_{T'})$.

  Onde $C_{T'}$ é o número do ciclo onde T' executa o *commit*.

- **Validação das Leituras nos Clientes**

  Para uma determinada transação somente de leitura R em um cliente, o seguinte protocolo é seguido antes de cada operação de leitura:

  Seja RS(R) o conjunto, $(O_{bi}, ciclo)$, das leituras executadas anteriormente pela transação R, ou seja, R leu o último valor committed de O_{bi} até o início do ciclo *ciclo*.

  Então, a leitura sobre um item O_{bj} é permitida se e somente se:

  $\forall (O_{bi}, ciclo) \in RS(R) \ (C(i,j) < ciclo)$

  Se esta condição falhar então a transação é abortada.

O seguinte teorema é apresentado em [32]: Uma transação somente leitura R pode executar o commit seguindo o protocolo acima se e somente se SG(R) é acíclico.

Nesta abordagem, o cliente não necessita contactar o servidor para executar as operações de leitura, nem escutar continuamente o canal de *broadcast*. Além disso, algumas histórias que não seriam aceitas na serialização tradicional são consideradas corretas. Por outro lado, o cliente precisa ler a matriz de controle, enviada no início de cada *bcycle*. Para ler estas informações, o cliente necessita escutar o canal de *broadcast* durante um período de tempo maior, o que se torna problemático devido às limitações no fornecimento de energia pelas baterias dos computadores portáteis e ao alto custo da transmissão. O cliente também precisa armazenar uma matriz de ordem n X n (onde n é o número de itens do banco de dados), entretanto, os computadores portáteis têm, em geral, uma pequena capacidade de memória. Uma outra desvantagem consiste no fato de que determinar se uma história é consistente em atualização é um problema NP-Completo. Por isso, o algoritmo aproximativo polinomial APPROX é usado. Entretanto, este algoritmo descarta algumas histórias corretas segundo o critério da consistência de atualização. Outro fato importante é que o servidor precisa manter uma estrutura de dados $LIVE_h(T)$ para cada transação de atualização T, o que pode degradar a performance das transações submetidas no servidor.

4.8 Quadro Comparativo

A seguir apresentamos um quadro comparativo que sintetiza as principais diferenças entre os mecanismos para controle de concorrência em ambientes de *broadcast* analisados neste trabalho. Essa comparação é realizada com base nos seguintes parâmetros:

- **Concorrência:** O grau de concorrência pode ser determinado dividindo-se o número de transações concluídas(*committeds*) pela adição entre o número de transações iniciadas (submetidas) e o número de reinícios (caso alguma transação seja abortada e em seguida reiniciada).

- **Overhead:** Diz respeito ao processamento (execução) de atividades desnecessárias ou redundantes.

- **Tamanho das informações adicionais enviadas por** *broadcast***:** Esta é uma medida importante, pois a transmissão consome largura de banda e faz com que o cliente necessite escutar o canal de *broadcast* por um período de tempo maior. Além disso, o volume dos dados transmitidos em *broadcast* afeta o tempo de resposta das transações clientes, uma vez que o acesso aos dados é seqüencial os clientes terão que esperar que os dados do seu interesse apareçam no canal.

- **Tempo de Execução:** Significa a duração das transações somente de leitura. Este medida pode ser quantificada como sendo o número *bcycles* por transação.

- **Atualidade:** Estado do banco de dados visto pelos clientes.

- **Tolerância a Desconexão:** Indica se o método suporta a desconexão dos clientes. Esta medida é importante devido a limitada capacidade de energia das baterias dos computadores portáteis e a própria mobilidade dos clientes, que podem mover-se para áreas não cobertas pelos sistemas de comunicação.

	Relatório de Invalidação	Múltiplas Versões	Teste do Grafo de Serialização	Consistência de Atualização
Concorrência	Mínima	Moderada	Moderada	Alta
Overhead de Processamento	Pequeno	Moderado	Considerável (Mantém grafos no servidor e nos clientes)	Moderado
Tamanho das Informações Adicionais	Depende da taxa de atualização e do span	Depende da taxa de atualização e do span	Depende da atividade no servidor	Considerável
Tempo de Execução	Não afetada	Aumenta para transações longas	Não afetada	Não afetada
Atualidade	O estado quando a última leitura é executada	O estado quando a primeira leitura é executada	Um estado entre a primeira e a última operação	Um estado entre a primeira e a última operação
Tolerância a Desconexão	Nenhuma	Alguma, depende do span e da taxa de atualização	Nenhuma	Completa

Figura 4.1: Comparação entre os principais mecanismos para controle de concorrência em ambientes de broadcast.

4.9 Serializabilidade Semântica

Apesar de não ter sido proposta diretamente para ambientes de *broadcast*, a serializabilidade semântica será estendida a fim de ser utilizada como critério de corretude pelos mecanismo de controle de concorrência desenvolvidos para este ambiente.

A serializabilidade semântica, proposta em [8], baseia-se na utilização de informações semânticas sobre os objetos do banco de dados (e não sobre as transações). A idéia principal consiste em proporcionar diferentes visões de atomicidade para cada transação e, por esta razão, permitir um maior entrelaçamento entre as transações. Isto é alcançado através da aceitação de *schedules* que não são serializáveis, mas que preservam a consistência do banco de dados.

Para demonstrar a aplicabilidade e a viabilidade da serializabilidade semântica apresentaremos como exemplo uma aplicação de uma corretora de seguros hipotética. Consideraremos que a seguradora opera com dois tipos básicos de seguro: o seguro de veículos e o seguro de imóveis. Dois corretores, José e Maria, são responsáveis pela venda e pela gerência das informações que irão compor o cálculo do valor do seguro. A Figura 4.2 mostra a organização da companhia seguradora. As linhas contínuas representam diferentes tipos de seguro. As linhas pontilhadas as responsabilidades dos funcionários.

Em [8], Brayner et al. observam que algumas aplicações "vêem" o banco de dados como uma coleção de grupos de objetos distintos. Além disso, estas aplicações apresentam a seguinte propriedade: Com relação às operações de atualização, não existe nenhuma relação entre objetos de grupos diferentes, isto é, um objeto de um determinado grupo jamais será atualizado com base em valores de objetos que pertencem a outros grupos. Por exemplo, a nossa corretora hipotética "vê" o banco de dados BD como uma coleção de dois grupos de objetos distintos: um grupo consiste dos objetos relacionados ao seguro de veículos e o outro dos objetos associados ao seguro de imóveis. Logicamente, o resultado final de uma atualização sobre um objeto do seguro de veículos não será influenciado por um valor lido de um objeto do

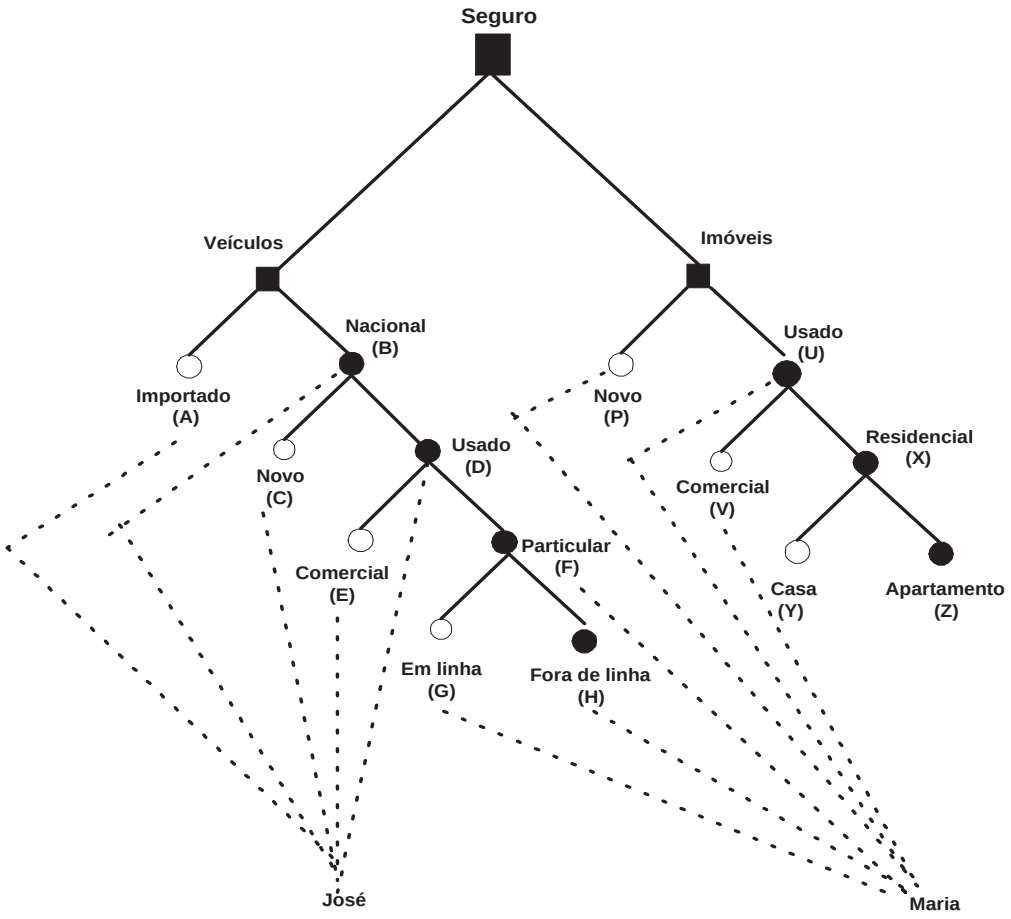

Figura 4.2: Organização de uma empresa seguradora.

seguro de imóveis. E vice versa.

A utilização de conhecimento semântico no processamento de transações introduz um alto grau de paralelismo e, conseqüentemente, uma maior cooperação entre as transações de longa duração.

4.9.1 O Modelo

A serializabilidade semântica baseia-se no fato que muitas aplicações não convencionais vêem o banco de dados como uma coleção de grupos de objetos logicamente disjuntos e que não existem dependências de atualização entre objetos de grupos distintos. Usando este conhecimento semântico, a unidade de atomicidade é refinada

e por conseguinte o paralelismo entre as transações é ampliado [5].

Unidades Semânticas

Um banco de dados pode ser visto por muitas aplicações como uma coleção de "ilhas semânticas", cada uma delas representando uma coleção de objetos (entidades do mundo real). Como foi mencionado anteriormente, não existem dependências de atualização entre as "ilhas semânticas". Essas ilhas semânticas recebem o nome de unidades semânticas.

Um conceito adicional chamado "depende de" precisa ser discutido antes da definição formal de unidade semântica. Dizemos que x "depende de" y, se o resultado de alguma operação de atualização sobre o objeto x é uma função de (isto é, depende de) um valor de y lido por algum programa da aplicação que acessa o BD, onde $x, y \in BD$. O conjunto dependências-de(x) representa todos os objetos para os quais x "depende de".

Definição 4 *Seja BD um banco de dados.* $SU_i, 0 < i \leq n$, *são unidades semânticas de BD, se e somente se:*

(i) $BD = \bigcup_{i=1 \text{ até } n} SU_i$,

(ii) $\forall\, 1 \leq i, j \leq n,\ i \neq j :\ SU_i \cap SU_j = \phi$, *e*

(iii) $(\forall x \in SU_i, y \in SU_j, i \neq j) \Rightarrow (x \notin conjuntodepende{-}de(y)) \wedge (y \notin conjunto\,depende - de(x))$

Intuitivamente, a condição (iii) diz que uma atualização sobre um objeto de uma determinada unidade semântica depende somente dos valores dos objetos da mesma unidade semântica.

Exemplo 1. Considere o BD = A,B,C,D,E,F,G,H,P,U,V,X,Y,Z apresentado na seção anterior. Como foi dito anteriormente, uma aplicação de uma corretora de seguros vê o BD como um conjunto de dois grupos de objetos disjuntos: o grupo de objetos relacionados com o seguro de veículos e o grupo de objetos associados

ao seguro de imóveis. Além disso, podemos observar facilmente que uma operação de atualização sobre um objeto relacionado ao seguro de veículos nunca dependerá dos valores dos objetos associados ao seguro de imóveis. Assim, duas unidades semânticas podem ser especificadas para esta aplicação: SU_v = A,B,C,D,E,F,G,H representando o conjunto dos objetos do seguro de veículos, e SU_i = P,U,V,X,Y,Z representando o conjunto de objetos do seguro de imóveis.

Obviamente, se o gerenciamento das transações for explorar a noção de unidades semânticas, o *scheduler* deve receber informações (dos usuários) sobre a especificação destas unidades. Uma das possibilidades é descrever as unidades semânticas utilizando-se metadados junto com o esquema do banco de dados e dar ao *scheduler* a habilidade de obter estas informações. Uma proposta alternativa consiste em fornecer uma especificação das unidades semânticas para o *scheduler*. Em ambos os casos, o *scheduler* deve suportar o modelo transacional clássico se nenhuma informação sobre as unidades semânticas estiver disponível.

Transações

Uma transação consiste em uma seqüência finita de operações de leitura e escrita sobre objetos do banco de dados. A notação $r_i(x)$ e $w_i(x)$ será utilizada para representar operações de leitura e escrita executadas por uma transação T_i sobre um objeto x. $OP(T_i)$ representa o conjunto de todas as operações executadas pela transação T_i. $OP_{SU_a}(T_i)$ representa o conjunto das operações de T_i que são executadas sobre os objetos da unidade semântica SU_a. Com $<_{T_i}$, vamos denotar a relação de precedência entre duas operações da transação T_i. Será assumido que se uma transação executa de modo isolado das outras transações então ela preserva a consistência do banco de dados.

No modelo transacional clássico, assume-se que uma operação de escrita em uma transação T é função de todos os valores lidos anteriormente pelas operações de T. Entretanto, se a noção de unidade semântica é utilizada, uma operação de escrita em

uma transação T é função somente dos valores lidos anteriormente pelas operações de T que pertencem a mesma unidade semântica do objeto que está sendo escrito.

Definição 5 *Seja $SU_a \subseteq BD$ uma unidade semântica de um banco de dados BD e x um objeto de BD, onde $x \in SU_a$. O conjunto das leituras anteriores semanticamente relevantes para uma operação de escrita $w_i(x)$ em uma transação T_i é definida como a seguir:*

$$s\_pre\_read(w_i(x)) := \{r_i(y) : (r_i(y) <_{T_i} \ w_i(x)) \wedge (y \in SU_a)\}$$

Definição 6 *Seja T uma transação. A semântica de Herbrand para uma determinada operação de escrita $w_i(x) \in OP(T)$, denotada por $H_T(w(x))$, é definida como a seguir:*

$H_T(w(x)) := f_{w(x)}(r(y_1), r(_{y2}), \ ... \ , r(y_m)$, *onde*

$r(y_k) \in s\_pre\_read(w(x))$, $0 < k \leq m$

Considere que o conjunto $read(w(x))$ consiste de todas as operações de leitura $r(y_k) \in OP(T)$, onde $r(y_k) < w(x)$ e $w(x) \in OP(T)$ para $0 < k < m$. Como vimos anteriormente, no modelo clássico de processamento de transações, uma operação de escrita $w(x)$ é uma função de todos os elementos pertencentes ao conjunto $read(w(x))$. Podemos observar facilmente que $s\_pre\_read(w_i(x)) \subseteq read(w(x))$. Logicamente, a cardinalidade do conjunto $s\_pre\_read(w_i(x))$ é menor ou igual a cardinalidade do conjunto $read(w(x))$. Por esta razão, a serializabilidade semântica provê uma identificação mais precisa das operações de leitura que terão influência sobre uma determinada operação de escrita . Esta propriedade proporciona o refinamento do conceito de unidade de atomicidade, como veremos a seguir.

Definição 7 *Seja T uma transação no banco de dados $BD = \bigcup_{i=1 \ até \ n} SU_i$, onde SU_i, $0 < i \leq n$, são unidades semânticas de BD. Para cada $OP_{SU_i}(T) \neq \phi$, define-se uma unidade atômica de T, denotada por $\prod_{[OP_{SU_i}]}(T)$. Cada unidade atômica consiste de uma subseqüência das operações de T executadas sobre os objetos de SU_i, onde:*

$\forall\, p,\, q \in OP_{SU_i}(T),\; p \prod_{[OP_{SU_i}(T)]}(T) q \;\Leftrightarrow\; p <_T q$

Considere a seguinte transação:

$T_{maria} = r_{maria}(E)\, r_{maria}(P)\, w_{maria}(F)\, w_{maria}(U)\, w_{maria}(V)$

Observe que a transação T_{maria} consiste de duas unidades atômicas:

$\prod_{[OP_{SU_{veículo}}(T_{maria})]}(T_{maria}) = r_{maria}(E)\, w_{maria}(F)\, e$
$\prod_{[OP_{SU_{imóel}}(T_{maria})]}(T_{maria}) = r_{maria}(P)\, w_{maria}(U)\, w_{maria}(V)$

Obviamente, nesta abordagem uma transação pode consistir de várias unidades atômicas. Estas unidades podem ser identificadas automaticamente e dinamicamente pelo *scheduler*.

Execução Correta de Transações Concorrentes

As transações são executadas concorrentemente através do entrelaçamento de suas operações. A execução de várias transações entrelaçadas é chamada de *schedule*. Logicamente, nem todos os *schedules* são válidos, ou seja, preservam a consistência do banco de dados. Por esta razão, a identificação de *schedules* corretos é um ponto fundamental no gerenciamento de transações.

Neste modelo é introduzido o conceito de *schedules* semanticamente seriais. Estes *schedules* preservam a consistência do banco de dados. Uma classe de *schedules* que não são semanticamente seriais, mas que preservam a consistência do banco de dados, por terem um comportamento semelhante aos *schedules* seriais, também foi observada. Estes *schedules* foram chamados de *schedules* semanticamente serializáveis. Desta forma, podem ser definidos mecanismos automáticos para o controle da concorrência, que utilizem informações semânticas sobre o banco de dados a fim aumentar o paralelismo entre as transações. Isto é conseguido relaxando-se o conceito de que uma transação é uma unidade de atomicidade, ou seja, neste modelo uma transação pode consistir de uma ou mais unidades de atomicidade (unidade de

atomicidade $\subseteq$ unidade de recuperação $\subseteq$ transação), permitindo assim um maior entrelaçamento entre as transações [8].

Outro importante benefício da serializabilidade semântica é a possibilidade de determinar se um *schedule* é ou não semanticamente serializável verificando-se a existência de ciclos em um grafo direcionado, chamado grafo de serialização semântica (S_eSG). Esta estratégia é similar à idéia do grafo de serialização proposta no modelo clássico de transações [11]. Entretanto, o grafo de serialização semântica (S_eSG) apresenta arestas nomeadas em contraste com o grafo de serialização convencional, como mostraremos a seguir.

Definição 8 *Seja S um schedule sobre o conjunto de transações $T = \{T_1, T_2, \ldots, T_n\}$. O grafo de serialização semântica para S, denotado por $S_eSG(S)$, é um grafo direcionado $S_eSG(S) = (N, E)$ no qual $N = T$. E representa o conjunto das arestas nomeadas $T_i \rightarrow^{SU_n} T_j$, onde $T_i, T_j \in N$ e existem duas operações $p \in OP(T_i), q \in OP(T_j)$, $p <_S q$, sobre um determinado objeto da unidade semântica SU_n, as quais estão em conflito.*

O seguinte teorema é demonstrado em [5]: Um *schedule* S é semanticamente serializável se e somente se o grafo de serialização semântica para S, $S_eSG(S)$, não contém ciclo sobre as arestas com a mesma nomeação.

Podemos observar facilmente que o algoritimo para checar se o grafo de serialização semântica para S é acíclico ou não pode ser de ordem $O(m.n^2)$, onde m é o número de "nomeações" e n o número de transações em S. Logo, o grafo de serialização semântica proporciona um método eficiente de identificar *schedules* semanticamente serializáveis.

Para ilustrar a utilização do grafo de serialização semântica, tomemos o exemplo 1.

Considere o seguinte conjunto de transações:

$T_{José} = r_{José}(X)\, r_{José}(E)\, w_{José}(X)\, C_{José}$

$T_{Maria} = r_{Maria}(X)\, r_{Maria}(E)\, w_{Maria}(E)\, C_{Maria}$

Considere agora o *schedule* S mostrado na figura 4.3.

$$r_{Maria}(X)\ r_{José}(X)\ r_{José}(E)\ W_{José}(X)\ r_{Maria}(E)\ W_{Maria}(E)\ C_{José}\ C_{Maria}$$

Figura 4.3: Schedule S.

A figura 4.4 mostra o grafo de serialização semântica para o *schedule* S.

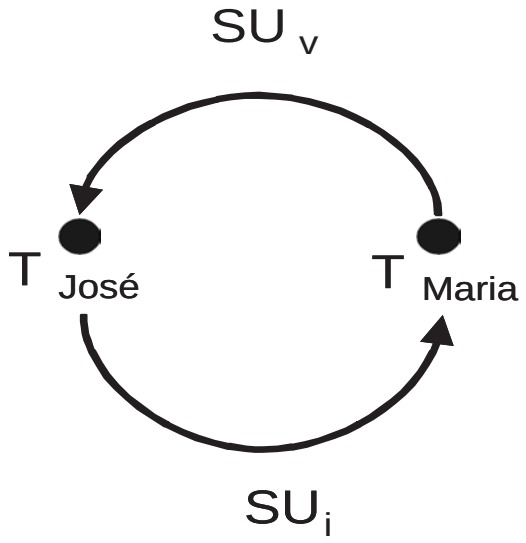

Figura 4.4: Grafo de serialização semântica para o schedule S.

A aresta $T_{José} \rightarrow^{SU_v} T_{Maria}$ indica que pelo menos uma operação $p \in OP(T_{José})$ é executada sobre um objeto da unidade semântica SU_v e conflita com uma operação $q \in OP(T_{Maria})$ em S, onde $p <_S q$.

Observe que o grafo de serialização semântica para S, denotado por $S_eSG(S)$, não contém ciclo sobre arestas com a mesma "nomeação". Logo, o *schedule* S é semanticamente serializável.

Como discutimos anteriormente, em um ambiente dinâmico é importante que o critério de corretude seja "*prefix-closed*", isto é, se um *schedule* S é correto então

qualquer prefixo de S também é correto. Em [8] encontramos o seguinte teorema: A serializabilidade semântica é "*prefix-closed*".

Capítulo 5

Teste do Grafo de Serialização Temporal

5.1 Introdução

Para solucionar o problema do controle de concorrência em ambientes de *broadcast* duas estratégias distintas podem ser utilizadas. Uma estratégia objetiva o desenvolvimento de protocolos de controle de concorrência baseados em algum modelo transacional já existente. A segunda estratégia consiste no desenvolvimento de novos modelos para o processamento de transações que sejam mais adequados a ambientes de *broadcast*. Seguindo a primeira abordagem, apresentamos o protocolo TGST (Teste do Grafo de Serialização Temporal) proposto inicialmente em [9], o qual utiliza a serializabilidade como critério de corretude. Neste trabalho, propomos também uma extensão ao protocolo TGST a fim de utilizar como critério de corretude a serializabilidade semântica proposta em [8].

Este capítulo esta organizado da seguinte forma: Na seção 5.2 apresentamos o protocolo do teste do grafo de serialização temporal (TGST), na seção 5.3 discutimos o funcionamento do protocolo TGST considerando a presença de operações de atualização nos clientes móveis e na seção 5.4 mostramos como estender o protocolo

TGST para utilizar a serializabilidade semântica como critério de corretude.

5.2 Teste do Grafo de Serialização Temporal

Como já mencionado anteriormente, os protocolos convencionais, que utilizam a serializabilidade como critério de corretude, são ineficientes para controlar a concorrência em ambientes de *broadcast*. Por esse motivo, diversos pesquisadores propuseram variados modelos e protocolos para solucionar o problema do controle de concorrência em ambientes de broadcast. Contudo, como mostrado no capítulo 4, tais propostas apresentam várias desvantagens.

Nesta seção, descreveremos e discutiremos um protocolo que garante um controle de concorrência eficiente em ambientes de *broadcast*.

O protocolo proposto, denominado Teste do Grafo de Serialização Temporal (TGST), é baseado em uma estratégia similar a utilizada pelo mecanismo de teste do grafo de serialização proposto em [11]: o monitoramento e gerenciamento dinâmico de um grafo que deve ser sempre acíclico. Em contraste com o protocolo do teste do grafo de serialização, o protocolo TGST explora informações temporais referentes ao momento em que um item de dado foi lido ou atualizado. Devido à complexidade de se permitir operações de atualização nos clientes, iremos considerar inicialmente que as transações nos clientes móveis são somente de leitura.

Na nossa abordagem as funções para o controle de concorrência estão divididas entre os clientes e o servidor. Portanto, assumimos que o servidor e os clientes apresentam funcionalidades específicas para o gerenciamento das transações. A seguir descreveremos estas funcionalidades.

Durante cada ciclo de *broadcast*, o servidor difunde os valores dos itens de dados juntamente com um marcador de tempo. Vamos assumir que os valores dos itens de dados enviados em *broadcast* durante cada ciclo correspondem ao estado do banco de dados até o início do *broadcast*, ou seja, correspondem aos valores produzidos por todas as transações que executaram operações de *commit* até o início do ciclo.

Iremos denominar essas transações de *"committed"*. Desta forma, o servidor mantém duas versões de cada item: o último valor *committed* e o último valor escrito. O servidor monta e gerencia o grafo de serialização temporal para o *schedule global*, envolvendo tanto as transações do servidor quanto as dos clientes móveis.

Periodicamente, o cliente deve enviar um pacote (mensagem) ao servidor contendo os itens lidos até o momento, juntamente com os marcadores de tempo correspondentes. As leituras já informadas não precisam ser enviadas novamente. Quando um cliente recebe um pedido de *commit* ou *abort* para uma transação móvel T_i, este deve enviar uma mensagem ao servidor contendo essa solicitação para T_i. O cliente deve esperar a resposta para efetuar a operação de *commit* ou de *abort*.

5.2.1 Cenário Exemplo

Para ilustrar a utilização e eficácia de nossa proposta, vamos tomar como exemplo uma aplicação de comércio eletrônico. Nesta aplicação, as ações das principais empresas de tecnologia são disponibilizadas para leilão em uma bolsa de valores eletrônica. Considere o seguinte conjunto de transações, que lêem e atualizam os valores das ações:

$T_1 : r_1(IBM)\, r_1(SUN)$

$T_2 : w_2(IBM)\, C_2$

$T_3 : r_3(IBM)\, r_3(SUN)$

$T_4 : w_4(SUN)\, C_4$

$T_5 : w_5(SUN)\, C_5$

As transações T_2 , T_4 e T_5 são executadas no servidor. Por outro lado, a transação T_1 é executada no cliente A, enquanto a transação T_3 no cliente B. Considere agora o *schedule* global SG apresentado na figura 5.1.

No cenário de execução apresentado na figura 5.1, vamos assumir que os pacotes com informação sobre as leituras executadas pelas transações clientes durante o $bcycle_n$ chegam ao servidor antes do envio do $bcast_{n+1}$. O grafo de serialização para

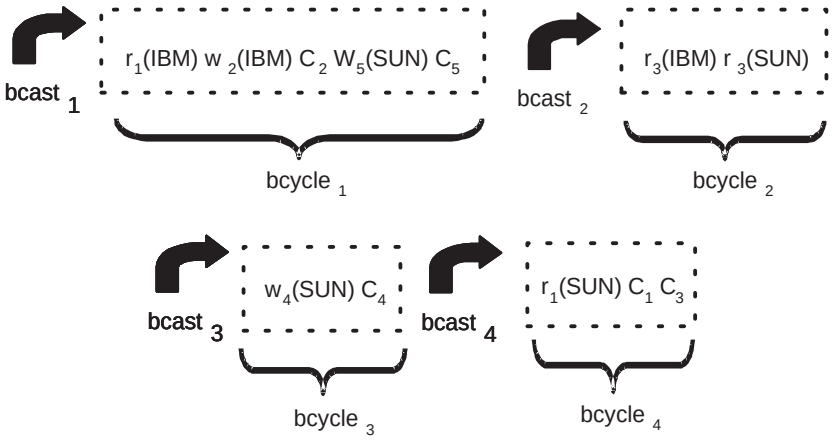

Figura 5.1: Schedule SG.

o *schedule* SG é ilustrado na figura 5.2. Observe que esse grafo apresenta um ciclo da forma $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. Portanto, o *schedule* SG não é serializável por conflito (não é correto).

Agora, vamos supor que, por um motivo qualquer (problemas nas linhas de comunicação, por exemplo), o pacote contendo informação sobre as leituras executadas por T_3 durante o $bcycle_2$ atrase. Neste caso, o servidor irá "ver" o *schedule* SG' mostrado na figura 5.3. O grafo de serialização para o *schedule* SG' é mostrado na figura 5.4.

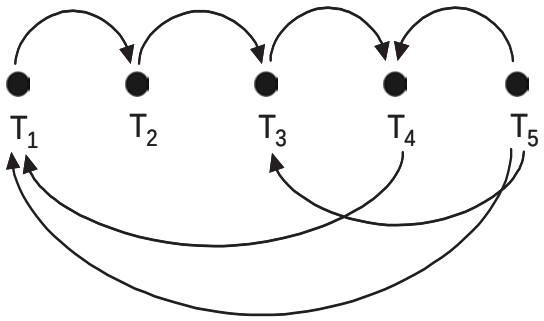

Figura 5.2: Grafo de serialização do Schedule SG.

Como podemos ver na figura 5.4, o grafo de serialização de SG' não apresenta ciclo. Conseqüentemente, o *schedule* SG' será considerado serializável por conflito,

Figura 5.3: Schedule SG': Schedule SG com informações atrasadas.

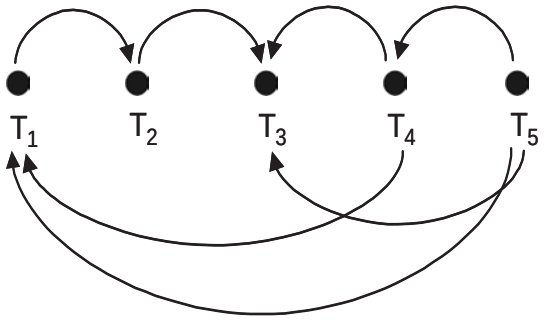

Figura 5.4: Grafo de serialização do Schedule SG com informações atrasadas.

ou seja, correto. Porém, como vimos anteriormente, a operação $r_3(SUN)$ foi temporalmente executada antes de $w_4(SUN)$. Isso significa que a aresta correta entre T_3 *e* T_4 no grafo de serialização deveria ser $T_3 \rightarrow T_4$ (veja figura 5.2) e não $T_4 \rightarrow T_3$, como mostra a figura 5.4. Portanto, uma execução incorreta foi considerada correta indevidamente. Para evitar a ocorrência de fenômenos como esse, propomos o protocolo TGST básico, descrito na seção a seguir.

5.2.2 Protocolo TGST Básico

O protocolo TGST consiste basicamente em monitorar e gerenciar um grafo que deve ser sempre acíclico. O grafo mantido pelo protocolo TGST é chamado grafo de serialização temporal. Este grafo é construído a partir de informações temporais referentes ao momento em que um item de dado foi lido ou atualizado. Estas informações temporais consistem em marcadores de tempo definidos como descrito a seguir. Seja $C(p_i(x))$ o valor do marcador de tempo para uma operação $p_i(x)$, se T_i é uma transação executada no servidor então $C(p_i(x))$ é o número do ciclo onde a operação $p_i(x)$ foi executada. Por outro lado, se T_i é uma transação móvel, então

$C(p_i(x))$ é o número do último ciclo de *broadcast* já concluído.

Seja S um *schedule* sobre um conjunto de transações $T = \{T_1, T_2, ..., T_n\}$. Definimos o grafo de serialização temporal (GST) para S, representado pela notação GST(S), como um grafo direcionado GST(S) = (N,A), onde N representa as transações de T (N = T). As arestas representam a existência de um conflito entre duas transações pertencentes a T. Assim, A representa o conjunto das arestas $T_i \rightarrow T_j$, onde T_i, $T_j \in N$ e existem duas operações $p \in OP(T_i)$, $q \in OP(T_j)$, onde p conflita com q e $C(p) \leq C(q)$.

Um escalonador (*scheduler*) implementando o protocolo TGST funciona como descrito a seguir. Quando um escalonador inicia sua execução o GST é criado como um grafo vazio. Durante cada ciclo de *broadcast*, o servidor difunde os valores dos itens de dados (último valor escrito por transações *committed*) juntamente com os marcadores de tempo correspondentes (número do último ciclo de *broadcast* já concluído). Este valor pode ser enviado no cabeçalho da mensagem ou junto com cada item de dado. A cada operação de leitura, o cliente guarda o valor e o identificador do item lido, juntamente com o marcador de tempo correspondente.

Periodicamente, o cliente informa ao servidor quais as operações de leitura executadas por ele. Isto é, o cliente envia um pacote contendo para cada leitura o identificador do item lido e o marcador de tempo correspondente.

A medida em que o escalonador recebe a primeira operação de uma nova transação T_i, um nó representando esta transação é inserido no grafo. Para cada operação $p_i(x) \in OP(T_i)$ que o escalonador recebe, ele checa se existe uma operação $q_j(x) \in OP(T_j)$ que conflita com $p_i(x)$ e que já foi escalonada. Caso $q_j(x)$ exista, será executada uma **verificação temporal** da seguinte forma: Se $C(q_j(x)) \leq C(p_i(x))$ o escalonador insere uma aresta da forma $T_j \rightarrow T_i$, caso contrário uma aresta da forma $T_i \rightarrow T_j$ será inserida no grafo. Em seguida, o escalonador verifica se a nova aresta introduz um ciclo no grafo. Em caso afirmativo, o escalonador rejeita a operação $p_i(x)$, desfaz o efeito das operações de T_i e remove a aresta inserida. Caso contrário, $p_i(x)$ é aceita e escalonada.

Para o cenário exemplo apresentado na seção 5.1, pode-se facilmente ver que o grafo produzido pelo protocolo TGST básico corresponde ao grafo de serialização correto para o *schedule* SG (figura 5.2). O protocolo descrito acima irá garantir que, mesmo que o pacote contendo a informação da operação $r_3(SUN)$ atrase, o escalonador identifique que $C(w_4(SUN)) > C(r_3(SUN))$, inserindo, dessa forma, a aresta $T_3 \rightarrow T_4$, e não $T_4 \rightarrow T_3$. Como se pode observar, o protocolo TGST captura a informação que a operação $r_3(SUN)$ foi temporalmente executada antes da operação $w_4(SUN)$.

5.2.3 Protocolo TGST Estendido

O protocolo TGST básico descrito na seção 5.2.2 pode ainda apresentar os seguintes fenômenos: (i) considerar incorreto um *schedule* serializável por conflito e (ii) considerar correto um *schedule* que não é serializável por conflito. Para ilustrar esses fenômenos, considere os dois exemplos apresentados a seguir.

Exemplo 1. O protocolo TGST básico considera incorreto um *schedule* serializável por conflito.

Considere o *schedule* SG' mostrado na figura 5.5. De acordo com o protocolo TGST básico, o grafo de serialização temporal para o *schedule* SG' a ser construído é semelhante ao grafo mostrado na figura 5.2.

O grafo apresentado na figura 5.2 apresenta um ciclo, logo o *schedule* SG' não é serializável por conflito, ou seja, não é correto.

Contudo, o valor lido pela operação $r_1(SUN)$ não corresponde ao valor escrito por $w_4(SUN)$. Observe que a transação T_4 ainda não executou o *commit* e os valores enviados em *broadcast* são resultantes apenas das operações das transações *committed*. Portanto, o valor lido pela operação $r_1(SUN)$ é anterior ao valor gerado pela operação $w_4(SUN)$. O grafo de serialização temporal correto para o *schedule* SG' é mostrado na figura 5.6. Assim, um *schedule* serializável por conflito (correto) foi considerado incorreto. Naturalmente, este fenômeno deve ser evitado, pois

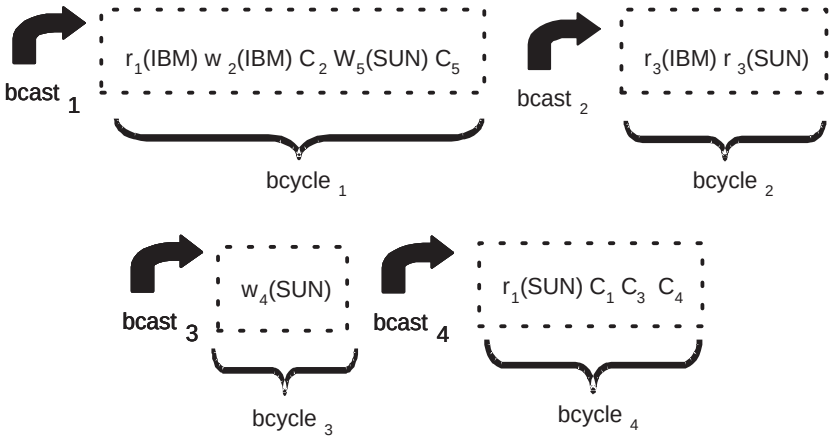

Figura 5.5: Schedule SG'.

provocaria uma operação de *abort* para a transação T_1, de acordo com o protocolo TGST básico.

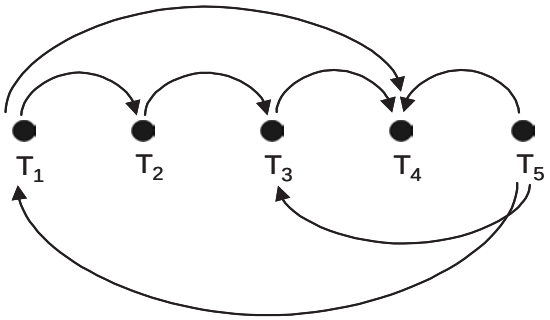

Figura 5.6: Grafo de serialização correto do Schedule SG'.

Exemplo 2. O protocolo TGST básico considera correto um *schedule* que não é serializável por conflito.

Para ilustrar a ocorrência desse fenômeno, considere o *schedule* SG'' mostrado na figura 5.7. Observe que o GST de SG'' (figura 5.8) não contém ciclos. Portanto, poderíamos afirmar que o *schedule* SG'' é correto. Contudo, o valor lido pela operação $r_3(SUN)$ não corresponde ao valor escrito por $w_4(SUN)$. Isso se deve ao fato de que a transação T_4 ainda não executou sua operação *commit*.

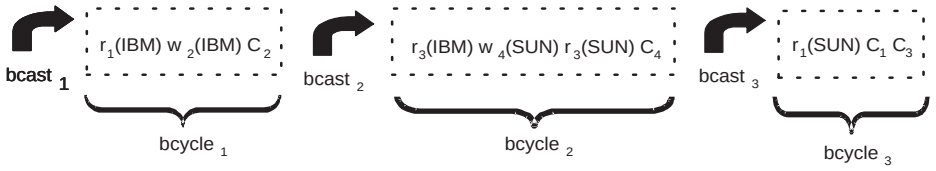

Figura 5.7: Schedule SG''.

Assim, o valor lido pela operação $r_3(SUN)$ é anterior ao valor gerado pela operação $w_4(SUN)$. Na figura 5.9, apresentamos o grafo de serialização temporal correto para o *schedule* SG''. O grafo da figura 5.9 é o que realmente captura a ordem correta de execução das operações do *schedule* SG''. Como podemos observar, existe um ciclo da forma $(T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1)$ em $GST(SG'')$. Conseqüentemente o *schedule* SG'' não é serializável por conflito. Em outras palavras, o *schedule* SG'' não é correto, podendo, portanto, produzir inconsistências no banco de dados.

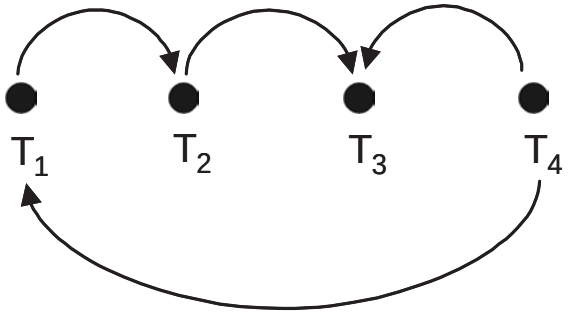

Figura 5.8: Grafo de serialização do Schedule SG''.

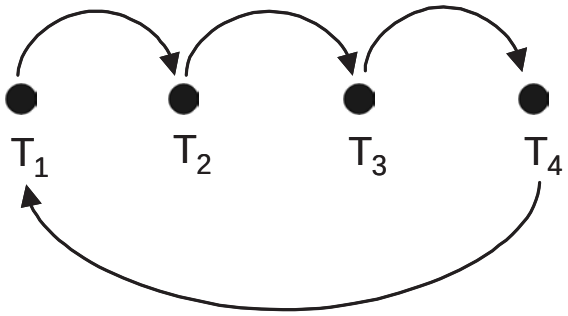

Figura 5.9: Grafo de serialização correto do Schedule SG''.

Para evitar a ocorrência desses fenômenos, propomos uma extensão ao protocolo TGST básico. Esta extensão está baseada na seguinte estratégia. Para cada item de dado x pertencente ao banco de dados, o servidor de *broadcast* envia, juntamente com o objeto x, um marcador de tempo definido como descrito a seguir. O marcador de tempo representa o número do ciclo que a transação T_i executou sua operação de *commit*, se $w_i(x) \in OP(T_i)$ e T_i é a última transação que executou uma operação de escrita sobre x. Quando uma transação móvel T_k executar uma operação $p_k(x)$, esse marcador de tempo será associado a $p_k(x)$. Para as operações das transações executadas no servidor o marcador de tempo continua sendo o número do ciclo onde a operação é realizada.

Com base no marcador de tempo descrito no parágrafo anterior, o grafo de serialização temporal é construído como descrito a seguir:

Passo 1. Para cada operação $p_i(x) \in OP(T_i)$ que o escalonador recebe, ele checa se existe uma operação $q_j(x) \in OP(T_j)$ que conflita com p_i(x) e que já foi escalonada. Caso q_j(x) exista, uma aresta será inserida entre T_i e T_j. Para que esta aresta seja incluída corretamente deveremos considerar dois casos distintos, os quais descrevemos a seguir:

Caso 1. T_i é uma transação executada no servidor. Neste caso, será executada uma verificação temporal da seguinte forma:

Se $C(q_j(x)) \leq C(p_i(x))$
  Então o escalonador insere uma aresta da forma $T_j \rightarrow T_i$.
Se não
  uma aresta da forma $T_i \rightarrow T_j$ será inserida no grafo.

Caso 2. T_i é uma transação móvel. Neste caso, será executada uma verificação temporal da seguinte forma:

Se $C(q_j(x)) < C(p_i(x))$
  Então o escalonador insere uma aresta da forma $T_j \rightarrow T_i$
Se não
  Se $C(q_j(x)) > C(p_i(x))$
    Então o escalonador insere uma aresta da forma $T_i \rightarrow T_j$
  Se não
    Se T_j já executou o *commit*
      Então o escalonador insere uma aresta da forma $T_j \rightarrow T_i$
    Se não o escalonador insere uma aresta da forma $T_i \rightarrow T_j$

Passo 2. O escalonador verifica se a nova aresta introduz um ciclo no grafo de serialização temporal. Em caso afirmativo, o escalonador rejeita a operação $p_i(x)$, desfaz o efeito das operações de T_i e remove a aresta inserida. Caso contrário, $p_i(x)$ é aceita e escalonada.

Para demonstrar a eficácia do protocolo TGST estendido, observe novamente os dois exemplos apresentados anteriormente (Exemplo 1 e Exemplo 2). No primeiro exemplo, durante a montagem do grafo de serialização temporal, o escalonador verificaria que $C(w_4(SUN)) > C(r_1(SUN))$ e iria inserir a aresta $T_1 \rightarrow T_4$, como mostra a figura 5.6, e não $T_4 \rightarrow T_1$, como mostrado na figura 5.2. No segundo exemplo, teríamos que $C(w_4(SUN)) > C(r_3(SUN))$ e, nesse caso, o escalonador adiciona a aresta $T_3 \rightarrow T_4$ ao grafo (figura 5.9), e não a aresta $T_4 \rightarrow T_3$, como mostrado na figura 5.8.

O mecanismo para controle de concorrência que estamos propondo utiliza serializabilidade para garantir a consistência dos dados. Vale ressaltar que serializabilidade já se tornou um padrão para controle de concorrência em bancos de dados. Por esse motivo, praticamente todo SGBD existente no mercado implementa este critério de corretude. A seguir demonstraremos que todo *schedule* produzido pelo protocolo TGST estendido é serializável por conflito, ou seja garante a consistência do banco de dados.

Teorema 5.1 *Seja TGST o conjunto de schedules sobre um conjunto T de transações produzido pelo protocolo TGST estendido e CSR o conjunto de todos os schedules serializáveis por conflito sobre T. Então TGST=CSR.*

Prova. É fácil mostrar que $TGST \subset CSR$. Nós precisamos apenas observar que todo *schedule* global S produzido pelo protocolo TGST estendido apresenta uma grafo de serialização temporal acíclico. Por definição, grafo de serialização temporal de S representa o grafo de serialização convencional para S, adicionado de informações temporais para capturar a ordem correta de execução das operações em S. Dessa forma, se o GST para S é acíclico, o grafo de serialização também é. Portanto, $S \in CSR$, conseqüentemente $TGST \subset CSR$. Para provar que $TGST \supset CSR$, precisamos mostrar que todo *schedule* $S \in CSR$ pode ser produzido pelo protocolo TGST estendido. Nós podemos mostrar isso por indução no tamanho de S que toda operação p em S não pode originar um ciclo no GST. Por esse motivo a operação p pode ser executada. Como já mencionado anteriormente, o GST representa o grafo de serialização convencional para S, adicionado de informações temporais.

5.2.4 Considerações Sobre a Comunicação das Leituras

Periodicamente, o cliente deve enviar um pacote ao servidor contendo um conjunto de itens lidos e seus marcadores de tempo. Cada pacote contém somente os itens lidos após o envio do pacote anterior. Estes pacotes recebem uma numeração única em cada cliente de acordo com a ordem em que são enviados. Nós identificamos três estratégias distintas para definir o momento em que o cliente deve enviar um pacote:

(i) Após cada operação de leitura: Neste caso um *overhead* de comunicação é gerado. Por outro lado, assim que um ciclo ocorre no TGST ele é detectado e uma mensagem é enviada ao cliente abortando a transação.

(ii) Ao final da transação: Nesta estratégia, o número de pacotes enviados é minimizado. Entretanto, se ocorrer um ciclo ele só será detectado ao final da transação.

(iii) A cada "n" itens lidos ou a cada intervalo de tempo "t" : Ciclos podem não ser detectados de maneira imediata. Porém, não é necessário que a transação termine para que ela seja abortada. Desta forma podemos encontrar um meio termo ótimo que pode ser usado como parâmetro para o escalonador .

Portanto, a idéia é permitir que a estratégia a ser utilizada seja definida como parâmetro pelo usuário. Com isso, o usuário poderia definir esse parâmetro em função da aplicação. Para aplicações com poucos pontos de *hot spots*, poderia ser definida, por exemplo, a estratégia de enviar os pacotes no final de cada transação.

5.2.5 Pontos de Falha na Comunicação

As redes de comunicação sem fio apresentam severas variações nas condições de comunicação. Este problema deve-se principalmente a dois motivos. O primeiro é que a largura de banda efetiva por usuário é bastante pequena. O segundo é que a taxa de erro na comunicação sem fio é significativamente maior que em uma rede fixa. Portanto, qualquer protocolo para controlar concorrência em ambientes de computação móvel deve considerar falhas na comunicação. A seguir descrevemos os tipos de falha que podem afetar o protocolo TGST e como solucionar estes problemas.

A mensagem que informa as leituras executadas pelo cliente foi perdida. Como os pacotes são numerados segundo a ordem de envio, se o servidor receber um pacote com uma numeração diferente da esperada ele colocará no cabeçalho dos próximos *bcasts* o pedido para que o cliente reenvie o pacote perdido. Quando este pacote for recebido o servidor retira a mensagem do cabeçalho.

A mensagem de pedido de *commit* ou *abort* enviada pelo cliente foi perdida. A

solução para este tipo de falha consiste na utilização de um mecanismo de *timeout*. Nesse caso, o cliente espera a resposta por um determinado período de tempo e se ela não chegar ele retransmite a mensagem com o pedido.

O servidor recebe um pedido de *commit* e estão faltando pacotes. O servidor colocará no cabeçalho dos próximos *bcasts* o pedido para que o cliente reenvie os pacote que estão faltando.

5.2.6 Um Método Coletor de Lixo

A manutenção do grafo pode consumir muito espaço de memória. A fim de minimizar este problema, nós propomos um procedimento coletor de lixo. A sua principal funcionalidade é remover com segurança nós e arestas do grafo. Isto significa que alguns nós e arestas do grafo são removidos pelo método coletor de lixo sem colocar em risco a corretude do protocolo TGST.

O procedimento coletor de lixo sobre um grafo = (N, E) é baseado no seguinte protocolo.

Passo 1: Para cada $G_i \in N$ faça:

> Se a última operação de T_i já foi escalonada e se não existe nenhuma aresta chegando em G_i , então as arestas com origem em G_i podem ser removidas do grafo.

Passo 2: Se um nó representando uma transação *committed* G_i é desconectado do grafo, ele pode ser removido.

A fim de mostrar que o primeiro passo do protocolo descrito acima não põe em risco a corretude de um TGST *scheduler* , considere o seguinte cenário. A última operação de T_i já foi executada. Neste caso, nenhuma aresta chegando em G_i pode vir a ser inserida. Se não existem arestas chegando em G_i , então G_i não pode se envolver mais em nenhum ciclo. Como resultado, as arestas com origem em G_i podem ser removidas. A prova do segundo passo é uma conseqüência do primeiro.

Desta forma, a execução do método coletor de lixo não afeta a corretude do protocolo TGST.

5.2.7 Considerações Sobre o Protocolo TGST

Além de utilizar um critério de corretude já consolidado, nossa abordagem ainda apresenta as seguintes vantagens. O cliente não necessita contactar o servidor para solicitar as operações de leitura, nem escutar continuamente o canal de *broadcast*. Diferentemente das propostas apresentadas em [31] e [32], na nossa abordagem as informações enviadas do servidor para os clientes a cada ciclo consistem apenas dos valores dos itens de dados e seus marcadores de tempo. Essa característica garante que a utilização do canal de *broadcast* é minimizada, o que proporciona uma economia no custo da transmissão e da limitada capacidade de energia dos computadores portáteis. Além disso, os clientes só precisam armazenar os valores dos itens de dados de seu real interesse, o que economiza os seus parcos recursos de memória. Essa última propriedade não é garantida pelas propostas [31] e [32], que foram discutidas no capítulo 3.

Por outro lado, o servidor precisa manter um grafo de serialização de todas as transações *committed* e ativas, incluindo tanto as transações do servidor quanto as transações somente de leitura dos clientes móveis, o que pode comprometer a escalabilidade. Outra desvantagem consiste no fato dos clientes terem que comunicar ao servidor os itens lidos por suas transações.

5.3 Atualização nos Clientes Móveis

Devido à complexidade de se permitir operações de atualização nos clientes móveis, consideramos até aqui que as transações executadas nestes clientes eram somente de leitura. Entretanto, as operações de atualização estão presentes na grande maioria das aplicações, sendo portanto de fundamental importância considerá-las em nossos protocolos.

Com esta finalidade iremos modificar o protocolo TGST para permitir operações de atualização. A estratégia utilizada consiste em encarar as operações de atualização pertencentes a transações móveis como se estas pertencessem a transações do servidor. Na verdade, uma operação de atualização $w_i(x) \in OP(T_i)$, onde T_i é uma transação móvel executada a partir de um cliente móvel CM_k, só será realmente efetuada quando a mensagem (pacote) que contém a operação $w_i(x)$, enviada pelo cliente CM_k, chegar ao servidor. Pois, somente neste instante é que efeitos da operação $w_i(x)$ estarão visíveis para as demais transações. Desta forma, o marcador de tempo para as operação de atualização pertencentes a transações móveis pode ser definido como a seguir: Seja $C(w_i(x))$ o valor do marcador de tempo para a operação $w_i(x) \in OP(T_i)$, onde T_i é uma transação móvel executada a partir de um cliente móvel CM_k, $C(w_i(x))$ é o número do ciclo corrente quando a mensagem que contem $w_i(x)$, enviada pelo cliente CM_k, chega ao servidor.

Cenário Exemplo

Para ilustrar a utilização desta estratégia, vamos tomar como exemplo uma aplicação onde a quantidade de produtos em estoque é atualizada por vendedores equipados com computadores móveis distribuídos em diferentes pontos de venda.

Exemplo1. Considere o seguinte conjunto de transações, as quais lêem e atualizam a quantidade dos produtos em estoque:

$T_1 : r_1(P_1)\, w_1(P_1 - 10)\, C_1$

$T_2 : r_2(P_1)\, r_2(P_2)\, C_2$

$T_3 : r_3(P_1)\, w_3(P_1 + 10)\, C_3$

Vamos assumir que inicialmente existem dez unidades de cada produto em estoque.

Primeiramente, considere que a transação T_3 é executada no servidor. Por outro lado, a transação T_1 é executada no cliente A, enquanto a transação T_2 no cliente B. Considere agora o *schedule* global SG_U apresentado na figura 5.10.

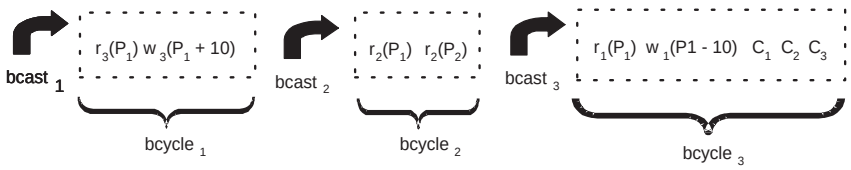

Figura 5.10: Schedule SG_U.

Vamos assumir que no cenário de execução apresentado na figura 5.10 os pacotes com as informações das operações executadas pelas transações clientes durante o $bcycle_n$ chegam ao servidor antes do envio do $bcast_{n+1}$.

Podemos observar facilmente que utilizando esta estratégia não teremos problemas com relação a inversão de arestas, pois as operações de atualização das transações móveis são tratadas como se pertencessem a transações do servidor.

Contudo, é importante investigar o problema de "atualização perdida", o qual pode gerar estados inconsistentes. Observe na figura 5.10 que tanto a operação $r_3(P_1)$ quanto a operação $r_1(P_1)$ lêem o valor 10 para o item P_1. Logo, a operação $w_3(P_1 + 10)$ irá escrever o valor 20, já a operação $w_1(P_1 - 10)$ irá escrever o valor 0 (zero). Assim, o valor do item P_1 após a execução do *schedule* será igual a 0 (zero). Contudo, se o item P_1 tinha inicialmente valor igual a 10 e foi efetuada uma adição de 10 unidades seguida de uma subtração de 10 unidades, o item P_1 deveria continuar com valor igual 10. Desta forma, a execução deste *schedule* levará o banco de dados a um estado inconsistente. Logicamente, esta execução deverá ser evitada (considerada incorreta).

A figura 5.11 mostra o grafo gerado pelo algoritmo para o *schedule* SG_U (figura 5.10). Como podemos observar, o grafo apresenta um ciclo $T_1 \rightarrow T_3 \rightarrow T_1$. Logo o *schedule* SG_U será considerado incorreto.

Torna-se fácil verificar que neste caso, quando temos duas transações T_i e T_j que possuem tanto operações de leitura quanto operações de escrita sobre um mesmo item, o algoritmo sempre identificará um ciclo entre T_i e T_j, não permitindo assim que o problema de "atualização perdida" ocorra.

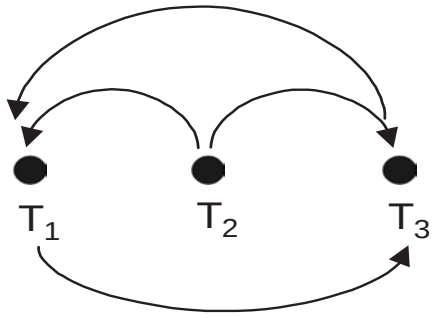

Figura 5.11: Grafo de Serialização Temporal para o Schedule SG_U.

Desta forma, mostramos que o protocolo TGST funciona corretamente na presença de operações de escrita nos clientes móveis. Contudo, no cenário analisado temos que uma transação sempre lê o valor de um item antes de atualizá-lo. Logo, ainda é necessário observar o comportamento do protocolo na presença de "atualizações cegas".

Exemplo2. Considere o seguinte conjunto de transações, as quais lêem e atualizam a quantidade dos produtos em estoque:

$T_1 : r_1(P_1)\, w_1(P_1 + 20)\, C_1$

$T_2 : r_2(P_1)\, r_2(P_2)\, C_2$

$T_3 : w_3(P_1, 5)\, C_3$

Vamos assumir que inicialmente existem dez unidades de cada produto em estoque.

Primeiramente, considere que a transação T_3 é executada no servidor. Por outro lado, a transação T_1 é executada no cliente A, enquanto a transação T_2 no cliente B. Considere agora o *schedule* global SG_C apresentado na figura 5.12.

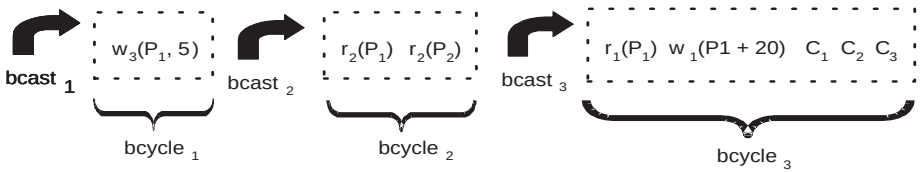

Figura 5.12: Schedule SG_C.

No cenário de execução apresentado na figura 5.12, vamos assumir que os pacotes com as informações das operações executadas pelas transações clientes durante o $bcycle_n$ chegam ao servidor antes do envio do $bcast_{n+1}$.

Observe que caso as transações fossem executadas serialmente os possíveis valores resultantes para o item P_1 seriam: 5 (T_1, T_3) ou 25 (T_3, T_1). Entretanto, como $r_1(P_1)$ lê um valor anterior ao valor gerado pela operação $w_3(P_1, 5)$, o valor para o item P_1 após a execução do *schedule* SG_C (figura 5.12) será 30. Logo, esta execução levará o banco de dados a um estado inconsistente. Logicamente, este fenômeno deve ser evitado.

A figura 5.13 mostra o grafo gerado pelo protocolo para o *schedule* SG_C (figura 5.12) . Como podemos observar, o grafo apresenta um ciclo $T_1 \rightarrow T_3 \rightarrow T_1$. Logo o *schedule* SG_C será considerado incorreto.

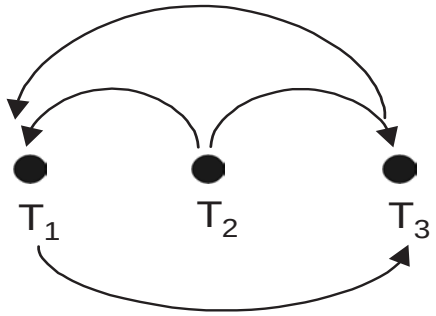

Figura 5.13: Grafo de Serialização Temporal para o Schedule SG_C.

Até aqui, foi analisado o funcionamento do protocolo assumindo-se que os pacotes com as informações das operações executadas pelas transações clientes durante o $bcycle_n$ chegam ao servidor antes do envio do $bcast_{n+1}$. Entretanto, para comprovarmos a corretude e a real eficiência do protocolo proposto, precisamos considerar a ocorrência de falhas e atrasos na comunicação.

Considere novamente o exemplo 1. Considere agora que a transação T_3 é uma transação móvel e que, por um motivo qualquer, o pacote contendo a informação da operação $w_3(P_1 + 10)$ executada pela transação T_3 durante o $bcycle_1$ atrase. Neste caso, o servidor irá ver o *schedule* SG'_U mostrado na figura 5.14.

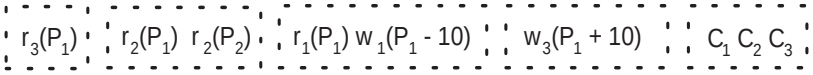

Figura 5.14: Schedule SG'_U. Schedule SG_U com informações atrasadas.

O grafo de serialização para o schedule SG'_U é mostrado na figura 5.15.

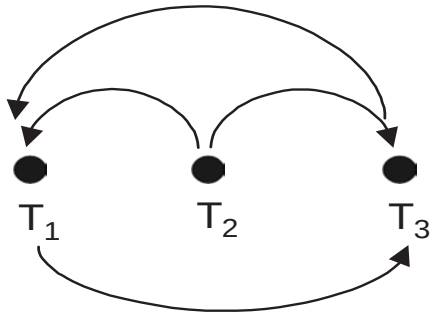

Figura 5.15: Grafo de Serialização para o Schedule SG'_U.

Para o cenário exemplo apresentado acima, pode-se facilmente ver que o grafo produzido pelo protocolo corresponde ao grafo de serialização correto para o schedule SG'_U. Assim, o protocolo proposto irá garantir que, mesmo que a informação da operação $w_3(P_1 + 10)$ atrase, o escalonador identifique um ciclo no grafo e evite o problema da "atualização atrasada".

Voltemos agora ao exemplo 2. Suponha que, por um motivo qualquer (problemas nas linhas de comunicação, por exemplo), o pacote contendo a informação da operação $w_3(P_1, 5)$ executada pela transação T_3 durante o $bcycle_1$ atrase. Neste caso, o servidor irá ver o schedule SG'_C mostrado na figura 5.16.

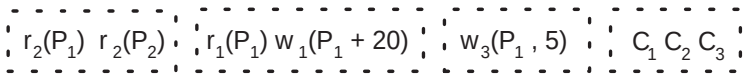

Figura 5.16: Schedule SG'_C. Schedule SG_C com informações atrasadas.

Observe que neste caso, a execução do *schedule* SG'_C levará o banco de dados a

um estado consistente, pois corresponde a uma execução serial (T_2, T_1, T_3), onde o item P_1 terá valor 5.

O grafo de serialização para o *schedule* SG'_C é mostrado na figura 5.17.

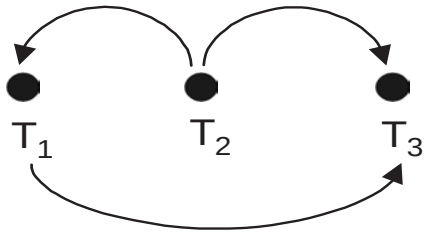

Figura 5.17: Grafo de Serialização para o Schedule SG'_C.

Para o cenário exemplo apresentado acima, pode-se facilmente ver que o grafo produzido pelo protocolo corresponde ao grafo de serialização correto para o *schedule* SG'_C.

5.4 Teste do Grafo de Serialização Temporal-Semântico

Nesta seção, investigaremos como o protocolo do teste do grafo de serialização temporal (TGST) pode ser estendido a fim de utilizar como critério de corretude a serializabilidade semântica proposta inicialmente em [8]. Este critério será examinado com a finalidade de introduzir um maior grau de concorrência entre as transações, diminuindo o número de transações abortadas e aumentando a eficiência do protocolo TGST.

A serializabilidade semântica, proposta em [8], baseia-se na utilização de informações semânticas sobre os objetos do banco de dados (e não sobre as transações). A idéia principal consiste em proporcionar diferentes visões de atomicidade para cada transação e, por esta razão, permitir um maior entrelaçamento entre as transações.

Isto é alcançado através da aceitação de *schedules* que não são serializáveis, mas que preservam a consistência do banco de dados.

Desta forma, acreditamos que o protocolo TGST terá ganhos em eficiência com a utilização da seriabilidade semântica, pois, uma vez que o grau de entrelaçamento entre as transações aumenta o número de transações abortadas diminui, o que é de grande importância se considerarmos que as transações em ambientes de computação móvel são de longa duração.

5.4.1 O Protocolo TGSTSe

Em contraste com o protocolo TGST, o TGSTSe explora informações semânticas proporcionadas pelo conceito de unidade semântica (ver seção 4.9).

Para cada item de dado x pertencente ao banco de dados, o servidor de *broadcast* envia, juntamente com o objeto x, o seguinte marcador de tempo: o número do ciclo que a transação T_i executou sua operação de *commit*, se $w_i(x) \in OP(T_i)$ e T_i é a última transação que executou uma operação de escrita sobre x. Quando uma transação móvel T_k executar uma operação de leitura $r_k(x)$, esse marcador de tempo será associado a $r_k(x)$. Por outro lado, o marcador de tempo para uma operação de escrita $w_k(x)$ pertencente a transação móvel T_k será o número do ciclo corrente quando o pacote (mensagem) que contém a operação $w_k(x)$, que foi enviado pelo cliente móvel CM_k, chegar ao servidor. Para as operações das transações executadas no servidor o marcador de tempo continua sendo o número do ciclo onde a operação é realizada.

Com base no marcador de tempo descrito no parágrafo anterior, o grafo de serialização temporal é construído como descrito a seguir:

Passo 1. Para cada operação $p_i(x) \in OP(T_i)$ que o escalonador recebe, ele checa se existe uma operação $q_j(x) \in OP(T_j)$ que conflita com p_i(x) e que já foi escalonada. Caso q_j(x) exista, uma aresta será inserida entre T_i e T_j ($T_j \overset{SU_N}{\rightarrow} T_i$ ou $T_i \overset{SU_N}{\rightarrow} T_j$) , onde x é um objeto da unidade semântica SU_N. Para que esta

aresta seja incluída corretamente deveremos considerar dois casos distintos, os quais descrevemos a seguir:

Caso 1. T_i é uma transação executada no servidor. Neste caso, será executada uma verificação temporal da seguinte forma:

Se $C(q_j(x)) \leq C(p_i(x))$
  Então o escalonador insere uma aresta da forma $T_j \overset{SU_N}{\rightarrow} T_i$.
Se não
  uma aresta da forma $T_i \overset{SU_N}{\rightarrow} T_j$ será inserida no grafo.

Caso 2. T_i é uma transação móvel. Neste caso, será executada uma verificação temporal da seguinte forma:

Se $C(q_j(x)) < C(p_i(x))$
  Então o escalonador insere uma aresta da forma $T_j \overset{SU_N}{\rightarrow} T_i$
Se não
  Se $C(q_j(x)) > C(p_i(x))$
    Então o escalonador insere uma aresta da forma $T_i \overset{SU_N}{\rightarrow} T_j$
  Se não
    Se T_j já executou o *commit*
      Então o escalonador insere uma aresta da forma $T_j \overset{SU_N}{\rightarrow} T_i$
    Se não o escalonador insere uma aresta da forma $T_i \overset{SU_N}{\rightarrow} T_j$

Passo 2. O escalonador verifica se a nova aresta introduz um ciclo no grafo de serialização temporal. Em caso afirmativo, o escalonador rejeita a operação $p_i(x)$, desfaz o efeito das operações de $\Pi_{[OP_{SU_N}]}\, T_i$ e remove a aresta inserida. Caso contrário, $p_i(x)$ é aceita e escalonada. Observe que, quando o escalonador identifica um ciclo no grafo, somente as operações pertencentes a unidade semântica envolvida no ciclo serão desfeitas. Isto é, não é necessário abortar a transação por completo. Isto reduz o processamento para desfazer operações como também preserva parte do trabalho já realizado pela transação.

Capítulo 6

Conclusões

O modo de disseminação de dados baseado em difusão (*broadcast*) está se transformando no principal modo de transmissão de informações em ambientes de computação móvel e comunicação sem fio. Neste cenário, as aplicações necessitam ler dados atuais e consistentes apesar das atualizações que podem ocorrer no servidor ou até mesmo nos clientes móveis. Contudo, devido às características destes ambientes, garantir leituras atuais, bem como a consistência dos dados, torna-se uma tarefa complexa.

Com o objetivo de solucionar eficientemente o problema do controle de concorrência em ambientes de *broadcast*, várias proposta têm sido apresentadas. Porém, a maioria dessas propostas requer que estruturas de controle complexas sejam enviadas aos clientes móveis. Assim, os clientes têm que permanecer por mais tempo em seu estado ativo para gerenciar estas estruturas, além de terem que armazená-las localmente.

Neste trabalho, apresentamos duas estratégias distintas para solucionar o problema do controle de concorrência em ambientes de broadcast. A primeira estratégia consiste no desenvolvimento de um novo protocolo para controle de concorrência em ambientes de broadcast baseado em serializabilidade. Neste sentido, desenvolvemos um novo protocolo, denominado de teste do grafo de serialização temporal (TGST), o qual explora informações temporais referentes ao momento em

que um objeto do banco de dados foi lido ou atualizado. O protocolo proposto evita que estruturas complexas, como as propostas em [31] e [32], sejam enviadas aos cliente móveis. Além disso, os clientes não necessitam gerenciar ou manipular essas estruturas para realizar suas transações. Portanto, o protocolo TGST reduz o tráfego de comunicação entre o servidor e os clientes, minimiza o tempo em que o cliente necessita escutar o canal de *broadcast* e os clientes só precisam armazenar os valores dos objetos de seu real interesse. Dessa forma, o protocolo proposto garante uma economia no custo de utilização dos canais de comunicação, na limitada energia dos computadores portáteis e nos seus escassos recursos de memória. A segunda estratégia consiste na utilização de um novo modelo para o processamento de transações que seja mais adequado a ambientes de broadcast. Como este objetivo propomos uma extensão ao protocolo TGST a fim de utilizar como critério de corretude a serializabilidade semântica proposta por [5]. Este critério foi utilizado com a finalidade de introduzir um maior grau de concorrência entre as transações, diminuindo o número de transações abortadas e aumentando a eficiência do protocolo TGST.

Outras contribuições importantes deste trabalho foram a investigação e identificação das principais propostas para o controle de concorrência em ambientes de *broadcast* e a implementação, em linguagem Java, do protocolo TGST.

Como trabalhos futuros, iremos realizar testes para medir e comparar as diversas estratégias para controle de concorrência em ambientes de *broadcast* e investigar como o protocolo TGST pode ser estendido a fim ser utilizado em ambientes de computação móvel com disseminação de dados baseada em requisição-resposta (*pull-based*).

Referências Bibliográficas

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. *Broadcast Disks: Data Management for Asymmetric Communications Environments.* Proceedings of the ACM SIGMOD Conference, 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. *Disseminating Updates on Broadcast Disks.* Proceedings of the 22 nd VLDB Conference, 1996.

[3] R. Alonso and H. F. Korth. *Database Systems Issues in Nomadic Computing.* Relatório Técnico do MITL, 1992.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Dabase Systems.* Addison-Wesley, 1987.

[5] A. Brayner. *Transaction Management in Multidatabase Systems.* Shaker Verlag, 1999.

[6] A. Brayner. *Lock Downgrading: An Approach to Increase Concurrency in Advanced Database Applications.* Database and Expert Systems, 2001.

[7] A. Brayner and T. Härder. *Global Semantic Serializability: An Approach to Increase Concurrency in Multidatabase Systems.* 6o Conference on Cooperative Information Systems(CoopIS 2001), 2001.

[8] A. Brayner, T. Härder, and N. Ritter. *Semantic Serializability: A Correctness Criterion for Processing Transactions in Advanced Database Applications.* DATA AND KNOWLEDGE ENGINEERING, 31, 1999.

[9] A. Brayner and J. M. Monteiro. *Temporal Serialization Graph Testing: An Approach to Control Concurrency in Broadcast Environments.* 15o Brazilian Symposium on Database Systems, 2000.

[10] A. Brayner and J. M. Monteiro. *Teste do Grafo de Serialização Temporal: Uma Abordagem para o Controle de Concorrência em Ambientes de Broadcast.* I Encontro de Pós-Graduação e Pesquisa da Universidade de Fortaleza, 2001.

[11] M. A. Casanova. *The Concurrency Problem of Database Systems.* In Lectures Notes in Computer Science, 116, 1981.

[12] P. K. Chrysanthis. *Transaction Processing in Mobile Computing Environment.* IEEE Workshop on Advances in Parallel and Distributed Systems, 1993.

[13] A. Datta, D. E. Vandermeer, A. Celik, and V. Kumar. *Broadcast Protocols to Support Eficcient Retrieval from Databases by Mobile Users.* ACM TODS, 24 (1), 1999.

[14] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. *The Notions of Consistency and Predicate Locks in a Database System.* Communications of the ACM, 9 (11), 1976.

[15] J. N. Gray. *The Transaction Concept: Virtues and Limitations.* In Proceedings of the 7th International Conference on VLDB, pages 144-154, 1981.

[16] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Database.* In M. Stonebraker, editor, Readings in Database Systems, pages 94-121. Morgan-Kaufmann, 1988.

[17] V. Hadzilacos. *A Theory of Reliability in Database Systems.* Journal of the ACM, 25:121-145, 1988.

[18] G. Herman. *The Datacycle Architecture for Very Hight Throughput Database Systems.* Proceedings of the ACM SIGMOD Conference, 1987.

[19] T. Härder. *Observations on Optimistic Concurrency Control Schemes.* Informations Systems, 9(2):111-120, 1984.

[20] T. Härder and A. Reuter. *Concepts for Implementing a Centralized Database Management System.* In Proceedings of the International Computing Symposium, 1983.

[21] T. Imielinski and B. R. Badrinath. *Data Management for Mobile Computing.* Communications of the ACM, 1993.

[22] T. Imielinski and B. R. Badrinath. *Mobile Wireless Computing: Solutions and Challenges in Data Management.* Relatório Técnico Rutgers University, 1993.

[23] T. Imielinski and B. R. Badrinath. *Mobile Wireless Computing: Challenges in Data Management.* Communications of the ACM, 1994.

[24] S. Kumar, E. Kwang, and D. Agrawal. *Capera An Activity Framework for Transaction Processing on Wide-Area Networks.* Proceedings of the 23 nd VLDB Conference, 1997.

[25] G. R. Mateus and A. A. F. Loureiro. *Introdução à Computação Móvel.* 11a Escola de Computação, 1998.

[26] B. Oki. *The Information Bus - A Architecture for Extensible Distributed Systems.* Proceedings of the SOSP Conference, 1993.

[27] C. H. Papadimitriou. *The Theory of database Concurrency Control.* Computer Science Press, 1986.

[28] W. Paper. *Web Page of Airmedia inc.* http:// www.airmedia.com, 2000.

[29] W. Paper. *Web Page of Directpc inc.* http:// www.directpc.com, 2000.

[30] W. Paper. *Web Page of Vitria Technology inc.* http://www.vitria.com, 2000.

[31] E. Pitoura and P. Chrysanthis. *Scalable Processing of Read-Only Transactions in Broadcast Push.* IEEE International Conference on Distributed Computing Systems, 1999.

[32] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. *Efficient Concurrency Control for Broadcast Environments.* Proceedings of the ACM SIGMOD Conference, 1999.

[33] S. Shekar and D. Liu. *Genesis and Advanced Traeler Informations Systems (ATIS): Killer Applications for Mobile Computing.* MOBIDATA Workshop, 1994.

[34] S. Sheng, A. Chandrasekaran, and R. W. Broderson. *A Portable Multimedia Terminal for Personal Communications.* IEEE Communications Magazine, Dezembro 1992.

[35] R. Tewari and P. Grillo. *Data Management for Mobile Computing on the Internet.* Communications of the ACM, 1995.

Índice Remissivo