Universidade Federal do Ceará
Centro de Tecnologia
Pós Graduação em Engenharia de Teleinformática

UMA ESTRATÉGIA PARA O GERENCIAMENTO DA REPLICAÇÃO PARCIAL DE DADOS XML

ÉRIKO JOAQUIM ROGÉRIO MOREIRA

DISSERTAÇÃO DE MESTRADO

Fortaleza
2009

Universidade Federal do Ceará
Centro de Tecnologia

ÉRIKO JOAQUIM ROGÉRIO MOREIRA

UMA ESTRATÉGIA PARA O GERENCIAMENTO DA REPLICAÇÃO PARCIAL DE DADOS XML

Dissertação submetida à Coordenação do Programa de Pós Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Mestre em Engenharia de Teleinformática.

Orientador: Prof. Javam de Castro Machado, D.Sc.

Fortaleza
2009

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

M836e Moreira, Ériko Joaquim Rogério.
Uma estratégia para o gerenciamento da replicação parcial de dados XML / Ériko Joaquim Rogério Moreira – 2009.
91 f. : il. color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Tecnologia, Departamento de Engenharia de Teleinformática, Programa de Pós-Graduação em Engenharia de Teleinformática, Fortaleza, 2009.
Área de Concentração: Sinais e Sistemas.
Orientação: Prof. Dr. Javam de Castro Machado.

1. Linguagem de programação - XML 2. Banco de dados. 3. Fragmentação de dados I. Título.

CDD 621.38

UMA ESTRATÉGIA PARA O GERENCIAMENTO DA REPLICAÇÃO PARCIAL DE DADOS XML

ÉRIKO JOAQUIM ROGÉRIO MOREIRA

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre.

Prof. Javam de Castro Machado, D.Sc.
Universidade Federal do Ceará

Prof. José Neuman de Souza, D.Sc.
Universidade Federal do Ceará

Prof. Giovanni Cordeiro Barroso, D.Sc.
Universidade Federal do Ceará

Prof. Sérgio Lifschitz, D.Sc.
Pontifícia Universidade Católica do Rio de Janeiro

Aprovada em 04 de Setembro de 2009

A minha genitora, Valdeni Rogério Moreira, pessoa que mais incentivou a minha carreira acadêmica, além de ser uma excelente mãe e minha melhor amiga.

AGRADECIMENTOS

A Deus, por ter me iluminado nas escolhas que fiz, e me confortado nos momentos em que minha vontade já não era tão determinante.

Ao professor Javam de Castro Machado, pelo companheirismo e conhecimento compactuados durante a orientação, os quais foram fundamentais à conclusão deste trabalho. Não obstante, encontro-me grato pela confiança e coragem desprendida a mim.

Aos professores José Neuman de Sousa, Giovanni Cordeiro Barroso e Sérgio Lifschitz pelas contribuições irrelevantes ao trabalho.

Ao Prof. M.Sc. Flávio Sousa, pelas sugestões aferidas em todos os aspectos do trabalho. No mais, agradeço pelo apoio proporcionado nos momentos difíceis que enfrentei.

Aos meus pais e irmãos, que sempre me incentivaram, apoiaram e compartilharem comigo momentos alegres e difíceis, direta e indiretamente, representando uma das bases da minha vida.

Aos amigos do Mestrado, pela excelente relação simbiótica que criamos. Em especial agradeço a Máiquel Sampaio, Simão Gurgel, Juliana Magalhães, Ney Mello, Lincoln Rocha, Leonardo Moreira, Luana Pires, Marcos Dantas e Diana Braga.

Aos amigos do NPD, por me ajudarem sempre. Em especial, agradeço a Hermes Abreu, Sandra Rodrigues, Marcos Camurça, Cláudia Castelo, Hannelore Brandenburg, Vera Juvêncio, Juliana Maria e Maria Aline, pela atenção especial prestada.

Aos amigos do BNB, em especial aos gerentes Osmar Pimentel e Márcio Maia, pelo apoio e compreensão que me foram concedidos.

Aos amigos Diego Rebouças, Ademar Carneiro, Leigo Gomes, Anchieta Guerreiro, Joaquim Guerreiro, Zaira Freitas e Helena Maurício. A amizade justifica os agradecimentos.

A todos que, direta ou indiretamente, contribuíram para a execução deste trabalho.

Não há problema que não possa ser solucionado pela paciência.

—CHICO XAVIER

RESUMO

XML tornou-se um padrão amplamente utilizado na representação e troca de dados entre aplicações na Web. Com isso, um grande volume desses dados está distribuído na Web e armazenado em diversos meios de persistência. SGBDs relacionais que suportam XML fornecem técnicas de controle de concorrência para gerenciar esses dados. No entanto, a estrutura de dados XML dificulta a aplicação dessas técnicas.

Adicionalmente, as técnicas de replicação têm sido utilizadas para melhorar o gerenciamento de grandes quantidades de dados XML. Pesquisas atuais de replicação de dados XML consistem em adaptar os conceitos existentes ao modelo semi-estruturado. Em especial, a replicação total apresenta uma grande quantidade de bloqueios, em decorrência das atualizações ocorrerem em todas as cópias da base. Por outro lado, a replicação parcial visa aumentar a concorrência entre as transações, com uma menor quantidade de bloqueios em relação à replicação total.

Este trabalho apresenta o RepliXP, uma estratégia para o gerenciamento da replicação parcial de dados XML. Ele é apresentado como um mecanismo que combina características de protocolos de replicação síncronos e assíncronos para diminuir o número de bloqueios de atualização. Para validar a estratégia, foram realizados testes de desempenho analisando o tempo de resposta das transações. Foram comparadas as abordagens de replicação total e replicação parcial no RepliXP. De acordo com os resultados obtidos, o RepliXP utilizando a estratégia de replicação parcial de dados XML proporcionou uma melhoria no tempo de resposta das transações concorrentes.

Palavras-chave: Sistemas de Banco de Dados, XML, Replicação Parcial, Fragmentação de Dados, Controle de Concorrência

ABSTRACT

XML has become a widely used standard in representing and exchanging data among Web Applications. Consequently, a large amount of data is distributed on the Web and stored in several persistence medias. Relational DBMSs XML-enabled provide concurrency control techniques to manage such data. However, XML data structure makes it difficult implementation of these techniques.

Additionally, replication techniques have been used to improve management of large amounts of XML data. Current researches of XML data replication consist of to adapt existing concepts to semi-structured model. In particular, full replication provides a large of locks, due to updates that have occurred on all copies of the base. Moreover, the partial replication aims to increase concurrency among transactions, with a smaller amount of blocks in relation to total replication.

This work presents the RepliXP, a approach for management of partial XML data replication. It is presented as a mechanism that combines features of synchronous and asynchronous replication protocols to reduce the amount of update locks. In order to evaluate the strategy, performance tests were carried out by analyzing the response time of transactions. Full and partial replication approaches were compared in RepliXP. According to the results, RepliXP using the strategy of partial XML data replication provided an improvement in response time of concurrent transactions.

Keywords: Database Systems, XML, Parcial Replication, Fragmentation Data, Concurrency Control

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Esta dissertação apresenta uma estratégia para o gerenciamento da replicação parcial de dados XML. Seu objetivo é proporcionar melhorias no desempenho dos sistemas de gerenciamento de banco de dados xml nativo (SGBDXN), diminuindo o tempo de resposta das transações. Esta melhoria no tempo de resposta advém do aumento no número de consultas concorrentes.

Neste capítulo, são apresentadas a motivação e a justificativa para o desenvolvimento deste trabalho, assim como os objetivos e as contribuições propostas. Ao final do capítulo, será descrito como está organizado o restante desta dissertação.

1.1 MOTIVAÇÃO E CARACTERIZAÇÃO DO PROBLEMA

XML [1] e padrões relacionados têm sido amplamente adotados como forma de representação e troca de dados. Esta adoção ocorre, principalmente, devido à simplicidade e à expressividade do modelo semi-estruturado. Por conta do volume de informações manipulado nesse formato, surgiu a necessidade de armazenar documentos XML de forma eficiente, utilizando Sistemas de Gerenciamento de Bancos de Dados (SGBD).

Com isso, alguns SGBDs tradicionais, tais como relacionais e orientados a objetos, passaram a dar suporte ao armazenamento e manipulação de dados XML, permitindo seu acesso por linguagens de consulta específicas, como o XPath [2] e XQuery [3]. Estes ficaram conhecidos como SGBDs *XML-Enabled* [4, 5]. Posteriormente, foram desenvolvidos SGBDs que manipulam dados XML de forma nativa, armazenando documentos nesse formato segundo uma estrutura lógica de árvore ou grafo, onde os nós representam elementos e atributos e as arestas definem os relacionamentos elemento/sub-elemento ou elemento/atributo [6]. Estes sistemas são denominados SGBD XML Nativos (SGBDXN) [7, 8, 9, 10, 11, 12].

No entanto, muitos fatores influenciam no desempenho de SGBDs XML. Em par-

ticular, o tamanho da base de dados pode ter um impacto significante na recuperação de dados XML. Isso acontece porque as ferramentas típicas para consulta sobre esses dados necessitam verificar se os documentos estão válidos, procedimento chamado de *parsing*, e que consiste em checar se os documentos XML estão sintática e semanticamente corretos. Outra característica que afeta o desempenho desses SGBDs é o custo de interpretação das linguagens de consulta XML, as quais são ferramentas poderosas e requerem algoritmos sofisticados para sua interpretação e execução.

Adicionalmente, as técnicas de replicação de dados têm sido utilizadas para aumentar o desempenho e a disponibilidade em SGBDs relacionais [13] e orientados a objetos [14]. Estes ganhos advêm da possibilidade de se executar consultas em um subconjunto dos dados, reduzindo assim o tempo de processamento. Além disso, estas soluções permitem que os dados da base possam ser acessados de forma concorrente por dois ou mais clientes. Um benefício secundário da replicação de dados é que as consultas podem ser executadas de forma distribuída, diminuindo seu tempo de execução.

Todavia, a flexibilidade dos dados XML impõe novos desafios, implicando na necessidade de novas técnicas de replicação. Pesquisas atuais de replicação de dados XML consistem em adaptar os conceitos existentes ao modelo semi-estruturado [7, 10, 11, 15]. Dentre estes trabalhos, o RepliX [15] aborda, de forma satisfatória, a maioria das questões relativas à replicação de dados XML, apresentando resultados e avaliando aspectos de desempenho, disponibilidade e escalabilidade. No entanto, a replicação total apresenta um grande número de bloqueios em decorrência da atualização dos dados. Isto acontece porque as alterações dos dados devem ser aplicadas a todas as cópias da base.

Por outro lado, a replicação parcial visa aumentar a concorrência entre as transações, com uma menor quantidade de bloqueios em relação à replicação total. As técnicas de fragmentação aplicadas na replicação parcial podem trazer ganhos de desempenho no processamento de consultas, principalmente sobre grandes volumes de dados [13]. Até o momento de conclusão deste trabalho, não foi encontrado na literatura nenhuma proposta para replicação parcial de dados XML.

1.2 OBJETIVO E CONTRIBUIÇÃO

Visando prover o gerenciamento eficaz da replicação parcial de dados XML, esta dissertação apresenta o RepliXP como um mecanismo para aumentar o desempenho em

SGBDXNs. O mecanismo serve-se da adaptação do RepliXP que aumenta a concorrência entre as transações. A solução adota a estratégia de fragmentação horizontal sobre coleções de documentos XML, diminuindo a granularidade dos bloqueios e aumentando a concorrência entre as transações.

A principal contribuição desta dissertação consiste da estratégia para o gerenciamento da replicação parcial de dados XML. Ela combina características de protocolos de replicação síncronos e assíncronos para diminuir o número de bloqueios nas transações concorrentes. A fim de avaliar a proposta, este trabalho estendeu o benchmark proposto por [16], o que constituiu contribuição complementar.

1.3 ESTRUTURA DA DISSERTAÇÃO

Os próximos capítulos deste ensaio estão estruturados da seguinte forma:

- Capítulo 2 - apresenta os conceitos relativos ao gerenciamento de dados XML, abordando assuntos básicos de XML, meios de armazenamento, protocolos de replicação e os trabalhos relacionados a presente investigação.
- Capítulo 3 - descreve em detalhes o RepliXP. Características, especificação, cenários de execução e algoritmos desenvolvidos são detalhados.
- Capítulo 4 - serão mostrados aspectos de implementação e a avaliação realizada no trabalho.
- Capítulo 5 - apresenta as considerações finais da pesquisa, incluindo trabalhos futuros.

CAPÍTULO 2

FUNDAMENTOS TEÓRICOS

Replicação é um tópico cada vez mais importante no contexto de SGBDs, sendo um dos fatores responsáveis pelo aumento no desempenho e na disponibilidade desses sistemas. Em um ambiente de banco de dados, a replicação consiste em criar cópias de uma base, as quais são chamadas de réplicas. As réplicas podem ser gerenciadas por um único servidor ou distribuídas dentre dois ou mais servidores. Denominamos por sítio um servidor que gerencia uma ou mais réplicas de uma base de dados. A replicação proporciona um aumento no desempenho por meio do acesso concorrente de leitura das réplicas. A base de dados torna-se mais disponível, uma vez que a indisponibilidade da base armazenada em um sítio pode ser suprida pelo acesso de uma réplica armazenada em outro sítio.

Embora o conceito de replicação seja um tanto quanto intuitivo, sua utilização induz ao problema de manutenção das réplicas. Quando um dado é alterado, as réplicas que possuem aquela informação precisam ser atualizadas, para que se mantenha um estado consistente da base distribuída [17]. Esta premissa corresponde a garantir o critério de serializabilidade (ou one-copy serializability) [18]. A manutenção da consistência do estado distribuído requer protocolos específicos para proporcionar a manutenção das réplicas da base de dados, os quais são denominados protocolos de replicação.

Segundo Guerraoui e Schiper [19], para satisfazer esse critério, é necessário que as operações de atualização concorrentes, enviadas por diferentes clientes, sejam executadas na mesma ordem em todas as réplicas. Além disso, cada operação de atualização deve ser executada em todas as réplicas de forma atômica.

Outro aspecto associado à replicação consiste na granularidade dos dados copiados, que pode ser total ou parcial. A replicação total corresponde ao caso em que cada informação da base original é copiada em todos os sítios. Na replicação parcial, apenas uma parte da base está copiada em todos os sítios. Em particular, vários aspectos de distribuição de dados estão inseridos na replicação parcial. A fragmentação consiste em particionar a base de dados em subconjuntos seguindo alguma heurística, tendo como

resultado um conjunto de fragmentos. Além do mais, é necessário determinar uma maneira como os fragmentos (e suas cópias) são alocados nas réplicas existentes, processo conhecido como alocação de dados. O processamento de consultas também sofre modificações, uma vez que uma consulta pode acessar dados que estão dispostos em réplicas distribuídas em vários sítios. Deve existir, portanto, uma estratégia para localização dos dados e decomposição das consultas, quando necessária.

2.1 PROTOCOLOS DE REPLICAÇÃO

Um protocolo de replicação define uma estratégia para garantir a consistência dos dados em um ambiente replicado. Na literatura, os protocolos de cópia primária e réplicas ativas são amplamente difundidos. Normalmente, é utilizada a estratégia lê somente uma e atualiza todas as réplicas (*read one write all* - ROWA). Nessa estratégia, a atualização de dados é uma operação distribuída e deve acontecer para todas as réplicas, ao passo que a leitura é uma operação isolada e envolve apenas uma réplica.

Uma transação é uma sequência de operações sobre itens de dados que devem ser executadas, de forma a satisfazer as propriedades ACID (atomicidade, consistência, isolamento e durabilidade) [20]. A transação se inicia quando é executada a operação (*begin*). Após este comando, são executadas uma ou mais operações sob os dados, que podem ser de consulta (*read*) ou inclusão/atualização (*write*). Ao final da execução destas operações, é disparada uma instrução de finalização da transação (*commit*), que implica na consolidação das alterações. Caso alguma operação não possa ser realizada, a transação é abortada por meio do comando (*abort*) e todas as alterações realizadas durante a execução são descartadas. Classificam-se como transações de leitura aquelas que contêm apenas operações de consulta, ao passo que uma transação de atualização é aquela em que contém pelo menos uma operação de atualização.

2.1.1 Cópia Primária

Neste protocolo, é definida uma réplica da base que sempre terá sua consistência garantida, a qual é chamada de réplica primária. O sítio no qual a réplica primária está armazenada é chamado sítio primário. As demais réplicas são chamadas de réplicas secundárias. De forma análoga ao sítio primário, os sítios secundários são aqueles em que estão armazenadas as réplicas secundárias. Os clientes estabelecem comunicação apenas

com o sítio primário. Para cada operação requisitada, o sítio primário gerencia as réplicas secundárias e envia a resposta para o cliente.

Todas as operações solicitadas pelos clientes são encaminhadas ao sítio primário, que coordena a execução de cada uma delas. Nas operações de leitura, o sítio primário envia uma resposta ao cliente sem consultar os demais sítios. Nas operações de atualização, o sítio primário propaga a solicitação do cliente para os sítios secundários, os quais executam a consulta sob cada réplica secundária local. Posteriormente, o sítio primário verifica se todas as réplicas secundárias estão atualizadas. Uma vez concluída a verificação com sucesso, a réplica principal é atualizada e a resposta da atualização é retornada ao cliente.

Caso o sítio primário verifique que a atualização não foi executada corretamente em alguma réplica secundária, o sítio secundário que armazena esta réplica é descartado do sistema distribuído. Caso ocorra uma falha ao aplicar a atualização na réplica primária, a operação é abortada e o erro é encaminhado ao cliente. Se houver indisponibilidade do sítio primário, um novo sítio deve ser escolhido dentre os secundários. A Figura 2.1 ilustra a troca de mensagens entre as réplicas.

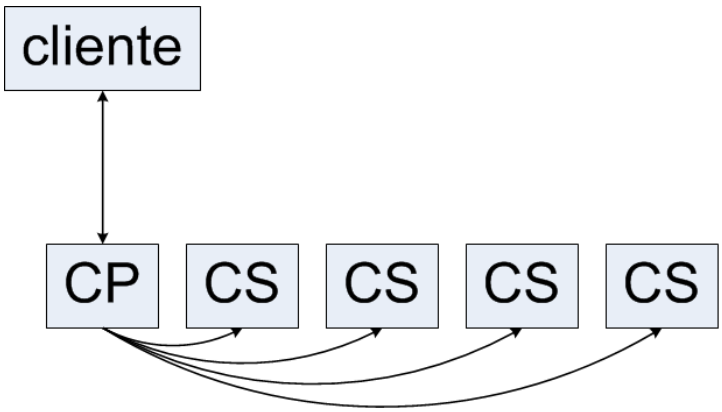


Figura 2.1 Protocolo de Replicação com Cópia Primária

Uma variação deste protocolo utiliza apenas um sítio secundário para cada primário, sendo conhecido como primário-*backup*. A desvantagem do protocolo primário-*backup* é que a disponibilidade é reduzida, uma vez que somente uma réplica de segurança é assegurada para as informações.

Alguns autores [19, 21, 22] descrevem uma otimização do protocolo de cópia primária. Na estratégia, o sítio primário executa a consulta de atualização localmente e repassa as modificações para os sítios secundários. Esta otimização ocorre, pois não é

necessário que haja uma fase de coordenação entre o sítio primário e os sítios secundários. Os sítios secundários apenas atualizam localmente os dados alterados a partir das modificações repassadas.

2.1.2 Réplicas Ativas

Os protocolos de réplicas ativas, também chamados de máquina de estados, têm por base manter a mesma sequência de execução das operações em todas as réplicas. Diferentemente da estratégia de cópia primária, a réplica ativa não possui um sítio centralizador. Com isso, para garantir a consistência, o protocolo assume a seguinte premissa: se todas as réplicas receberem as mesmas operações, e as executarem na mesma ordem, então o resultado produzido é igual em todas as réplicas. De acordo com Guerraoui e Schiper [21], o resultado de uma transação depende exclusivamente do valor inicial dos dados e da sequência de operações executadas sobre eles.

Para que esta premissa seja garantida, é necessário que cada transação requisitada contemple as seguintes condições:

- acordo: todos os sítios que não estão em suspeita de falha devem receber as mesmas operações;
- ordem: se uma réplica executa uma operação q_1 antes da operação q_2, então todas as réplicas devem seguir esta ordem.

As duas propriedades anteriormente descritas asseguram o critério de serializabilidade requerido pelos protocolos de replicação. A propriedade de *acordo* também é necessária ao protocolo de cópia primária e corresponde à característica de atomicidade global da execução de uma operação.

A ordem do processamento das operações pode ser uma propriedade relativa para solicitações concorrentes de clientes distintos. Se todos os sítios recebem requisições distintas, a consistência da base distribuída está garantida. No entanto, se dois clientes enviarem solicitações distintas para um mesmo dado, a ausência de um relógio global torna indeterminável a ordem real que estas requisições foram enviadas. Neste caso, qualquer ordem de entrega das mensagens pode ser aceita, contanto que esta ordem seja a mesma em todos os sítios.

Quando uma operação de atualização é solicita, o cliente propaga esta solicitação para todos os sítios. Os sítios trocam informações entre si para determinar a ordem de execução das operações, considerando as operações concorrentes. Posteriormente, cada sítio executa a atualização na sua réplica loca e retorna a confirmação da execução para o cliente. Não é necessária nenhuma confirmação posterior entre os sítios. O cliente aguarda a execução da solicitação até receber o primeiro resultado ou a maioria de resultados comuns. Na execução de uma operação de leitura, o cliente envia a solicitação para todos os sítios e aguarda apenas o primeiro resultado. A Figura 2.2 ilustra as trocas de mensagens entre um cliente e os sítios do sistema.

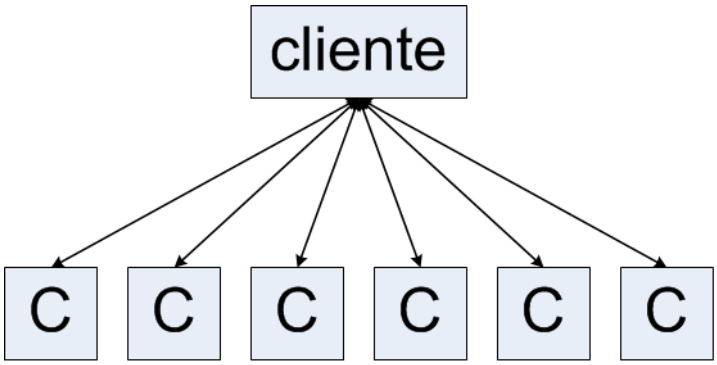


Figura 2.2 Protocolo de Replicação com Réplica Ativa

2.1.3 Comparação entre os Protocolos de Cópia Primária e Réplica Ativa

A principal vantagem do protocolo de cópia primária é que ele usa menor poder de processamento se comparado ao protocolo de réplicas ativas. Isso porque apenas o sítio primário realiza a execução da solicitação, inicialmente. Ao passo que, na estratégia de réplicas ativas, todas as réplicas executam as operações solicitadas. A principal desvantagem no protocolo de cópia primária consiste na sobrecarga do sítio primário, uma vez que este executa todas as consultas de leitura e atualização, diminuindo o grau de concorrência entre as requisições. Em contrapartida, no protocolo de réplicas ativas, as consultas são realizadas em todos os sítios, tendo o cliente a opção de aceitar a primeira resposta que for recebida.

Outra desvantagem da cópia primária é que, em caso de falha do sítio primário, o cliente deve eleger um novo sítio primário dentre os sítios secundários. Isso acontece porque a falha não é transparente ao cliente. No caso das réplicas ativas, qualquer réplica

pode apresentar falha ou indisponibilidade, de forma transparente para o cliente. Porém, as réplicas ativas necessitam de um mecanismo que garanta a consistência global do estado das réplicas. Isto não acontece na cópia primária, pois o sítio primário coordena todas as execuções realizadas pelos sítios secundários.

2.1.4 Replicação com Comunicação em Grupo

A abstração da comunicação em grupo facilita a concepção de aplicações distribuídas confiáveis que manipulem réplicas de dados. Uma solicitação externa realizada a um grupo de sítios é manipulada de forma atômica, garantindo entrega e ordem de propagação entre seus participantes. Os sítios que contém réplicas de um mesmo conjunto de dados podem formar um grupo de sítios ou grupo de replicação. Quando uma solicitação do cliente for aplicada ao grupo, todos seus sítios recebem as operações e a executam localmente, mantendo um estado distribuído consistente.

Um cliente realiza uma solicitação ao grupo por meio de uma primitiva de comunicação em grupo ao invés de enviar uma mensagem individual a cada sítio. Esta comunicação torna transparente o endereço de cada sítio do grupo, sendo necessário apenas o endereço único do grupo. Um sistema de comunicação em grupo contempla todas as primitivas necessárias para formar um grupo, identificar sítios com suspeita de falhas e inconsistência de réplicas.

O uso da comunicação em grupo é inapropriado para o caso da replicação parcial [17]. Isso porque, neste tipo de replicação, nem todos os sítios possuem uma cópia de todos os dados. Além do mais, para viabilizar as garantias que um mecanismo de comunicação em grupo proporciona, um grande número de troca de mensagens entre os sítios é realizado. Isto faz da solução de comunicação em grupo inadequada para ambientes com alta disponibilidade, como *clusters*, por exemplo.

No protocolo de cópia primária com a utilização de comunicação em grupo, cada sítio secundário é membro do grupo de replicação. As operações são recebidas e respondidas pelo sítio primário, sem acesso aos sítios secundários. As operações de atualização solicitadas pelos clientes são recebidas e executadas pelo sítio primário. As modificações dos dados são repassadas aos sítios secundários por meio de uma primitiva de comunicação em grupo, que garantem a entrega atômica da mensagem e mantêm a ordem de envio.

No caso do protocolo de réplicas ativas, todos os sítios pertencem ao mesmo grupo. Os clientes realizam as requisições para o endereço do grupo ao vindos de enviar mensagens aos sítios individualmente. A consistência das réplicas é garantida por primitivas de comunicação em grupo. Ao receber a requisição, cada sítio executa localmente a consulta e retorna a resposta ao cliente. A ordem na execução das consultas é garantida pelo mecanismo de comunicação em grupo, evitando que exista qualquer processo de concordância entre os sítios. No caso das operações de leitura, o cliente envia a requisição para um único sítio, pois todos os sítios do grupo possuem a réplica dos dados atualizada. Isso porque, caso um sítio falhe na execução de uma requisição, ele é excluído do grupo.

Ambos os protocolos que utilizam a abstração de grupos são aplicados para a replicação total, pois todos os sítios possuem uma réplica dos dados. O protocolo de réplicas ativas pode ser facilmente mapeado para uma abstração de comunicação em grupo. A cópia primária requer a existência do sítio primário, o que compromete a transparência da abstração de grupos.

2.1.5 Taxonomia de Protocolos de Replicação de Dados

Gray et al. [20] classificou os protocolos de replicação de bancos de dados usando dois parâmetros: consistência da base e execução do serviço para os clientes. Além do mais, em [17] são discutidos parâmetros adicionais que influenciam diretamente na replicação de bases de dados. Em particular, a granularidade da replicação também é discutida como uma classificação dos protocolos.

O parâmetro de consistência da base diz respeito à forma como a resposta é encaminhada ao cliente quando da realização de uma requisição. Sobre esses parâmetros, duas técnicas são citadas na literatura [20]:

- replicação preguiçosa (*lazy*): o sítio responde ao cliente imediatamente após receber a requisição, sem garantir que a réplica esteja consistente.
- replicação ávida (*eager*): o sítio responde ao cliente somente após garantir a consistência da réplica.

A replicação preguiçosa visa o desempenho enquanto que a replicação ávida visa a garantia da consistência. No entanto, na replicação ávida, o sítio espera bloqueado

até receber uma resposta de todos os sítios do grupo de replicação, antes de enviar uma resposta final ao cliente.

Quanto à execução do serviço do cliente, duas técnicas são possíveis [20]: cópia primária e qualquer réplica. Ambas as técnicas foram discutidas anteriormente neste capítulo como protocolos de replicação por cópia primária e réplicas ativas, respectivamente. No protocolo de cópia primária, apenas o sítio primário executa as requisições dos clientes, sendo os sítios secundários apenas *backups*. Em contrapartida, no protocolo de réplicas ativas, todos os sítios executam a requisição de um cliente, sendo desprendido um custo adicional para a coordenação entre eles.

Outro parâmetro abordado na literatura corresponde às técnicas de atualização de réplicas [17]. Este parâmetro diz respeito ao momento em que as atualizações executadas em um sítio são propagadas aos demais sítios do sistema distribuído. Nos protocolos baseados em cópia primária [23, 24], uma das réplicas é escolhida para realizar a propagação das atualizações para as demais réplicas. Ao passo que, nas abordagens baseadas em réplica ativa [25, 26], todas as réplicas executam a mesma sequência de atualizações de forma atômica.

A primeira técnica consiste da atualização imediata (*immediate update* ou *immediate write*). Nela, a propagação ocorre a cada consulta de atualização que é executada. Assim, cada consulta de atualização gera uma transação distribuída para propagar as modificações. Esta propagação pode ser realizada de forma linear ou constante. Na primeira forma, as modificações são enviadas a cada transação, enquanto que na forma constante é definido um intervalo de tempo configurável para o envio das atualizações. Geralmente, esse tipo de controle de consistência é usado quando não há necessidade de se obter os dados totalmente atualizados.

Por outro lado, a técnica de atualização adiada (*deffered update* ou *deffered write*) propaga as atualizações de uma transação apenas ao final da sua execução completa. Desta forma, cada transação gera apenas uma transação distribuída, a qual propaga as modificações aos demais sítios. No caso, as réplicas cooperam usando estratégias para manter a consistência das réplicas. Sistemas de banco de dados síncronos tradicionalmente utilizam o protocolo *two-phase commit* (2PC) [27]. Nessa abordagem, uma réplica é encarregada de coordenar e confirmar a difusão das modificações para as demais.

A escolha da melhor estratégia depende da disponibilidade de comunicação, da frequência das atualizações e do volume das informações requisitadas pelos usuários. A

principal desvantagem da atualização imediata é que ela corrompe a propriedade de isolamento de uma transação. O isolamento prevê que os resultados obtidos de uma transação somente são utilizados por uma transação externa após o encerramento e confirmação desses resultados [13]. Com a técnica de atualização imediata a propriedade de isolamento não é assegurada. Assim, o principal problema ocorre quando, por algum motivo, uma transação distribuída precisa ser abortada. Neste caso, cada sítio envolvido desfaz as modificações requisitadas pela transação distribuída, comprometendo o desempenho do serviço.

Outra desvantagem da técnica de atualização imediata consiste na ocorrência de conflitos entre transações. Duas transações são conflitantes quando realizam operações sob um mesmo conjunto de dados. Caso o conflito entre transações não seja tratado, as transações podem gerar resultados inconsistentes. O tratamento de conflitos é muito dispendioso, pois o grau de comunicação na rede é alto e todos os participantes devem estar conectados. Além do mais, as transações conflitantes podem causar *deadlocks*. Em caso de *deadlock*, uma das transações é abortada para que os sítios envolvidos possam ser liberados. Gray et al. [20] provaram que o protocolo 2PC é impraticável quando a quantidade de réplicas é grande, pois o número de *aborts*, *deadlocks* e mensagens trocadas cresce exponencialmente com relação ao número de réplicas.

2.1.6 Exemplos de Protocolos de Replicação

Hu e Kemme [25] apresentam o Postgres-R(SI), que consiste de uma extensão para o banco PostgreSQL, que se baseia no protocolo de máquinas de estado proposto por Pedone et al. [28]. Este protocolo trabalha de forma síncrona e utiliza a estratégia de réplicas ativas, sendo composto por três fases: (i) a execução local das transações numa das réplicas de forma otimista; (ii) a difusão atômica das atualizações para todas as réplicas; e (iii) um procedimento determinístico de certificação. A execução das transações é otimista, pois cada réplica processa localmente a transação sem qualquer sincronização com as demais, evitando assim a sobrecarga associada ao controle de concorrência distribuído [20]. Em seguida, o estado associado à execução da transação é difundido com garantias de atomicidade e ordem total na entrega, utilizando primitivas de comunicação em grupo (CG). Por fim, o procedimento de certificação garante que as transações que violem os pressupostos de serialização sejam abortadas. A ordenação total das mensagens aliada ao determinismo do procedimento de certificação assegura um estado global coerente entre

as várias réplicas do sistema.

Pacitti et al. [24] apresentam o protocolo RepDB, que baseia-se na estratégia de réplicas ativas e propaga as modificações de forma assíncrona usando uma primitiva de *multicast* confiável para difundir as atualizações. Nesse protocolo, as transações submetidas são interceptadas por um balanceador de carga, que escolhe uma réplica e a envia. Cada transação T é associada com um *timestamp* cronológico de valor C e é enviada através de *multicast* para as demais réplicas. Em cada réplica, um *delay* de tempo é introduzido antes de iniciar a execução de T. Esse *delay* corresponde a um limite superior ao tempo necessário para realizar o *multicast* de uma mensagem. Quando o *delay* expira, as transações com valor inferior a C são finalizadas (*commit*) antes de T, seguindo a ordem de *timestamp* cronológico (ordenação total). O uso dessa variável, associado às primitivas de ordenação, previne os conflitos e mantém a consistência.

O PDBREP [23] organiza os sítios em dois grupos: *leitura* e *atualização*. O protocolo possui quatro tipos de transações: atualização, leitura, propagação e *refresh*. As réplicas do grupo de atualização processam transações de atualização, que são aquelas que contêm pelo menos uma operação de modificação dos dados. As réplicas do grupo de leitura recebem apenas transações de leitura, isto é, não realizam quaisquer operações de escrita. Todas as réplicas são gerenciadas de forma assíncrona. As réplicas de leitura possuem filas locais, as quais armazenam as alterações recebidas do grupo de atualização para posterior execução. Essas alterações são efetivadas quando uma réplica está ociosa, através de uma transação de propagação. Há casos em que outra transação necessita de dados atualizados, fazendo com que os dados sejam modificados por meio de uma transação especial chamada de *refresh*.

2.2 ASPECTOS DE REPLICAÇÃO DE DADOS XML

2.2.1 XML

Em razão dessa crescente utilização do XML, torna-se necessária a existência de sistemas eficientes de armazenamento e recuperação de dados nesse formato. Existem várias estratégias para o gerenciamento de dados XML, dentre as quais se podem destacar: SGBDXNs e os SGBDs XML habilitados.

A flexibilidade na estrutura dos dados XML impõe novos desafios para o processa-

mento de consultas e de transações, de modo que novas técnicas devem ser desenvolvidas, tanto em ambientes centralizados como distribuídos. Isso decorre principalmente do modelo de representação destes dados, onde os documentos XML são representados em grafo, o que adiciona maior complexidade à sua estrutura e à heterogeneidade, visto que um mesmo subelemento pode ser omitido ou repetido várias vezes. Por exemplo, o processamento de consultas a dados XML ainda não dispõe de uma álgebra padrão, o que dificulta a decomposição e a otimização de consultas. Com relação aos protocolos para controle de concorrência, a maioria das soluções existentes proporciona baixo nível de concorrência, bloqueando grande parte dos dados XML.

Uma organização bastante utilizada no gerenciamento de dados XML é o conceito de coleções de documentos XML. Uma coleção é um agrupamento de documentos de acordo com sua relevância semântica [15]. As coleções têm papel similar às tabelas em SGBDs relacionais ou diretórios em um sistema de arquivos, também sendo usados nos SGBDs orientados a objetos. A tecnologia XML apresenta um conjunto de especificações que possibilita o gerenciamento de dados neste formato, tais como: validação e processamento de documentos, e linguagens de manipulação. Os grandes fabricantes de sistemas estão integrando o formato XML cada vez mais em seus produtos, tais como: SGBDs, *browsers* e ferramentas de desenvolvimento [29].

2.2.2 Formas de Armazenamento

Documentos XML podem ser armazenados de formas diversas, da maneira mais simples, em sistemas de arquivos, passando por SGBDs XML habilitados até SGBDXNs. Essas modalidades existem para serem utilizadas em diferentes contextos. Em aplicações de pequeno porte, onde os documentos são pequenos e raramente modificados, a abordagem de sistemas de arquivos oferece grandes benefícios, por ser o modo mais simples. Para aplicações de médio e grande porte, onde existem grandes volumes de documentos e operações de manipulação, o uso de SGBDs habilitados e SGBDXNs torna-se mais atrativa.

Em SGBDs habilitados, um SGBD tradicional emprega técnicas de mapear dados XML para seu formato, a fim de aproveitar o poder de processamento existente nesse modelo [30]. Como exemplo de SGBDs habilitados, podem ser mencionados os sistemas Oracle e o IBM DB2. Diversas técnicas de mapeamento são propostas pela comunidade acadêmica [31] [32] [33].

Segundo [9], SGBDXNs são sistemas construídos especialmente para o gerenciamento de dados XML, ou seja, possuem a capacidade de definir, criar, armazenar, manipular, publicar e recuperar documentos ou fragmentos de documentos XML. Nesses sistemas, um documento XML é representado como um grafo ou uma estrutura de árvore, onde os nós representam elementos e atributos, e as arestas definem os relacionamentos elemento/subelemento ou elemento/atributo [8]. Os SGBDXNs incorporaram muitas características presentes em sistemas tradicionais, tais como armazenamento, indexação, processamento de consultas, transação e replicação [7, 9, 8, 10, 12, 11].

Um repositório de dados XML pode ser de dois tipos: Único Documento (Single Document, SD) e Múltiplos Documentos (Multiple Document, MD) [34]. Em repositórios do tipo SD, todas as informações estão armazenadas em um único documento XML (com um esquema definido em XML Schema ou DTD). Já em repositórios MD, também existe a definição de um esquema, porém, o repositório é formado por múltiplas instâncias (documentos XML) desse esquema, formando uma coleção de documentos.

XPath e a XQuery constituem as principais linguagens para manipulação de dados XML. O XPath consiste em expressões de caminho utilizadas para recuperar elementos na estrutura em árvore do XML. A XPath contempla um conjunto de expressões de seleção que podem ser utilizados em conjunto com as expressões de caminho para selecionar os elementos com base no conteúdo de cada um deles [2]. A XQuery consiste em uma linguagem com um vasto conjunto de funcionalidades, que permite estruturas de laço e aninhamentos, além de suportar a execução de operações entre documentos XML. A XQuery segue uma estrutura similar à linguagem SQL, contendo cláusulas específicas para seleção, projeção, ordenação, junção e outras operações sobre dados. Uma consulta XQuery segue a estrutura FLWOR e permitem a utilização de expressões XPath na sua composição [35]. Segue abaixo um exemplo de consulta XQuery:

FOR $l IN DOC("livros.xml")/livraria/livro
WHERE $l/preco > 30
ORDER BY DESC $l/ano
RETURN $l/autor

Segue abaixo a descrição das principais cláusulas XQuery:

- FOR LET: indica qual documento/coleção será utilizada na consulta;

- WHERE (opcional): define os critérios de seleção a serem aplicados aos elementos XML.
- ORDER BY (opcional): define os critérios de ordenação a serem aplicados aos elementos XML.
- RETURN: descreve quais elementos serão retornados, bem como será montado o resultado para o cliente.

Além destas cláusulas, XQuery permite outras cláusulas como GROUP BY, bem como funções MAX, MIN, COUNT e AVERAGE.

XPath e XQuery não possuem suporte a operações de atualização de documentos, tais como inserção ou remoção [36]. Alguns trabalhos propõem soluções para esse problema. Em [37] é apresentada uma linguagem para atualização de documentos XML denominada XUpdate. Já [38] discute alterações no núcleo da linguagem XQuery, fornecendo suporte a atualização.

2.2.3 Fragmentação

A fragmentação de dados consiste em determinar alternativas para divisão dos dados em unidades menores chamadas fragmentos, de tal forma que estas partes possam ser replicadas e distribuídas. De acordo com Ozsu et al. [13], para que a uma fragmentação garanta a integridade dos dados, é necessário que sejam verificados os critérios de completude, disjunção e reconstrução. A completudo consiste em mostrar que todas as informações da base original estão contidas em algum fragmento. A disjunção visa garantir que, para quaisquer dois fragmentos, não existem informações comuns entre eles. Por fim, a reconstrução consiste em mostrar que qualquer informação obtida a partir da base original pode ser recuperada a partir de operações em dois ou mais fragmentos.

No modelo relacional, temos duas maneiras de fragmentar os dados: a fragmentação *horizontal*, que divide uma tabela em função das suas tuplas, e a fragmentação *vertical*, que divide uma tabela com base em conjuntos de atributos. Outro tipo de fragmentação, conhecida como híbrida ou mista, é obtida pela combinação das duas estratégias descritas anteriormente.

A maioria dos trabalhos de fragmentação de dados no contexto XML tenta adaptar as técnicas tradicionais para solucionar esse problema, sendo inicialmente abordados por Ma et al. [39] e Bremer e Gertz [40]. Estes trabalhos adaptam as ideias de fragmentação em bases de dados relacionais e orientadas a objetos para o modelo de dados XML, considerando suas características específicas.

Ma e Schewe [41] elaboraram técnicas de fragmentação de dados XML como adaptação das estratégias do modelo de objetos. Nesse trabalho, os autores propuseram as fragmentações: horizontal, que agrupa os elementos de um documento XML aplicando um critério de seleção; vertical, que divide a estrutura do esquema DTD; e tipo híbrido chamado *split*, que combina as estratégias de fragmentação horizontal e vertical para dividir um documento em um conjunto de documentos com esquema diferente do documento original. Esta estratégia não apresenta um modelo forma para provar os critérios de completude e corretude (disjunção e reconstrução). Além do mais, sua abordagem não é apropriada para repositórios MD, necessitando que os documentos sejam integrados em uma visão SD.

Bremer e Gertz [40] apresentam princípios de fragmentação adotados por bancos de dados relacional e orientado a objetos adaptados ao modelo XML. Esse trabalho utiliza um esquema de sumário *Repository Guide* para auxiliar na divisão dos dados e verificação dos critérios de completude e disjunção. São apresentadas a fragmentação vertical, que consiste no particionamento do sumário *Repository Guide*, e fragmentação horizontal, que realiza a divisão do documento aplicando condições aos seus atributos. Apesar de contemplar os critérios de fragmentação, o formalismo apresentado não define as técnicas desenvolvidas. Além do mais, a proposta limita-se a linguagem XPath e não apresenta resultados que validem diretamente a proposta de fragmentação.

Buneman et al. [42] adaptaram a técnica de vetorização, que consiste na divisão dos dados em colunas, a documentos XML. Sua estratégia consiste em decompor um documento em um conjunto de vetores e armazená-los em tabelas relacionais. Cada vetor contém um caminho desde a raiz até uma folha da árvore XML. Assim, para permitir a execução de consultas, a solução suporta um subconjunto da linguagem XQuery, que pode ser decomposta e executada distribuidamente. Resultados experimentais demonstraram melhorias no processamento das consultas, mas o subconjunto limitado da XQuery e a verificação apenas do critério de *reconstrução* nesse trabalho não viabilizam sua utilização em bases de XML.

Andrade et al. [43] criaram o PartiX, que consiste em uma arquitetura para o processamento de consultas XQuery sobre bases de dados XML fragmentadas. Diferentemente dos demais trabalhos, as operações de fragmentação não são aplicadas sobre um único documento XML (*Single Document*), mas sim a uma coleção deles (*Multiple Documents*). A fragmentação horizontal consiste em uma operação de seleção que satisfaz um determinado predicado, a vertical consiste em uma operação de projeção, e a híbrida, uma combinação das duas anteriores. Os experimentos realizados demonstraram melhorias no desempenho das consultas frente ao modelo centralizado. Nesses experimentos, os fragmentos são disjuntos, e as subconsultas foram executadas em paralelo. Apesar dos resultados satisfatórios apresentados, a fragmentação do PartiX é aplicada a uma coleção de documentos XML, gerando fragmentos que são subconjuntos da base original. Assim, os elementos XML não são fragmentados, inviabilizando a utilização do PartiX a documento XML único. Por fim, a decomposição de consultas XQuery considera um subconjunto bastante limitado desta linguagem.

Kurita et al. [44] propôs uma estratégia eficiente para processamento de consultas para grandes bases de dados XML, em ambientes distribuídos. A ideia principal desse trabalho é balancear os custos de armazenamento e processamento de consultas dentre os nós do sistema. Para isso, é utilizada a fragmentação vertical como estratégia de particionamento da base de dados, que se baseia na razão entre o tamanho da base e o número de nós para os quais os fragmentos serão distribuídos. Além disso, é proposta uma estratégia de realocação dinâmica dos dados para manter o balanceamento do sistema, que consiste em modificar a estrutura dos fragmentos e mover dados XML entre os nós. Seus experimentos consideram apenas consultas que não necessitassem de operações de junção entre as estruturas. Os resultados demonstraram melhorias no desempenho das consultas utilizando sua estratégia de fragmentação associada à realocação dinâmica dos dados. Porém, é citado que o sistema pode se tornar ineficiente caso as consultas sejam aplicadas a sítios específicos.

Embora existam várias estratégias de fragmentação de dados, elas podem ou não ser estratégias vantajosas. Em bases de dados com um grande volume de informações, a fragmentação pode ser uma alternativa para a execução de consultas de forma eficiente. Esse ganho no desempenho é possível devido ao fato das consultas poderem ser decompostas em subconsultas, sendo executadas paralelamente em diferentes nós do sistema, o que aumenta sua vazão. Além do mais, estas subconsultas são executadas em um subconjunto (fragmento) da base de dados original, podendo aumentar o desempenho dado

que a operação é aplicada em sob um volume menor de dados.

Porém, existem situações em que a fragmentação pode se comportar de forma insatisfatória, degradando assim o desempenho do sistema. Existem cenários em que a recuperação dos dados pode implicar em operações de junção e união sob os fragmentos, gerando um custo adicional. De forma similar, a análise semântica das operações pode sofrer de um custo adicional caso seja necessário acessar bases de fragmentos em dois ou mais sítios [13].

2.2.4 Alocação de Dados

De acordo com Dowdy e Foster [45], o critério de otimalidade para o problema de alocação de dados leva em consideração dois quantificadores: *custo* e *desempenho*. O custo está relacionado ao somatório dos custos referentes a consultas analisadas, atualização dos fragmentos e comunicação. O desempenho está associado às métricas de *tempo de resposta* e *vazão* de comunicação dos nós. O modelo de alocação deve buscar um custo mínimo para as operações aplicadas no sistema, além de minimizar o tempo de resposta para cada consulta e maximizar a vazão em cada nó do sistema. Na literatura, encontramos vários modelos para alocação de dados [46, 13]. Outros trabalhos adaptam as estratégias de alocação para o modelo de dados XML [40, 44].

2.3 TRABALHOS RELACIONADOS

O eXist [10] é um SGBDXN de código livre desenvolvido em Java. Neste banco de dados, os documentos XML são organizados em coleções e armazenados em arquivos. Esses documentos são decompostos e, através de um esquema de numeração, são atribuídos números aos seus elementos e atributos, e assim armazenados de acordo com o modelo DOM. O eXist possui um mecanismo de replicação baseada em comunicação em grupo para sincronizar as réplicas. Ele utiliza replicação total dos dados e trabalha de acordo com o protocolo de cópia primária, propagando as alterações de forma síncrona. Quando uma operação de atualização é enviada ao grupo de replicação, ela é redirecionada para a cópia primária e bloqueia as cópias secundárias. Quando a cópia primária executa a atualização, esta é propagada para as cópias secundárias, que posteriormente são desbloqueadas para executar as novas requisições.

O X-Hive/DB [11] é um SGBDXN comercial desenvolvido pela X-Hive Corporation. Entre os padrões que suporta estão XQuery, XML Schema, XUpdate e DOM. Além de trabalhar no tradicional modelo cliente-servidor, o X-Hive/DB pode atuar como *WebService*, possuindo também diversas outras interfaces de acesso. O X-Hive apresenta um mecanismo de replicação baseado em cópia primária, executando as atualizações de forma assíncrona. As atualizações são armazenadas em um *log* e enviadas posteriormente para as cópias secundárias. Como as modificações são executadas de forma assíncrona, as aplicações que acessam as cópias secundárias podem ler dados desatualizados.

O Tamino é um SGBDXN desenvolvido pela Software AG [7], que usa uma evolução do banco ADABAS como seu armazenador de dados. Esse banco possui estruturas de índices, suporte para tratamento de informações do esquema XML, mecanismo para o processamento de transação e uma camada de *interface* para a *Web*. O Tamino permite a replicação de duas formas: protocolo de cópia primária e o protocolo 2PC. Quando o Tamino é configurado com o protocolo de cópia primária, a replicação ocorre de forma similar ao banco X-Hive. No caso do protocolo 2PC, a execução é semelhante aos bancos tradicionais.

Sousa et al. [15] apresentam o RepliX, um mecanismo para replicação de dados XML, desacoplado de qualquer SGBD. Ele melhora a disponibilidade desses sistemas, redirecionando as requisições dos clientes para réplicas operacionais, assim como o desempenho, se beneficiando de acessos concorrentes às réplicas. O RepliX combina técnicas síncronas e assíncronas, além de primitivas de comunicação em grupos para garantir a consistência das réplicas. O critério *one-copy serializability* [27] é usado neste trabalho como modelo de corretude, o qual garante a integridade dos dados. Uma visão geral da arquitetura do RepliX pode ser observada na Figura 2.3.

O RepliXDriver é o componente que permite o acesso ao mecanismo. Este *driver* é uma interface simples para a execução de transações, encapsulando as funcionalidades do RepliX e permitindo que o desenvolvedor se abstraia da sua arquitetura. O RepliX-Coordinator é composto de três partes: o escalonador, responsável por identificar o tipo das transações; o roteador, que decide onde as transações serão executadas; e o balanceador, que realiza balanceamento de carga entre as bases de dados. O RepliXNode é o componente acoplado a cada um das bases de dados. Esse componente acessa o BDXN e executa as transações solicitadas, repassando os resultados ao RepliXDriver que, por sua vez, repassa-os à aplicação. No capítulo seguinte, detalhamos cada um desses componentes.

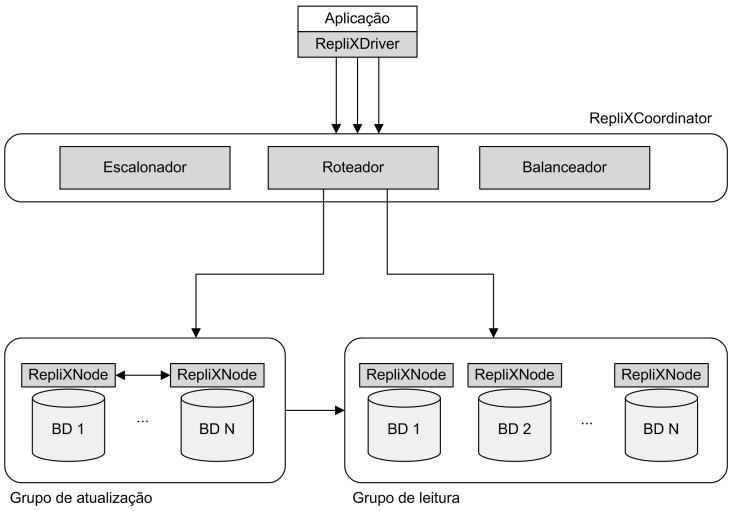


Figura 2.3 Arquitetura do RepliX

No RepliX, os sítios (nós de rede com recursos computacionais) são divididos em dois grupos distintos: grupo de leitura e grupo atualização, que tratam transações de leitura e atualização respectivamente. O RepliX realiza a distinção entre as transações, classificando-as de acordo com o conteúdo de suas operações. Transações que contenham apenas operações de leitura são consideradas de leitura. Caso a transação contenha pelo menos uma operação de modificação (inserção, atualização ou remoção), é classificada como de atualização. A estratégia de dividir os sítios em grupos é um aspecto importante do RepliX. Essa abordagem visa melhorar o desempenho do sistema e diminuir conflitos durante as operações de atualização, já que o controle de concorrência a dados XML ainda apresenta muitas limitações.

Quando uma transação é submetida, o RepliX verifica o seu conteúdo e a direciona para um dos grupos. Caso a transação seja direcionada para o grupo de atualização, um sítio deste grupo a recebe. Esse sítio é chamado de *primário* e é o responsável por verificar conflitos com as demais transações que estão sendo executadas localmente e, em seguida, enviar um *multicast* com a propriedade de ordenação total para os demais sítios desse grupo. Esses sítios são chamados de *secundários* em relação ao primário que enviou o *multicast* e realizam um *teste de certificação*, que verifica se uma transação local no *primário* está em conflito com as demais transações em execução nos *secundários*. Esse teste garante o critério de *serializabilidade*: a transação é abortada se a sua confirmação gera estado inconsistente no grupo de atualização. Se a transação passa no *teste de certificação*, então ela é confirmada no grupo de atualização.

As transações executadas no grupo de atualização recebem um identificador único, o que permite sua identificação pelo RepliX. As modificações aplicadas ao grupo de atualização são serializadas e enviadas continuamente pelo sítio primário ao grupo de leitura, através de um *multicast* com a propriedade de ordenação FIFO. Essas modificações são adicionadas em filas locais de cada sítio do grupo de leitura e executadas na mesma sequência que foram aplicadas ao grupo de atualização.

O grupo de leitura executa dois tipos de transações: propagação e *refresh*. Transações de propagação são executadas durante o tempo ocioso de um sítio, ou seja, quando não estão sendo executadas transações de leitura ou transações de *refresh*, com o objetivo de efetivar as atualizações. Transações de *refresh* são aplicadas para adicionar as transações contidas na fila local a um sítio do grupo de leitura.

Durante a execução das transações de leitura em um determinado sítio, o RepliX gerencia as réplicas através da aplicação das transações de propagação e de *refresh*. Caso novas modificações sejam adicionadas na fila local, o RepliX continua a execução da consulta nesse sítio e posteriormente executa uma transação de propagação, adicionando o conteúdo da fila ao banco de dados local. Quando uma nova transação é direcionada para esse sítio, o RepliX realiza as seguintes verificações: (i) se a nova transação necessita de dados que foram atualizados, uma transação de *refresh* é executada, (ii) caso contrário, a transação é executada sem a necessidade de transações de *refresh* ou propagação.

Avaliou-se o RepliX considerando diversas características de replicação, cuja metodologia consistiu em comparar um SGBDXN convencional em relação ao RepliX acoplado a ele. Para isso, foi utilizado o *benchmark* XMark para processar cargas semelhantes num mesmo cenário de execução a ambos os casos de teste, e o SGBD XML Sedna. Pela análise dos resultados obtidos, foi possível verificar que o RepliX melhorou o desempenho e a disponibilidade do SGBDXN, mesmo em cenários de replicação com grande proporção de atualizações.

Apesar do mecanismo de replicação do eXist se basear em primitivas de CG, elas são utilizadas apenas para a troca confiável de mensagens. O X-Hive e o Tamino aplicam técnicas tradicionais para prover replicação. Contudo, essas técnicas não são apropriadas para replicar dados XML. O protocolo de cópia primária utilizado pelos bancos eXist, X-Hive e Tamino não é tolerante a falhas nem favorece a escalabilidade [13]. Nenhum dos trabalhos analisados apresentou resultados que comprovem a eficiência de seus mecanismos. Além disso, as soluções por eles apresentadas não possuem portabilidade, já

que são fortemente acopladas a SGBDs específicos.

A Tabela 2.1 faz um comparativo entre as soluções apresentadas e o RepliXP, considerando como critérios o tipo de protocolo de replicação, cujos valores são cópia primário (CP) ou réplica ativa (RA), as formas de propagação das atualizações síncrona (S) ou assíncrona (A) e a granularidade da replicação, que pode ser parcial (P) ou total (T).

	eXist	**X-Hive**	**Tamino**	**RepliX**	**RepliXP**
Protocolo	CP	CP	CP ou RA	CP e RA	CP e RA
Propagação	A	A	S ou A	S e A	S e A
Granulosidade	T	T	T	T	P e T

Tabela 2.1 Comparativo entre os trabalhos relacionados e o RepliXP

2.4 CONCLUSÃO

Neste capítulo, foram apresentados os fundamentos teóricos que embasaram o RepliXP. Em particular, foram apresentados os principais protocolos de replicação e um comparativo entre eles. Posteriormente, foram expostos aspectos de replicação sob dados XML. Por fim, foram elencados os trabalhos relacionados e realizada uma análise critica sob cada um deles.

CAPÍTULO 3

MECANISMO DE REPLICAÇÃO PARCIAL DE DADOS XML

Este capítulo descreve o RepliXP, um mecanismo para replicação parcial de dados XML. Seu propósito fundamental é aumentar o desempenho de sistemas de bancos de dados XML. O RepliXP utiliza um protocolo para controle de concorrência eficaz, que melhora o acesso concorrente aos dados. Também são discutidos aspectos arquiteturais, especificação, cenários de execução e os algoritmos desenvolvidos.

3.1 CARACTERÍSTICAS

O objetivo deste trabalho é desenvolver um mecanismo para replicação parcial de dados XML. O RepliXP utiliza um protocolo de replicação parcial, que aumenta o paralelismo entre as transações. A propagação das atualizações de dados é feita de forma assíncrona, o que minimiza a quantidade de bloqueios.

O PartiX [16] foi utilizado neste trabalho como estratégia de fragmentação horizontal de dados. Esta metodologia é aplicada a uma coleção de documentos XML, que corresponde a uma base MD. As cópias dos fragmentos podem ser distribuídas nos sítios utilizando qualquer heurística de alocação de dados. A identificação dos fragmentos e decomposição das consultas é baseada na estratégia proposta por [16].

O critério *one-copy serializability* [18] também fora adotado neste trabalho como modelo para garantir a integridade dos dados. Esse modelo garante que a execução de transações concorrentes produz resultados equivalentes a uma execução seqüencial do mesmo conjunto de transações em uma instância do banco de dados. Isso significa que clientes de um sistema replicado o enxergam como um banco com apenas uma instância e que operações de leitura sempre obtêm dados atualizados.

O RepliXP consiste de uma extensão do RepliX [15] com suporte à replicação

parcial de bases XML. Segue abaixo as principais adaptações realizadas a partir do RepliX:

- foi adotado uma protocolo de replicação assíncrona baseado em cópia primária para os sítios de atualização. Com isso, evita-se que hajam transações abortadas em decorrência de conflitos;
- inseriu-se um novo elemento arquitetural para realizar o processamento da consulta distribuída. Foi elaborada uma estrutura de catálogo de dados, que armazena as informações de definição dos fragmentos e distribuição de réplicas dentre os sítios;
- alterou-se a estratégia de execução das consultas de leitura, pois os dados lidos por uma consulta podem estar em dois ou mais sítios distintos;
- o controle da concorrência foi modificado, a fim de aumentar a quantidade de transações executadas paralelamente. Características do protocolo PDBREP [23] foram acopladas ao mecanismo para garantir o estado consistente da base distribuída;
- a estratégia de balanceamento de carga foi modificada com o objetivo de proporcionar uma melhor distribuição na execução distribuída das consultas. Para isso, ele considera como carga a quantidade de consultas que estão sendo executadas no sítio.

O RepliXP considera a sequência clássica de execução de transações, na qual clientes iniciam uma transação, enviando uma requisição *begin* para, em seguida, fazer requisições de escrita e/ou leitura [47]. A transação é finalizada por uma requisição *commit* ou *abort*. Se a requisição *commit* for enviada e executada com sucesso, as atualizações persistem no meio de armazenamento. Caso contrário, se for enviada uma requisição *abort*, a transação é cancelada e suas modificações são desfeitas.

3.2 ARQUITETURA

O RepliXP é composto por três componentes de software que se comunicam remotamente, para que o mecanismo execute as operações sob os dados de forma correta. Alguns componentes são divididos em módulos que realizam atividades específicas e interagem

entre si por meio de chamadas locais. A Figura 3.1 ilustra os elementos da arquitetura e a integração entre eles, assim como os módulos que compõem cada um deles.

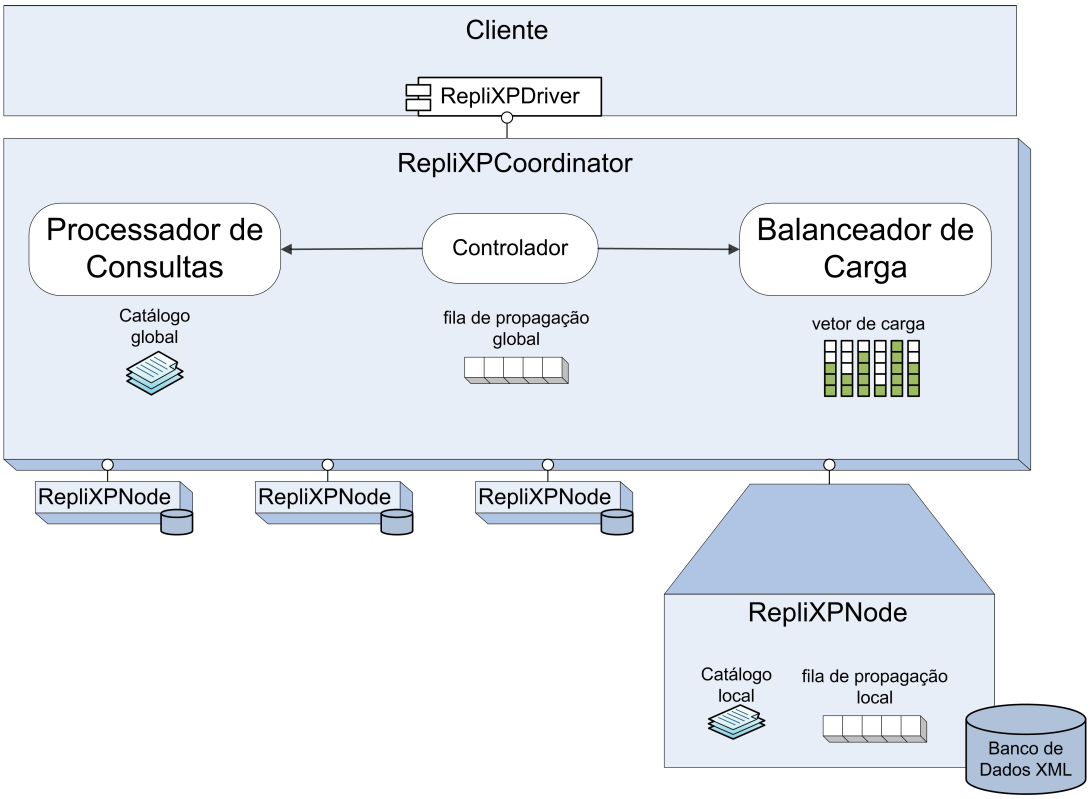


Figura 3.1 Arquitetura do RepliXP

O *RepliXPDriver* corresponde ao componente pelo qual a aplicação cliente acessa o mecanismo. A aplicação cliente realiza chamadas a métodos locais do RepliXPDriver e ele se encarrega de transformá-las em requisições remotas com o RepliXPCoordinator.

O *RepliXPCoordinator* é o componente que coordena a execução de todas as transações submetidas pelos clientes. Este componente é dividido em três módulos: Controlador, Processador de Consultas e Balanceador de Carga. O *Controlador* é o módulo central do componente RepliXPCoordinator. A função do Controlador é orquestrar a execução das transações, distribuindo as requisições dentre os sítios do sistema. A requisição a um sítio é feita via conexão remota com o componente RepliXPNode daquele sítio. Outra função do Controlador é manipula uma estrutura chamada fila de propagação global. Esta fila tem por objetivo propagar as modificações aos sítios do sistema. Os demais módulos auxiliam o Controlador na manipulação das transações. O *Processador de Consultas* tem a finalidade decompor ou reescrever as consultas, quando necessário. Para tanto, ele utiliza as informações dos fragmentos e alocação de dados que estão presentes

na estrutura de catálogo de dados global. O *Balanceador de Carga* controla a carga de trabalho dos sítios que executam as consultas de leitura. Ele indica ao Controlador qual sítio possui a menor carga dentre os sítios que podem ser consultados.

O *RepliXPNode* consiste do componente que é executado em cada sítio que possui uma base de dados. Sua função é receber as requisições do RepliXPCoordinator e aplicá-las localmente. Ele possui duas estruturas: *fila de propagação local* e *catálogo de dados local*. A fila de propagação local armazena as modificações que são repassadas pelo RepliXPCoordinator, executando-as quando nenhuma consulta estiver sendo executada no sítio. O catálogo local consiste de um subconjunto de informações do catálogo global e serve para manter a consistência da base de dados do sítio, garantindo o critério de serializabilidade.

3.3 ESPECIFICAÇÃO

Seja uma base de dados XML *multiple document* homogênea $D = \{d_1, d_2, \ldots, d_n\}$, onde todos os elementos $d_i \in D$ respeitam a mesma estrutura. É aplicado um processo de fragmentação de dados XML sobre esta base, formando um conjunto de fragmentos $F = \{f_1, f_2, \ldots, f_k\}$. Cada fragmento é definido como $f_i = \langle D, \sigma_\mu \rangle$, onde σ_μ corresponde a uma expressão de seleção, sendo μ um predicado de seleção ou composição de predicados desse tipo. Um fragmento f_i é composto por todos os documentos $d_i \in D$ que satisfaçam a expressão σ_μ. Este processo de fragmentação garante que cada documento de D está contido em algum fragmento e que os fragmentos de F não possuem intersecção entre si.

O sistema é composto por um conjunto de sítios $S = \{s_c, s_1, s_2, \ldots, s_n\}$ conectados por uma rede de computadores. O sítio s_c corresponde ao coordenador, o qual é responsável por processar todas as requisições submetidas ao mecanismo. Os demais sítios são classificados em dois grupos: sítios de leitura e sítios de atualização, que processam as transações de leitura e atualização, respectivamente. O RepliXP classifica as transações em dois tipos: transação de leitura e transação de atualização. Caso a transação contenha pelo menos uma operação de modificação (inserção, atualização ou remoção), é classificada como de atualização. O tipo da transação é informado pelo cliente no momento da solicitação de execução. Cada sítio de leitura ou atualização possui um SGBDXN com suporte a armazenamento de coleções de documentos XML e processamento das linguagens padrão deste formato (XQuery, XPath e XUpdate).

Essa abordagem visa melhorar o desempenho do sistema e diminuir conflitos durante as operações de atualização, já que o controle de concorrência a dados XML ainda apresenta muitas limitações. Com essa estratégia, apenas uma parte dos sites precisa ser atualizada a cada modificação, de forma assíncrona. A Figura 3.2 ilustra uma possível distribuição dos sítios de um sistema no modelo adotado. Os sítios de atualização s_1 e s_2 possuem a base completa (livraria), enquanto que cada sítio de leitura s_3, s_4 e s_5 possui apenas um subconjunto dos fragmentos, distribuídos segundo alguma estratégia de alocação.

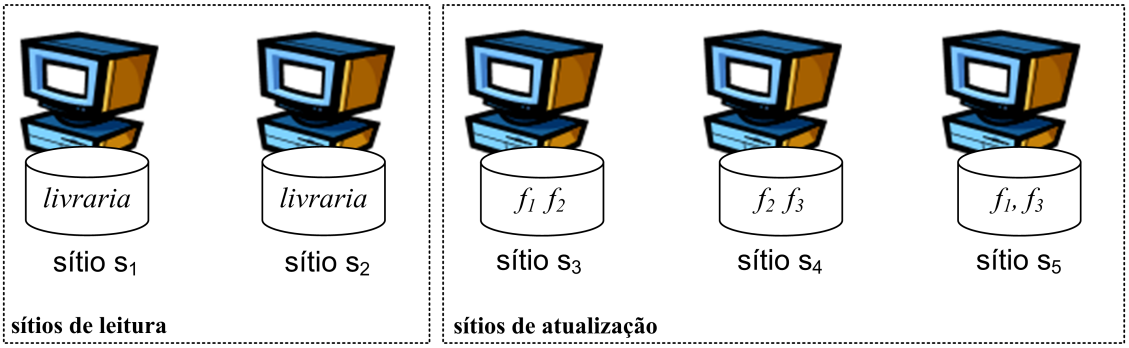


Figura 3.2 Exemplo de Distribuição dos Sítios e Fragmentos

No modelo adotado, uma transação corresponde a um conjunto de comandos sequenciais $T = \{b, q_1, q_2, \ldots, q_m, c\}$ o qual é submetido por uma aplicação cliente. Os comandos b e c representam instruções para iniciar e finalizar uma transação, respectivamente. Cada comando $q_i \in T$ consiste em uma operação de leitura ou atualização utilizando a linguagem XQuery [3].

Quando uma transação é solicita pelo cliente, o RepliXP verifica seu tipo, direcionando sua execução para um dos grupos de sítios. Caso a transação seja de atualização, ela será executada nos sítios de atualização. Cada sítio do grupo de atualização possui uma cópia completa da base original. Ao receber uma transação de atualização T_a, o coordenador repassa sequencialmente todas as consultas da transação para o sítio primário de atualização s_p. Este sítio executa as operações na sua base de dados local e, posteriormente, envia as atualizações para os demais sítios de atualização (sítios secundários). Este envio é realizado pelo sítio s_p ao submeter uma primitiva de comunicação em grupo *multicast* com a propriedade de ordenação total para todos os sítios secundários, garantindo o critério de serializabilidade.

Durante o envio das operações da transação T_a para o sítio primário, o coordenador seleciona as atualizações para propagá-las aos sítios de leitura. Como cada sítio

de leitura possui a base de dados parcialmente replicada, a modificação pode afetar dados de dois ou mais fragmentos alocados em diferentes sítios. Com isso, cada operação de atualização é decomposta em um conjunto de suboperações. Cada suboperação de atualização é aplicada aos dados de um único fragmento.

O primeiro passo no processo de decomposição da operação consiste em identificar os fragmentos envolvidos na atualização. Este processo necessita das informações de como a base está fragmentada. Para satisfazer essa necessidade, o sítio coordenador possui um catálogo de dados global, contendo informações sobre cada um dos fragmentos da base. Este catálogo consiste de um arquivo XML que armazena as seguintes informações para cada fragmento:

- identificador: código que identifica unicamente um fragmento no sistema;
- critério de seleção: expressão de seleção que define quais documentos da base original estão contidos em um fragmento;
- versão global: inteiro que determina a versão global de um fragmento. Esta versão é utilizada pelo mecanismo para garantir a consistência dos dados;
- sítios de alocação: identificador e endereço dos sítios que possuem uma cópia dos dados do fragmento. Esta informação é necessária para o segundo passo da decomposição de consultas.

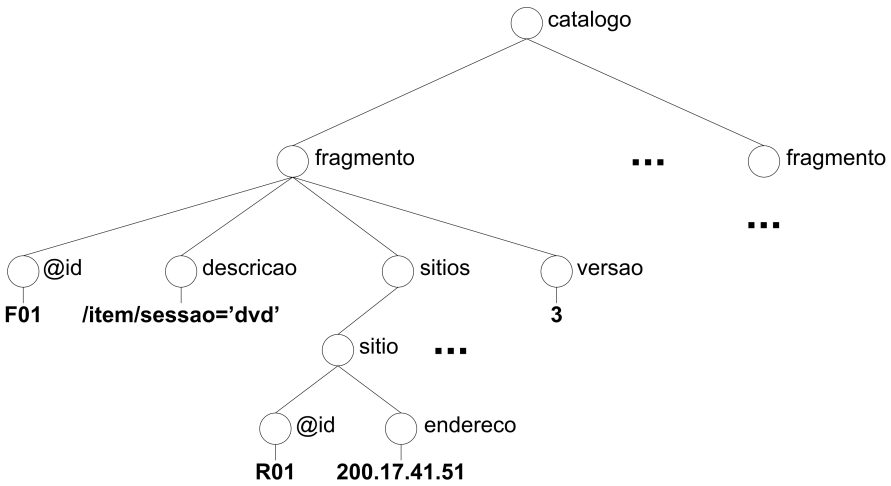


Figura 3.3 Esquema do Catálogo de Dados Global

A Figura 3.3 ilustra a estrutura de um catálogo de dados global com alguns valores de exemplo. O fragmento F01 possui o critério de fragmentação /item/sessao='dvd'. Um dos sítios de leitura em que o fragmento está alocado é o sítio R01 cujo endereço é 200.17.41.51. A versão global do fragmento é igual a 3.

Cada fragmento $f_i \in F$ da coleção D possui um conjunto de predicados simples $P_{f_i} = \{P_1, P_2, \ldots, P_k\}$. Seja q uma consulta baseada na especificação XQuery com um conjunto de predicados $P = \{p_1, p_2, \ldots, p_m\}$ e um conjunto de expressões de caminho $E = \{e_1, e_2, \ldots, e_n\}$. As expressões pertencentes a E correspondem às expressões encontradas nos predicados e na cláusula de retorno de q.

Considere $F_q = \{f_1, f_2, \ldots, f_x\}$, $F_q \subseteq F$, o conjunto de fragmentos envolvidos na consulta q. Para que sejam identificados os fragmentos do conjunto F_q, os seguintes passos são executados:

1. Para cada $p_j \in P$, se $p_j \in P_{f_i}$, então adicionar o fragmento f_i no conjunto F_q;
2. Para cada $e_l \in E$, se e_l é pré-fixada por algum caminho simples $P_y \in P_{f_i}$, e e_l não tem nenhuma comparação de valor e nenhuma aplicação de função, então adicionar o fragmento f_i no conjunto F_q.

O passo 1 analisa os predicados usados na consulta que também são utilizados na definição dos fragmentos. O passo 2 trata os casos em que a consulta contém expressões de caminho que possuem um prefixo que participa em alguma definição dos fragmentos baseado em um teste existencial. Vale ressaltar que estes dois casos não são exclusivos. Ambos podem se apresentar em uma única consulta. Se nenhuma das ocorrências for verificada na consulta, o conjunto F_q recebe todos os fragmentos de F.

O passo seguinte da decomposição é a criação das subconsultas. Este procedimento é feito a partir da consulta original e dos fragmentos de F_q. Inicialmente, as cláusulas `ORDER BY` e `GROUP BY` são retiradas da consulta. A consulta q possui como parâmetro da função `collection` na cláusula `LET` o valor D, pois a consulta é aplicada sob a base original. Assim, para cada fragmento $f_i \in F_q$ é criada uma subconsulta com o mesmo conteúdo da consulta q, modificando apenas o parâmetro da função `collection` pelo valor do identificador do fragmento f_i.

As operações de atualização que correspondem à inserção de um novo documento na base não necessitam de serem decompostas. No entanto, de acordo com o princípio de

disjunção dos fragmentos, um dado não pode pertencer a dois ou mais fragmentos. Assim, o documento precisa ser armazenado em um único fragmento. Para tanto, é necessário que a consulta seja reescrita, passando a informar o fragmento para o qual se destina a inserção.

De acordo com as linguagens de atualização para dados XML [38], o documento XML e a coleção destino são os principais parâmetros na inserção de um documento numa base de dados MD. Na reescrita, o critério de seleção σ_μ de cada fragmento $f_i \in F$ é aplicado ao documento. Seja $f \in F$ o fragmento para qual a operação retornou `verdadeiro`. Neste caso, o valor da coleção de destino da inserção é substituído pelo identificador do fragmento f.

Uma vez que todas as operações de atualização estejam decompostas e reescritas, elas são agrupadas numa única transação de propagação T_p. Para que as transações de propagação sejam enviadas para os sítios de leitura de forma assíncrona, elas são adicionadas em uma fila de propagação global Q_G. Um processo secundário do coordenador envia continuamente as transações que são incluídas na fila, garantindo a atomicidade e a ordem na entrega aos sítios de leitura.

O próximo passo do coordenador no processamento da transação T_a é atualizar a versão global dos fragmentos alterados. A versão consiste de um valor inteiro que é associado a cada fragmento, tanto no coordenador quanto nos sítios de leitura. As versões do fragmento do coordenador e do sítio de leitura são comparadas antes que a consulta de leitura seja executada. Este mecanismo de versionamento tem como objetivo garantir que as operações de leitura sejam aplicadas a dados atualizados. Por fim, o coordenador finaliza a transação no sítio primário, o qual consiste as alterações na sua base de dados.

Toda transação de propagação que chega a um sítio de leitura é adicionada a sua fila de propagação local Q_L. Esta fila armazena sequencialmente todas as transações de propagação que não foram aplicadas ao sítio. Quando nenhuma consulta está sendo executada no sítio, ele aciona um processo que executa as transações de propagação presentes na fila Q_L. No entanto, caso uma consulta seja iniciada durante este processo de atualização, os dados podem estar inconsistentes.

Para evitar que esta situação ocorra, o RepliXP possui um mecanismo de bloqueio que evita a manipulação de dados destualizados. O sítio possui a variável `status`, que indica seu estado em um determinado instante. Caso a variável `status` esteja com o valor `bloqueado`, nenhuma outra operação pode ser executada. Assim, quando o processo de

execução das transações de Q_L é acionado, o valor da variável `status` é modificado para `bloqueado`. Ao final do processamento das transações, o valor `desbloqueado` é atribuído à variável `status`.

Posteriormente, as transações de propagação são executadas sequencialmente na base local. Uma transação de propagação pode conter consultas de atualização que acessam dados que não estão presentes no sítio. Para saber quais fragmentos possuem uma réplica armazenada na sua base local, o sítio de leitura possui um catálogo de dados local. Este catálogo tem a mesma estrutura do catálogo de dados global. No entanto, gerencia apenas as informações de identificador e versão local dos fragmentos.

Cada consulta de atualização de uma transação de propagação é analisada antes de ser executada. Se a consulta modifica dados de um fragmento que não está armazenado no sítio, a alteração é descartada. Caso contrário, a consulta de atualização é executada na base local. Quando todas as consultas de uma transação de propagação são executadas, o sítio acrescenta uma unidade à versão local dos fragmentos alterados.

Quando uma transação de leitura é submetida por um cliente, o coordenador controla a execução de suas consultas nos sítios de leitura. A decomposição de uma consulta de leitura pode acontecer pelos mesmos motivos da decomposição de uma consulta de atualização. Ela é decomposta em subconsultas utilizando as regras de decomposição para atualizações. Para cada subconsulta, o coordenador associa a versão global do fragmento a ser lido. Este valor é utilizado pelo sítio de leitura que receber a subconsulta para checar a consistência da réplica do fragmento.

Cada subconsulta é submetida a um sítio de leitura que possua uma cópia do fragmento a ser lido. No entanto, dois ou mais sítios de leitura podem conter uma mesma réplica do fragmento, o que implica na escolha de um dos sítios para executar a subconsulta. A escolha do sítio de leitura é feito com base na sua carga de trabalho. A carga de trabalho é definida como a quantidade de consultas que estão sendo executadas pelo sítio num determinado instante.

Para controlar a carga do sistema, o coordenador possui o vetor de carga *vetor de carga* $W = \{w_1, w_2, \ldots, w_n\}$. Cada $w_i \in W$ armazena a carga de trabalho do sítio s_i. Quando uma consulta é submetida a um sítio de leitura s_i, o valor da sua carga w_i é incrementado em uma unidade. Quando um sítio s_i retorna uma resposta ao coordenador, o valor da carga w_i é reduzida em uma unidade. O coordenador escolhe o sítio de menor carga para executar a subconsulta. Caso todos os sítios comparados tenham o

mesmo valor de carga, o sítio é escolhido aleatoriamente. Por fim, o coordenador envia a subconsulta para o sítio de leitura escolhido.

Quando uma consulta de leitura chega a um sítio de leitura, a primeira ação tomada é verificar o valor da variável `status`. Caso o valor da variável seja igual a `desbloqueado`, a consulta de leitura pode ser aplicada à base local do sítio. O passo inicial para a execução da consulta é verificar a consistência do fragmento a ser lido. Se o valor da versão do fragmento requerido pela consulta for inferior à versão do fragmento na base local, implica que os dados do fragmento estão desatualizados. Para atualizar os dados do fragmento, o sítio inicia uma transação de *refresh*. Caso o valor da versão requerida pela consulta seja igual ao valor da versão local, a consulta é executada na base local e o resultado é retornado ao coordenador. Caso o valor da variável `status` seja igual a `bloqueado`, o processo da consulta passa a aguardar uma notificação de desbloqueio. Quando o sítio notificar o desbloqueio para seus processos de leitura que estão aguardando, as consultas em espera são liberadas para execução.

Quando todas as consultas são finalizadas, o coordenador realiza a composição do resultado final a partir dos resultados de cada subconsulta. Esta composição consiste em aplicar uma operação de união entre os resultados preliminares. Caso a consulta original tenha a cláusula `GROUP BY`, o resultado final pode estar incorreto. Neste caso, é aplicada uma consulta com a cláusula `GROUP BY` da consulta original sob o resultado final. Por fim, este resultado é retornado à aplicação cliente.

A transação de *refresh* tem por objetivo atualizar os dados da base de um sítio de leitura. Transações deste tipo são disparadas pelo sítio quando uma consulta requisita a leitura de um dado desatualizado. Esta situação acontece quando existem uma ou mais transações na fila de propagação local Q_L que modificam dados do fragmento a ser lido pela consulta. Uma transação de *refresh* somente pode ser iniciada quando o valor da variável `status` for igual a `desbloqueado`. Caso contrário, o processo da transação fica em espera até que o valor da variável seja `desbloqueado`.

Ao ser iniciada, o primeiro passa da transação de *refresh* é atribuir o valor `bloqueado` à variável `status`. Isto evita que novas transações de propagação e *refresh* sejam iniciadas. Além do mais, impede que consultas de leitura acessem dados inconsistentes. Posteriormente, cada transação da fila de propagação local Q_L é executada até que a versão local do fragmento seja igual à versão requerida pela consulta. As consultas de Q_L são executadas considerando os passos seguidos numa transação de propagação.

Quando os dados do fragmento estão consistentes, o sítio atualiza a variável `status` para o valor `desbloqueado`, notificando os processos que estão aguardando.

3.4 CENÁRIOS

Para ilustrar o funcionamento do RepliXP, foi elaborado um cenário de execução para cada tipo de transação. O ambiente apresentado é composto por um sítio coordenador C01, dois sítios de atualização U01 e U02, e os sítios de leitura R01, R02 e R03. A Figura 3.4 ilustra o cenário inicial da execução.

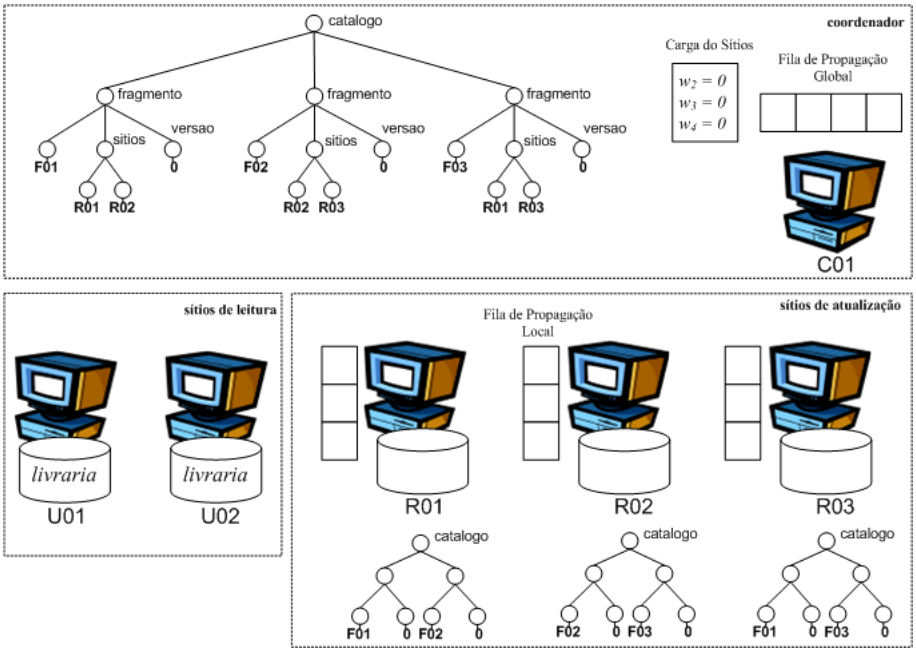


Figura 3.4 Cenario Inicial

A base de dados *livraria* é composta por um conjunto de documentos que seguem a estrutura descrita na Figura 3.5.

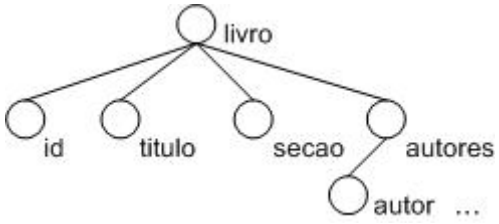


Figura 3.5 Cenario Inicial

Para o exemplo, os fragmentos estão especificados conforme a Tabela 3.1. O fragmento F01 contém todos os documentos cujo elemento *secao* tem o conteúdo igual a *romance*. Por sua vez, o fragmento F02 contém todos os documentos cujo elemento *secao* tem o conteúdo igual a *documentario*. Por fim, o fragmento F03 corresponde aos documentos que não estão em F01 e F02, ou seja, documentos cujo conteúdo do elemento *secao* não possuem os valores *romance* e *documentario*.

Fragmento	**Definição**
$F01$	$\langle livraria, \sigma_{/secao='romance'} \rangle$
$F02$	$\langle livraria, \sigma_{/secao='documentario'} \rangle$
$F03$	$\langle livraria, \sigma_{/secao\neq'romance' \wedge /secao\neq'documentario'} \rangle$

Tabela 3.1 Fragmentos do Cenário

Os sítios de atualização possuem a cópia completa da base original, enquanto que os sítios de leitura possuem fragmentos da base. O sítio U01 corresponde ao sítio primário de atualização, ao passo que U02 é um sítio secundário. Conforme os dados do catálogo global, o sítio R01 possui as coleções dos documentos especificadas por F01 e F02. O sítio R02 contém os documentos definidos pelos fragmentos F02 e F03. Por último, o sítio R03 contém as coleções de documentos definidas em F01 e F03. Inicialmente, cada elemento do vetor de versão global dos fragmentos possui o valor 0. Em relação ao vetor de versão local, o valor da versão é 0 para os fragmentos que estão alocados no sítio. As filas de propagação global e local não possuem transações.

A Figura 3.6 ilustra o cenário da execução de uma transação de atualização T_1.

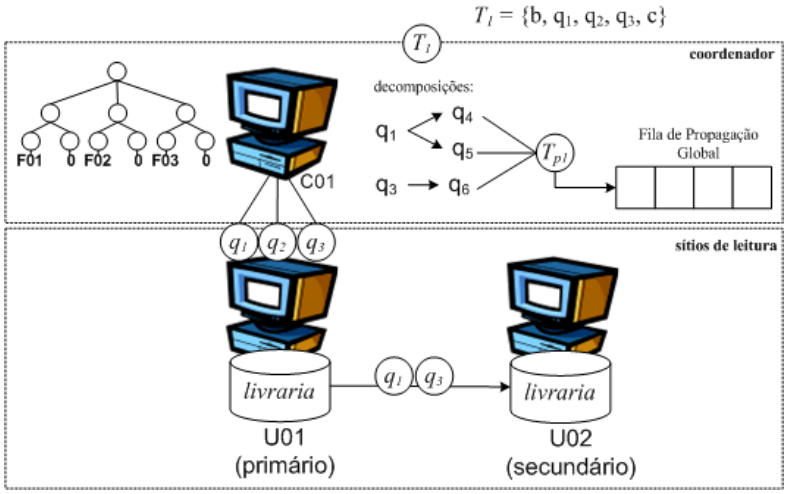


Figura 3.6 Detalhes da Execução de T_1

Ele tem início quando uma aplicação cliente envia a requisição da transação $T_1 = \{b, q_1, q_2, q_3, c\}$, onde cada comando é interpretado sequencialmente. As consultas desta transação estão descritas na Tabela 3.2. Os comandos b e c correspondem às indicações de início e fim da transação, respectivamente.

Consulta	**Valor**
q_1	UPDATE replace $l in collection("livraria")/livro/secao[text()='romance' or text()='documentario']/../preco with <preco>$l*0.85</preco>
q_2	for $l in collection("livraria")/livro/titulo where $l/../autores/autor = 'Paulo Coelho' return $l
q_3	LOAD 'livro.xml' 'livraria'
q_4	UPDATE replace $l in collection("f1")/livro/secao[text()='romance' or text()='documentario']/../preco with <preco>$l*0.85</preco>
q_5	UPDATE replace $l in collection("f2")/livro/secao[text()='romance' or text()='documentario']/../preco with <preco>$l*0.85</preco>
q_6	LOAD 'livro.xml' 'f3'

Tabela 3.2 Consultas da Transação de Atualização T_1

Cada consulta é processada pelo sítio coordenador C01 e repassada para ser executada no sítio primário de atualização, que neste cenário é o sítio U01. Cada consulta de atualização é decomposta pelo coordenador em uma ou mais subconsultas, onde cada uma delas acessa dados de um único fragmento. A consulta q_1 consiste da alteração de dados correspondentes aos fragmentos F01 e F02. Assim, esta consulta é decomposta nas subconsultas q_4 e q_5, onde cada uma delas acessa especificamente um dos fragmentos. A consulta q_3 consiste na inclusão de um novo documento na base de dados. No caso de uma inclusão, a decomposição consiste em reescrever a consulta de tal forma que o documento seja armazenado corretamente em um fragmento específico. O coordenador compara a descrição dos fragmentos com os dados do novo documento para descobrir em qual fragmento a consulta q_3 deve ser executada. Neste caso, esta consulta é reescrita na consulta q_6 para incluir o documento livro.xml no fragmento F03. As consultas q_4, q_5 e q_6 compõem a transação de propagação T_{p1}.

Ao chegarem ao sítio primário U01, as consultas da transação são executadas pelo SGBD local. As consultas de atualização q_1 e q_3 são propagadas para os sítios secundários de forma assíncrona. Após o sítio primário concluir a execução das consultas localmente, o

coordenador adiciona a transação de propagação T_{p1} na fila de propagação global. Além disso, ele atualiza o valor do vetor de versão global de fragmentos, adicionando uma unidade à versão dos fragmentos alterados F01,F02 e F03. As transações de propagação que são constantemente enviadas aos sítios de leitura pelo coordenador. A Figura 3.7 ilustra o cenário final da execução da transação T_1.

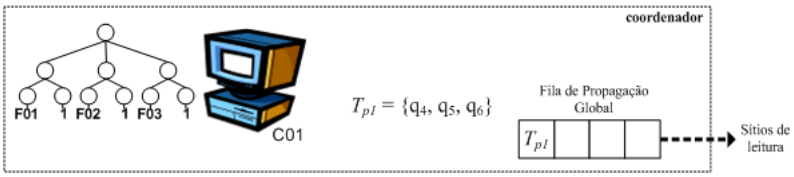
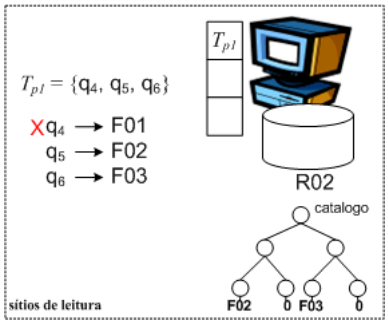
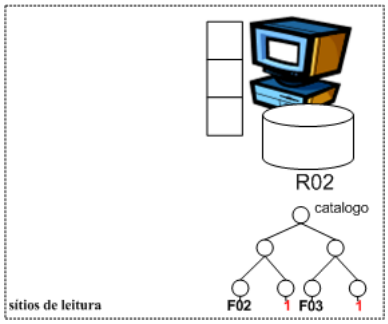


Figura 3.7 Cenário pós-execução da Transação T_1

No momento em que o sítio estiver desbloqueado e nenhuma consulta estiver sendo executada por ele, o sítio de leitura executa as transações que estão presentes na fila de propagação local. A Figura 3.8(a) corresponde ao cenário de execução da transação de propagação T_{p1} no sítio de leitura R02. Após o sítio ser bloqueado, é iniciada a execução da transação $T_{p1} = \{q_4, q_5, q_6\}$. Para cada consulta $q_i \in T_{p1}$, é verificado se o sítio possui o fragmento cujos dados são alterados. No processamento de T_{p1}, o sítio descarta a consulta q_4, a base não possui uma réplica do fragmento F01. As consultas q_5 e q_6 são executadas, pois os dados dos fragmentos F02 e F03 estão alocados na base do sítio R02. Após serem executadas as consultas, a transação é finalizada e a versão local dos fragmentos alterados por T_{p1} é incrementada em uma unidade. A Figura 3.8(b) ilustra o cenário posterior à execução da transação.



(a) Detalhes da Execução de T_{p1}



(b) Cenário pós-execução da Transação T_{p1}

Figura 3.8 Cenários de Execução da Transação de Propagação T_{p1}

O cenário da Figura 3.9 ilustra a execução da transação de leitura $T_2 = \{q_4, q_5, q_6\}$. As consultas desta transação estão descritas na Tabela 3.3. A transação é decomposta

pelo coordenador em um conjunto de subconsultas, de tal forma que cada subconsulta realiza a leitura de dados pertencentes a um único fragmento. Não é necessário decompor a consulta q_7, pois ela já acessa dados contidos em um fragmento específico. No entanto, q_7 é reescrita na consulta q_9 para ser aplicada a um sítio específico. No caso da consulta q_8, ela é decomposta nas subconsultas q_{10} e q_{11}.

Cada subconsulta gerada é encaminhada ao sítio de leitura que possui a réplica do fragmento a ser lido. Caso um fragmento esteja armazenado em mais de um sítio, a consulta é direcionada ao sítio que possui a menor carga de trabalho. Se dois sítios possuem a mesma carga, o sítio é escolhido aleatoriamente. Considerando a subconsulta q_9, os sítios R01 e R02 possuem uma réplica dos dados lidos. Como a carga de trabalho destes sítios é igual, o sítio R01 é escolhido. A subconsulta q_{10} realiza uma leitura de dados do fragmento F01, podendo ser executada nos sítios R01 e R02. Como a carga de R01 é maior que a carga de R02, dado que a subconsulta q_9 está em execução, a subconsulta q_{10} é enviada para o sítio R02. A subconsulta q_{11} é enviada para o sítio R01 por conta que os sítios R01 e R02 estão com cargas iguais.

Consulta	**Valor**
q_7	for $l in collection(”livraria”)/livro where $l/secao = ’documentario’ and $l/vendas > 10000 return count($l)
q_8	for $l in collection(”livraria”)/livro where $l/secao = ’romance’ or $l/secao = ’documentario’ order by $l/preco return $l
q_9	for $l in collection(”f2”)/livro where $l/secao = ’documentario’ and $l/vendas > 10000 return count($l)
q_{10}	for $l in collection(”f1”)/livro where $l/secao = ’romance’ or $l/secao = ’documentario’ order by $l/preco return $l
q_{11}	for $l in collection(”f2”)/livro where $l/secao = ’romance’ or $l/secao = ’documentario’ order by $l/preco return $l

Tabela 3.3 Consultas da Transação de Leitura T_2

Cada consulta que é enviada para um sítio de leitura é associada ao valor da versão global do fragmento a ser lido. Ao chegar ao sítio de leitura, a versão requisitada

pela consulta é comparada à versão local do fragmento. Caso a versão local seja inferior ao valor da versão requisitada, é aplicada uma transação de *refresh* para atualizar os dados do fragmento. Neste cenário, todos os fragmentos estão atualizados. Após o sítio executar a consulta, seu resultado é retornado ao coordenador. Com relação à consulta q_8, seu resultado é composto pela união dos resultados de suas subconsultas q_{10} e q_{11}.

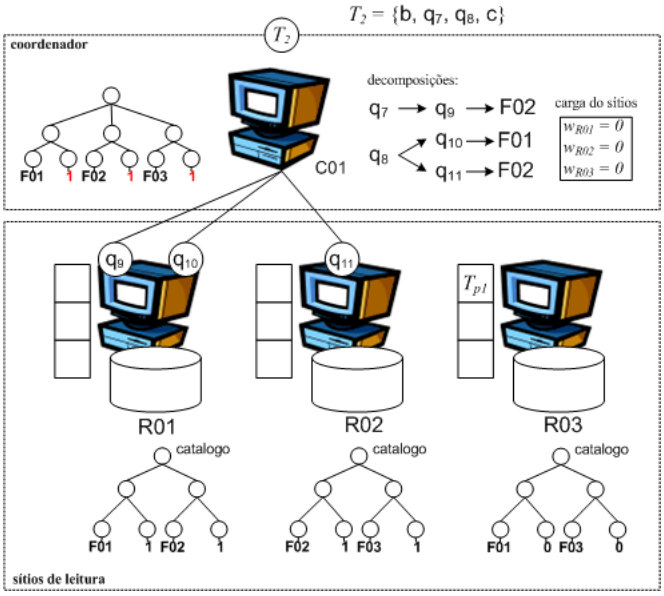


Figura 3.9 Cenário de Execução da Transação de Leitura T_2

A Figura 3.10 ilustra a execução de uma transação de leitura T_3. Esta transação é constituída da consulta q_{12}, que realiza uma leitura sobre os dados do fragmento F03. Esta consulta é reescrita na subconsulta q_{13}, a qual é aplicada à coleção F03. Utilizando o critério de menor valor de carga, a consulta q_{13} é enviada ao sítio R03. Ao ser comparada a versão requisitada pela consulta com a versão local do fragmento, é verificado que seus dados estão desatualizados. Assim, é aplicada uma transação de *refresh* sobre o fragmento F03. Esta transação consiste em executar as transações de propagação presentes na fila local Q_L até que a versão do fragmento solicita pela consulta seja igual à versão local do fragmento. Na execução da consulta q_{13}, a versão local do fragmento é 0 enquanto que a versão requisitada pelo fragmento é 1. O sítio R01 é bloqueado e a transação de propagação T_{p1} é retirada da fila Q_L e executada. Após sua execução, a versão local do fragmento torna-se igual à versão solicitada pela consulta. Com isso, o sítio é desbloqueado e a consulta q_{13} é executada. A transação de *refresh* consiste da composição de todas as transações de propagação executadas durante o bloqueio do sítio.

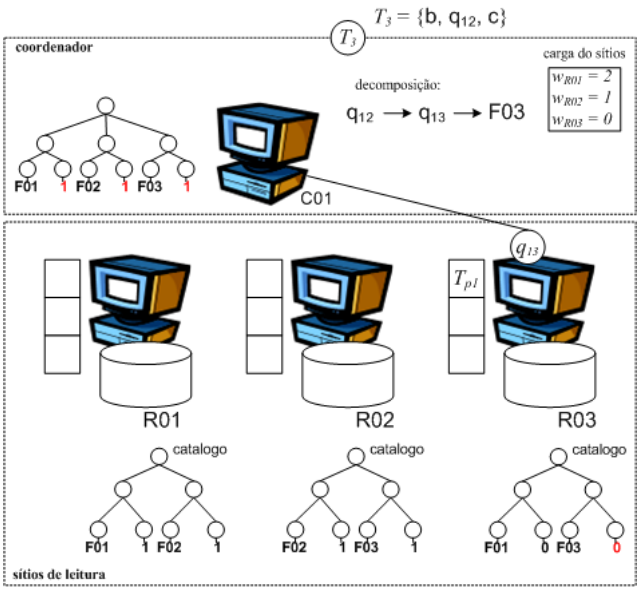


Figura 3.10 Cenário de Execução da Transação de Leitura T_3 com Transação de *Refresh*

3.5 ALGORITMOS

Esta seção descreve os principais algoritmos do RepliXP. O Algoritmo 1 descreve os passos executados pelo sítio coordenador ao processar uma transação solicitada por um cliente. Ao solicitar uma conexão ao coordenador, o cliente informa um valor para o parâmetro booleano *autoCommit*, correspondente ao tipo da transação. Se este parâmetro assumir o valor verdadeiro, trata-se de uma transação de leitura; caso contrário, temos uma transação de atualização. Após o início do processamento da transação, o valor deste parâmetro não pode ser modificado.

Uma transação T é constituída de um conjunto de instruções enviadas sequencialmente ao coordenador. Cada instrução é processada de acordo com seu tipo (linha 1). A instrução pode assumir um dos seguintes tipos: *begin*, *commit* ou uma operação de leitura ou atualização de dados. A primeira instrução da transação corresponde ao *begin*, que indica o início da transação. Com base nesta instrução, é realizado um teste para categorizar transação (linha 3). A instrução *begin* possui a variável *autoCommit*, que indica o tipo da transação a ser iniciada. Se o valor desta variável for $falso$ (linha 4), é submetida ao sítio primário a solicitação de início de uma transação de atualização, por meio do método *beginTransaction* (linha 5).

Quando a instrução é uma operação de dados q_i (linha 8), seu processamento é

condicionado ao tipo de transação ao qual está associada. Se a transação é do tipo atualização, a operação é direcionada ao método *processarOperacaoTransacaoAtualizacao* (linha 10). Caso contrário, o método *processarOperacaoTransacaoLeitura* é executado (linha 12). Em ambos os casos, a variável R recebe o retorno da execução da operação. Ao final do processamento de q_i, se o conteúdo da variável R assumir o valor $ERRO$ (linha 14), implica que houve um erro na execução da operação, tendo como consequência o cancelamento da transação (linha 15).

Algoritmo 1: Processamento da Transação no Sítio Coordenador

Entrada: Transação $T = \{begin, q_1, q_2, \ldots, q_m, commit\}$

```
1  para cada instrucao ∈ T faça
2      selecione instrucao faça
3          caso instrucao = begin
4              se autoCommit = falso então
5                  beginTransaction(T, s_p);
6              fim
7          fim
8          caso instrucao = q_i
9              se autoCommit = falso então
10                 R ← processarOperacaoTransacaoAtualizacao(q_i);
11             senão
12                 R ← processarOperacaoTransacaoLeitura(q_i);
13             fim
14             se R = ERRO então
15                 cancelar(T);
16             fim
17         fim
18         caso instrucao = commit
19             se autoCommit = falso então
20                 para cada f_i ∈ F_t faça
21                     V_i ← V_i + 1;
22                 fim
23                 add(Q_G, T_p);
24                 commitTransaction(T, s_p);
25                 autoCommit ← verdadeiro;
26             fim
27         fim
28     fim
29 fim
```

Por fim, o cliente envia uma instrução de *commit*, indicando que a transação T pode ser finalizada. Neste caso (linha 18), é verificado se a transação é do tipo atualização. Caso esta condição seja satisfeita (linha 19), o vetor de versão global dos fragmentos é atualizada. A variável F_t contém o conjunto de fragmentos alterados pela transação T. A versão de cada fragmento f_i contido em F_t é acrescida em uma unidade (linha 21). Em seguida, a transação de propagação T_p, que corresponde às modificações de dados de T, é adicionada à fila de propagação global Q_G (linha 23). Posteriormente, é submetida uma instrução de fim de transação ao sítio primário, por meio do método *commitTransaction* (linha 21). Por fim, é atribuído o padrão à variável *autoCommit*, que corresponde ao

valor *verdadeiro* (linha 23).

O Algoritmo 2 corresponde ao método $processarOperacaoTransacaoAtualizacao$, executado pelo sítio coordenador ao processar as operações de uma transação de atualização. O primeiro passo deste método é enviar uma solicitação de execução da operação ao sítio primário (linha 1). O retorno da execução é atribuído à variável R. O passo seguinte consiste em verificar se a operação foi executada corretamente. Se o conteúdo da variável R é igual a $ERRO$ (linha 2), implica que a operação não foi executada corretamente. Assim, a mensagem $ERRO$ é retornada pelo método (linha 3). Caso a operação seja executada corretamente, o método verifica se seu tipo é de atualização (linha 5). Se esta condição for verificada, a variável T_p recebe o conjunto de subconsultas referente à decomposição da operação q, resultado do método *decompor* (linha 6). Posteriormente, o conjunto de fragmentos alterados pela operação é unido ao conjunto F_t (linha 7). Por último, o resultado da operação é retornado pelo procedimento (linha 10).

Algoritmo 2: Processamento da Transação de Atualização

Entrada: Operação q
Saída: Resultado da operação

```
1  R ← execute(s_p, q);
2  se R = ERRO então
3  |   retorna ERRO;
4  senão
5  |   se tipo(q) = atualizacao então
6  |   |   T_p ← T_p ⋃ decompor(q);
7  |   |   F_t ← F_t ⋃ fragmentos(q);
8  |   fim
9  fim
10 retorna R;
```

O Algoritmo 3 descreve os passos do sítio primário s_p na execução de uma transação de atualização. Ele recebe como entrada um conjunto de instruções I enviadas seqüencialmente ao sítio primário pelo coordenador. Uma instrução pode ser o comando $beginTransaction$, a execução de uma operação pelo comando $e(q_i)$ ou o comando $commitTransaction$. Para cada tipo de instrução, um conjunto específico de passos é executado. Caso a instrução seja um comando de $beginTransaction$ (linha 3), uma transação T é iniciada no SGBD local pela chamada ao método *begin* (linha 4). No caso da execução de uma operação $e(q_i)$ (linha 6), ela a operação q_i é executada localmente dentro do escopo da transação T, pela chamada ao método *execute*. O resultado da operação q_i é atribuído à variável R (linha 7).

Posteriormente, é verificado se a operação foi executada corretamente. Se o valor de R for igual a $ERRO$ (linha 8), é executado o comando de recuperação *rollback* aplicado

à transação T (linha 9), fazendo com que o SGBD desfaça todas as alterações realizadas pela transação T. Se o valor de R for diferente de $ERRO$, é verificada se a operação q_i é do tipo atualização. Caso essa condição for satisfeita (linha 11), a consulta é adicionada à variável T', que corresponde ao conjunto de consultas de atualização da transação T (linha 12). Se a instrução corresponder ao comando *commitTransaction* (linha 16), o comando *commit* é aplicado à transação T (linha 17), fazendo com que todas as alterações desta transação sejam consolidadas na base local. Após este passo, as consultas de atualização presentes em T' são propagadas aos sítios de atualização secundários de forma assíncrona.

Algoritmo 3: Processamento da Transação de Atualização no Sítio Primário

Entrada: Conjunto de instruções $I = \{beginTransaction, e(q_1), e(q_2), \ldots, e(q_m), commitTransaction\}$
Saída: Mensagem de *SUCESSO* ou *ERRO*

```
1  para cada cmd ∈ I faça
2      selecione cmd faça
3          caso cmd = beginTransaction
4              T.begin();
5          fim
6          caso cmd = e(q_i)
7              R ← T.execute(q_i);
8              se R = ERRO então
9                  T.rollback();
10             senão
11                 se tipo(q_i) = atualizacao então
12                     T' ← T' ⋃ q_i;
13                 fim
14             fim
15         fim
16         caso cmd = commitTransaction
17             T.commit();
18             propagar T' para sítios secundários;
19         fim
20     fim
21 fim
```

As transações de propagação que chegam à fila Q_G são continuamente enviadas aos sítios de leitura. Ao chegar a um sítio de leitura, a transação de propagação T_p é incluída na fila de propagação local Q_L deste sítio. Existe um processo do sítio de leitura que verifica continuamente se o sítio está ocioso, ou seja, se o sítio não está executando alguma transação de leitura. Se o sítio estiver ocioso, as transações de propagação presentes na fila local Q_L são executadas na sequência em que chegaram ao sítio. O Algoritmo 4 descreve os passos desta execução.

Inicialmente, o sítio s é bloqueado para evitar que novas transações de leitura sejam processadas pelo sítio (linha 1). Posteriormente, o procedimento verifica se existe alguma transação na fila de propagação local Q_L (linha 2). Se o tamanho de Q_L for maior que zero, as transações presentes na fila local são processadas. O próximo passo consiste em atribuir à variável *tam* a quantidade de transações de propagação presentes em Q_L

(linha 3). O próximo passo consiste de uma estrutura de repetição, iniciando do valor 1 até *tam*, contendo os procedimentos para execução de cada transação de propagação (linha 4).

Algoritmo 4: Processamento das Transações da Fila de Propagação Local

```
   Entrada: Fila Q_L
 1 bloquear(s);
 2 se (length(Q_L) > 0) então
       // considera apenas os elementos que estavam na fila Q_L naquele instante
 3     tam ← length(Q_L);
 4     para cada i ← 1, ..., tam faça
 5         T_p ← next(Q_L);
 6         T_p.begin();
 7         para cada q ∈ T_p faça
 8             f ← fragmento(q);
 9             se f ∈ s então
10                 R ← execute(q);
11                 se R = ERRO então
12                     T_p.rollback();
13                     desbloquear(s);
14                 senão
15                     F_p ← F_p ⋃ f;
16                 fim
17             fim
18         fim
19         T_p.commit();
20         para cada f_i ∈ F_p faça
21             v_i ← v_i + 1;
22         fim
23     fim
24 fim
25 desbloquear(s);
```

A cada laço da repetição, uma transação é retirada da fila pela chamada ao método *next* e atribuída à variável T_p (linha 5). A transação T_p tem início quando o procedimento faz chamada ao método *begin*(), que inicia uma nova transação no SGBD local (linha 6). Cada operação q pertencente a T_p é analisada (linha 7), com o objetivo de verificar se o fragmento alterado pela operação está armazenado no sítio s. Assim, o procedimento verifica para qual fragmento é aplicada a operação q, atribuindo seu valor à variável f (linha 8). Se o fragmento f não esteja armazenado no sítio s, a operação q é descartada. Caso contrário (linha 9), a operação é executada pelo sítio s e seu resultado é atribuído à variável R (linha 10).

Posteriormente, é verificado o resultado da execução de q. Se a variável R assumir o valor *ERRO* (linha 11), é feita uma chamada ao método *rollback* referente à transação T_p (linha 13). Este comando faz com que o SGBD local desfaça todas as alterações aplicadas pela transação. Além do mais, é feita uma chamada ao método *desbloquear*, que libera o sítio para a execução de novas operações de leitura (linha 13). Caso a consulta seja executada corretamente, o fragmento f é incluído na variável F_p (linha 15), a qual

armazena todos os fragmentos alterados pela transação T_p.

Ao serem aplicadas corretamente todas as operações de T_p ao sítio local, a transação é finalizada pela chamada do método *commit* (linha 19). Este método faz com que todas as alterações da transação T_p sejam consolidadas. Uma vez finalizada a transação, o procedimento atualiza o valor do vetor local de versão dos fragmentos V_L. Para isso, é acrescentada uma unidade ao valor da versão local de cada fragmento f_i contido na variável F_p (linhas 20 e 21). Ao final da execução de todas as transações de propagação, o sítio é desbloqueado para que as operações de leitura possam ser executadas (linha 25). Ao ser chamado o método *desbloquear*, uma notificação é enviada a todas as requisições que estão aguardando o desbloqueio.

O Algoritmo 5 corresponde ao método $processarConsultaTransacaoLeitura$ executado pelo sítio coordenador para processar cada consulta q de uma transação de leitura. A consulta q pode realizar a leitura em dois ou mais fragmentos. Com isso, o primeiro passo desse procedimento é decompor a consulta q em um conjunto de subconsultas, onde cada uma delas é aplicada a um único fragmento. Esta decomposição é feita pela chamada do método *decompor*, cujo retorno é atribuído à variável T' (linha 1). O próximo passo consiste de um laço para realizar um conjunto de passos para cada subconsulta q' contida em T' (linha 2).

Algoritmo 5: Processamento da Transação de Leitura no Sítio Coordenador

Entrada: Consulta q
Saída: Resultado da consulta

```
1  T' ← decompor(q);
2  para cada q' ∈ T' faça
3  |   f_i ← fragmento(q');
4  |   s ← menorCarga(f_i);
5  |   aumentarCarga(s);
6  |   r ← execute(s, q', V_i);
7  |   diminuirCarga(s);
8  |   se r = ERRO então
9  |   |   retorna ERRO;
10 |   senão
11 |   |   R ← R⋃r;
12 |   fim
13 fim
14 retorna R;
```

Para cada q', o primeiro passo é descobrir qual fragmento é lido por esta consulta. Isto é feito pela chamada ao método $fragmento$, cujo retorno é atribuído à variável f_i (linha 3). O passo seguinte consiste em descobrir qual o sítio de leitura que possui o fragmento f_i está com a menor carga naquele momento. É feita uma chamada ao método $menorCarga$, que consulta a carga dos sítios de leitura e retorna a informação do sítio

com menor carga. Este retorno é atribuído à variável s (linha 4). Posteriormente, é adicionada uma unidade à carga do sítio s (linha 5) e a consulta é submetida a este sítio. Juntamente com a consulta, é enviado o valor da versão global do fragmento ao qual ela é aplicada. Ao receber o resultado da consulta, ele é atribuído à variável r (linha 6). Ao ser finalizada a execução da consulta, a carga do sítio s é reduzida em uma unidade (linha 7). O próximo passo consiste do teste para verificar se a consulta foi executada corretamente. Se o retorno da consulta r for igual a $ERRO$ (linha 8), a execução da consulta é finalizada e o processamento retorna a mensagem $ERRO$ (linha 9). Caso a consulta seja executada corretamente, seu resultado é unido aos resultados das outras subconsultas de q na variável R (linha 11). Ao fim da execução de todas as subconsultas q', o resultado final R correspondente à consulta de q é retornado (linha 14).

Cada sítio de leitura possui um controle quanto ao bloqueio para operações de leitura. Assim, as consultas que chegam a um sítio de leitura somente são executadas quando o mesmo encontra-se desbloqueado. Caso uma consulta seja encaminhada a um sítio de leitura bloqueado, a consulta somente será executada quando o sítio for desbloqueado. A execução de cada consulta que chega a um sítio de leitura é descrita pelo Algoritmo 6.

Algoritmo 6: Processamento da Consulta de Leitura no Sítio de Leitura

Entrada: Consulta de leitura q, Versão Global do Fragmento V_i
Saída: Resultado R

```
1 se v_i < V_i então
2 |   refresh(f_i, V_i);
3 fim
4 R ← executar(q);
5 retorna R;
```

O primeiro passo é comparar a versão local do fragmento f_i com a versão exigida pela consulta (linha 1). Caso a versão local seja menor que a versão requisitada na consulta, é feita uma chamada ao método $refresh$. Este método consiste em atualizar o fragmento para que a consulta possa ser executada. Assim, o método $refresh$ implica na execução de uma transação de *refresh*, que atualiza os dados da base local até que a versão local do fragmento f_i seja igual à versão solicitada pela consulta (linha 2). Após a atualização do fragmento, a consulta é executada e seu resultado é atribuído a variável R (linha 4). Por fim, o resultado R é retornado (linha 5).

Uma transação de *refresh* consiste em atualizar a base de dados para que uma operação de leitura possa ser executada sob um fragmento que se encontra desatualizado. Esta atualização é realizada por executar as transações de propagação contidas na fila de

propagação local do sítio de leitura até que a versão local do fragmento a ser atualizado seja igual à versão solicitada pela consulta. O Algoritmo 7 descreve os passos executados por um sítio de leitura ao executar uma transação de *refresh*.

Algoritmo 7: Processamento da Transação de Refresh no Sítio de Leitura

Entrada: Versão Global do Fragmento V_i

```
1  bloquear(s);
2  v' ← v;
3  T_r.begin();
4  repita
5  |   T' ← next(Q_L);
6  |   para cada q ∈ T_p faça
7  |   |   f ← fragmento(q);
8  |   |   se f ∈ s então
9  |   |   |   R ← execute(q);
10 |   |   |   se R = ERRO então
11 |   |   |   |   T_r.rollback();
12 |   |   |   |   desbloquear(s);
13 |   |   |   senão
14 |   |   |   |   F' ← F' ⋃ f;
15 |   |   |   fim
16 |   |   fim
17 |   fim
18 |   para cada f_i ∈ F' faça
19 |   |   v'_i ← v'_i + 1;
20 |   fim
21 até v_i < V_i ;
22 T_r.commit();
23 v ← v';
24 desbloquear(s);
```

O primeiro passo é realizar o bloqueio do sítio, para evitar que novas consultas ou transações de *refresh* sejam iniciadas 1). Posteriormente, a variável v' recebe uma cópia do vetor de versão local de fragmentos v (linha 2), a qual será alterado durante a execução da transação. Em seguida, a transação de *refresh* T_r é iniciada no SGBD local ao executar o método *begin* (linha 3). A partir de então, é iniciado uma estrutura de repetição que executa as transações de propagação da fila local Q_L até que a versão local do fragmento f_i se iguale à versão requisitada (linha 4).

Para cada iteração do laço, a próxima transação de propagação de Q_L é atribuída à variável T' (linha 5). Em seguida, as operações de atualização q contidas em T' são processadas sequencialmente (linha 6). O processamento de cada operação q inicia-se por atribuir à variável f o fragmento a ser alterado (linha 7). Posteriormente, é verificado se o sítio possui uma cópia dos seus dados (linha 8). Caso esta condição seja satisfeita, a operação é executada no SGBD local dentro do escopo da transação T_r ao ser chamado o método *execute* (linha 9). O resultado da operação é atribuído à variável R. O passo seguinte verifica se a operação foi executada corretamente ao comparar o valor de R com

o valor *ERRO* (linha 10). Caso a comparação seja verdadeira, é executado o método *rollback* (linha 11), fazendo com que o SGBD local desfaça todas as alterações aplicadas anteriormente. Com isso, o sítio é desbloqueado pela chamada do método *desbloquear* (linha 12) e o processamento da transação é cancelado.

Caso a operação tenha sido executada corretamente, o fragmento alterado por q é incluído no conjunto F' (linha 14). Esta variável armazena os fragmentos alterados pela transação T'. Ao final da execução das operações de T', é acrescida uma unidade à versão do vetor v' para cada fragmento f_i contido em F' (linha 19). Ao se verificar que a versão local do fragmento é igual a versão requisitada, é feita uma chamada ao método *commit* no escopo de T_r (linha 22). Esta operação solicita que o SGBD local consolide todas as modificações aplicadas pela transação de *refresh* T_r. Após consolidadas as modificações, o vetor v recebe os valores de v' (linha 23) e o sítio é desbloqueado pela chamada do método *desbloquear* (linha 24).

O Algoritmo 8 corresponde ao método *decompor*, o qual é utilizado para decompor as operações de leitura e atualização dos dados, haja vista que toda operação pode ser aplicada a dois ou mais fragmentos. A decomposição de uma operação q é realizada considerando a definição dos fragmentos da base original.

Algoritmo 8: Decomposição da Operação

Entrada: Consulta $q = \{P = \{p_1, p_2, \ldots, p_m\}, E = \{e_1, e_2, \ldots, e_n\}\}$

```
1  Seja P_{f_i} os predicados do fragmento f_i;
2  para cada p_j ∈ P faça
3  |   se p_j ∈ P_{f_i} então
4  |   |   F_q ← F_q ⋃ f_i;
5  |   fim
6  fim
7  para cada e_l ∈ E faça
8  |   se e_l é pré-fixada por algum P ∈ P_{f_i} então
9  |   |   F_q ← F_q ⋃ f_i;
10 |   fim
11 fim
12 se F_q = ∅ então
13 |   F_q ← F;
14 fim
15 para cada f ∈ F_q faça
16 |   Criar uma operação q' com o conteúdo de q, modificando a coleção da cláusula LET para f;
17 |   Sub ← Sub ⋃ q';
18 fim
19 retorna Sub;
```

O Algoritmo 8 tem como entrada uma operação q, que é composta por um conjunto de predicados de seleção P e um conjunto de expressões de caminho E. Cada predicado do conjunto P é comparado aos predicados dos fragmentos P_f (linha 2). Se

o predicado da operação corresponde ao predicado de um fragmento f_i, então este fragmento é utilizado pela operação q e é incluído ao conjunto F_q (linha 4). Por conseguinte, cada expressão de caminho do conjunto E é comparada às expressões de caminho dos predicados de fragmentos P_f, considerando apenas o início dos predicados (linhas 7 e 8). Se a comparação for verdadeira para um fragmento f_i, então este fragmento é utilizado pela operação q e é incluído ao conjunto F_q (linha 9)

Ao final desta fase, se o conjunto F_q estiver vazio (linha 12), implica que a operação deve ser aplicada a todos os fragmentos de F. Neste caso, é atribuído ao conjunto F_q todos os elementos de F (linha 13). Por fim, para cada fragmento f_i pertencente ao conjunto F_q (linha 15), é criada uma operação q' específica para este fragmento (linhas 16 e adicionada à variável Sub (linha 17). Ao final, o conjunto de operações Sub é retornado (linha 19).

3.6 CONCLUSÃO

Neste capítulo, apresentou-se o RepliXP como um mecanismo para prover a replicação parcial de base de dados XML, cujo objetivo é melhorar o desempenho desses sistemas provendo a replicação parcial de dados. Inicialmente, foram apresentadas as principais características da solução e suas diferentes em relação aos trabalhos relacionados. A seguir, foi descrita sua especificação, um conjunto de cenários de execução e os principais algoritmos.

CAPÍTULO 4

IMPLEMENTAÇÃO E AVALIAÇÃO

Este capítulo descreve os componentes do RepliXP e apresenta a avaliação adotada, a qual compara o mecanismo descrito neste trabalho utilizando a replicação parcial e a replicação total de dados XML. Inicialmente, são explorados os detalhes de implementação do RepliXP e, posteriormente, a metodologia aplicada na sua avaliação, considerando aspectos de desempenho aplicados na avaliação de banco de dados distribuídos.

4.1 ASPECTOS DE IMPLEMENTAÇÃO

O RepliXP preserva a maioria das características de implementação adotadas pelo trabalho precursor. Sua plataforma de desenvolvimento é centrada na linguagem Java e tecnologias associadas [48]. Características como portabilidade, reusabilidade, facilidades de processamento distribuído e *multithreading* fizeram com que esta tecnologia fosse mantida. A portabilidade consiste na característica de um sistema ser independente quanto sua a plataforma de execução. A plataforma Java é formada por componentes e APIs que pode ser incorporados às aplicações, aumentando a produtividade e, consequentemente, diminuindo o tempo de desenvolvimento. As tecnologias associadas à linguagem Java oferecem diversos recursos para a comunicação entre dispositivos, tais como comunicação entre processos (*socket*), distribuição de objetos (RMI) e troca de mensagens via os protocolos padrões da internet. Java possui recursos de programação *multithreading*, os quais utilizam primitivas de sincronização baseadas no uso de monitores. Eles proporcionam um ambiente fácil e seguro para o desenvolvimento de aplicações multitarefas.

Em relação à forma de comunicação entre os sítios, o RepliXP manteve a estratégia de acesso remoto a objetos distribuídos utilizando a especificação Java RMI. Esta tecnologia permite que objetos sejam publicados em um servidor e acessados remotamente, podendo ser feitas chamadas remotas aos seus métodos. O RMI possui um alto nível de abstração, permitindo que o acesso a objetos remotos seja feito de forma fácil e transparente. O RepliXP utiliza o RMI para criar o canal de comunicação entre

dois dispositivos. O dispositivo destino possui um objeto que é publicado localmente e acessado via conexão remota pelo dispositivo origem.

Em algumas situações no RepliXP, é necessário que uma mensagem seja entregue a um conjunto de máquinas de forma atômica, garantindo a ordem de envio das mensagens. Optou-se por utilizar uma estratégia baseada no protocolo 2 Phase-Commit (2PC) [27], ao contrário de algum mecanismo de comunicação em grupo. Esta decisão foi tomada pelo fato do RepliXP ser especificado para ambientes de cluster, onde a comunicação entre os sítios é confiável. A estratégia utilizada permite um melhor desempenho que a utilização de uma estratégia de comunicação em grupo. Além do mais, a tolerância a falhas não é o foco deste trabalho.

O RepliXP foi implementado estendendo-se o código do RepliX [15], adicionando-se novas funcionalidades para permitir a replicação parcial de dados XML. O RepliXP é constituído de alguns componentes de *software*, os quais são descritos em nível de implementação. Esta descrição consiste do comentário das classes de cada componente e serviços associados. Os métodos de acesso aos atributos dos objetos (*get* e *set*) foram omitidos dos comentários e das figuras que ilustram os diagramas de classes.

RepliXPDriver

O RepliXPDriver é uma implementação da especificação do *driver* JDBC [49], adaptado ao contexto distribuído e às necessidades do RepliXP. Este componente permite a uma aplicação cliente conectar-se ao RepliXPCoordinator, assim como criar transações e submeter consultas para serem executadas. A Figura 4.1 apresenta o diagrama de classes referente ao RepliXPDriver.

As conexões entre os componentes do RepliXP são feitas pela realização das interfaces `RemoteDriverManager` e `RemoteConnection`. A interface `RemoteDriverManager` tem por objetivo criar uma conexão com o componente destino. Ela possui o método `getConnection`, que retorna um objeto do tipo `RemoteConnection`. Este objeto representa a referência a um objeto remoto localizado no componente conectado. A interface `RemoteConnection` possui os métodos necessários para a execução de uma transação. Ambas as interfaces estendem a interface Remote do RMI, possibilitando que objetos desse tipo sejam acessados remotamente. Cada componente possui uma implementação de cada uma destas interfaces, definindo o comportamento particular de cada método.

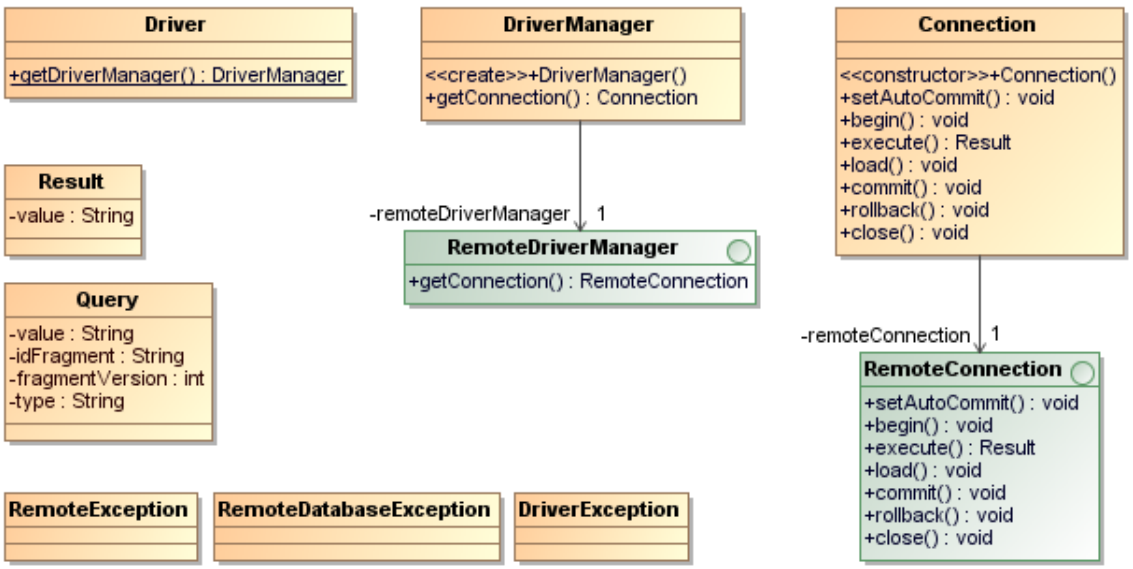


Figura 4.1 Diagrama de classes do RepliXPDriver

O método `begin` cria uma nova transação no componente conectado. O método `setAutoCommit` recebe como parâmetro um valor booleano, o qual é atribuído à variável `autoCommit` da conexão remota. Se o valor for verdadeiro, a transação consolida as modificações na base de dados após o final da execução de uma consulta de atualização. Caso o valor seja falso, todas as modificações de uma transação são consolidadas explicitamente pela aplicação cliente, após a execução do método `commit`.

O método `execute` recebe como parâmetro um objeto do tipo `Query`, que corresponde à consulta a ser executada. A classe `Query` representa uma abstração para uma consulta ou inserção de um documento XML na base. O atributo `value` representa a consulta ou o conteúdo do documento a ser armazenado. Os atributos `idFragmento` e `fragmentVersion` correspondem ao identificador do fragmento e ao valor da versão do fragmento requerida pela consulta. O atributo `type` armazena o tipo da consulta, o qual pode assumir os valores `READ`, `UPDATE` e `LOAD`. O método `execute` retorna um objeto do tipo `Result`. A classe `Result` representa uma abstração para o retorno de uma consulta. O atributo `value` é um objeto do tipo `String` que armazena o resultado de uma consulta. O método `load` recebe como parâmetro um objeto do tipo `Query`. O método `rollback` desfaz todas as alterações aplicadas em uma transação e o método `close` finaliza a conexão.

A classe `Driver` permite à aplicação cliente criar uma instância de `DriverManager`.

O método estático `getDriverManager` recebe por parâmetro o endereço e porta do servidor. Este método monta a URI de conexão RMI e realiza uma chamada remota ao objeto `CoordinatorDriverManager` do componente RepliXPCoordinator. Este objeto remoto é encapsulado em uma nova instância da classe `DriverManager`, que é retornada para o cliente.

A classe `DriverManager` encapsula a interface `RemoteDriverManager`. O objetivo do encapsulamento é abstrair do cliente a forma como é concebida a comunicação com o RepliXP. Essa abstração permite que o mecanismo de comunicação, que atualmente é o RMI, possa ser modificado sem que a aplicação cliente necessite de alterações. Além do mais, estas classes tratam as exceções que podem ser disparadas pelo componente conectado, repassando apenas exceções do RepliXP (`DriverException`, `RemoteException` e `RemoteDatabaseException`). O método `getConnection` da classe `DriverManager` retorna uma nova instância da classe Connection. Esta instância contém uma referência ao objeto remoto `CoordinatorDriverManager`.

A classe `Connection` encapsula a interface `RemoteConnection`. Ela possui os mesmos métodos da interface que encapsula. Cada método desta classe realiza uma chamada ao método correspondente no objeto remoto `CoordinatorDriverManager`. Os métodos `execute` e `load` realizam atividades adicionais antes de realizar a chamada ao objeto remoto.

O método `execute` recebe por parâmetro uma `String` com a consulta a ser executada. É criada uma instância da classe `Query` atribuindo o conteúdo da consulta ao atributo `value`. O atributo `type` recebe o valor `READ` caso a consulta seja uma leitura ou `UPDATE`, caso contrário. A instância de `Query` é passada como parâmetro ao ser chamado o método execute da instância remota `CoordinatorDriverManager`. O resultado da execução remota da consulta é retornado ao cliente.

O método `load` recebe por parâmetro o caminho físico do documento XML a ser armazenado. O documento é carregado em memória e seu conteúdo é convertido em uma String utilizando a ferramenta dom4j [50]. Uma nova instância da classe `Query` é criada e o conteúdo do documento XML é atribuído ao seu atributo `value`. Oa atributo `type` dessa instância é atribuido o valor `LOAD`. A instância de `Query` é passada por parâmetro ao método `load` da instância remota `CoordinatorDriverManager`.

A Figura 4.2 ilustra o diagrama de sequência correspondente à efetivação de uma conexão entre cliente e o componente RepliXPCoordinator. O cliente faz uma chamada

ao método estático `getDriverManager` da classe `Driver` (chamada 1). Este método cria um objeto do tipo `DriverManager` (chamada 2). O construtor desta classe realiza uma chamada remota (`lookup`) ao componente RepliXPCoordinator, que retorna a referência de um objeto remoto do tipo `CoordinatorDriverManager` (chamada 3).

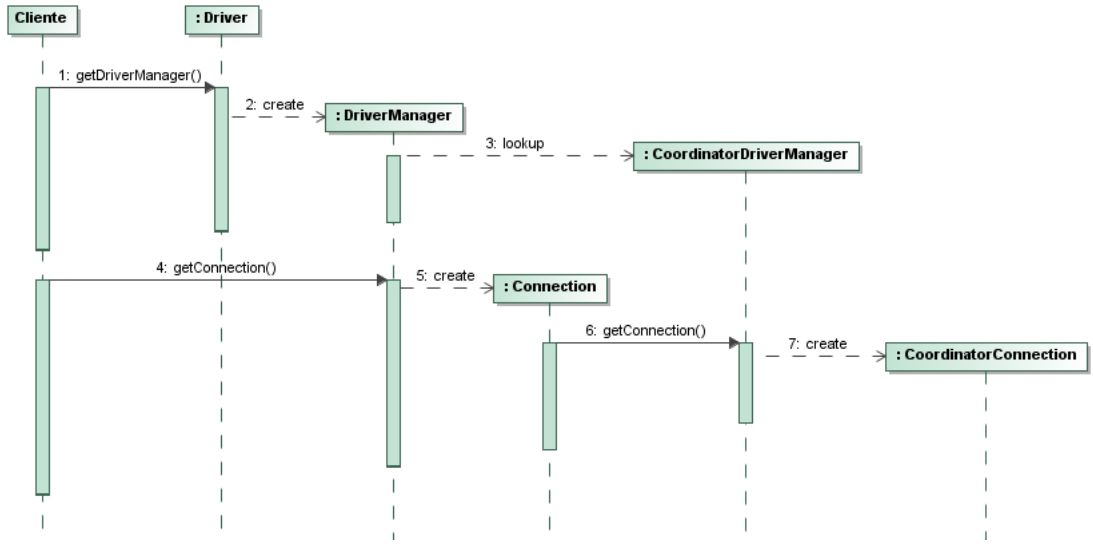


Figura 4.2 Diagrama de seqüência do RepliXPDriver

Posteriormente, o cliente realiza uma chamada ao método `getConnection` da instância da classe `DriverManager` (chamada 4). Este método cria um objeto do tipo `Connection` (chamada 5). O construtor desta classe realiza uma chamada ao método `getConnection` do objeto remoto do tipo `CoordinatorDriverManager` (chamada 6), que retorna uma referência a uma nova instância remota do tipo `CoordinatorConnection` (chamada 7).

RepliXPCoordinator

O RepliXPCoordinator é o componente que gerencia o ciclo de execução das transações submetidas por uma aplicação cliente. A Figura 4.3 mostra o diagrama de classes desse componente. O componente RepliXPCoordinator possui uma classe que serve para iniciar sua execução pela chamada ao método `main`. Este método instancia um objeto do tipo `CoordinatorDriverManager` e o publica como objeto remoto utilizando RMI.

ref

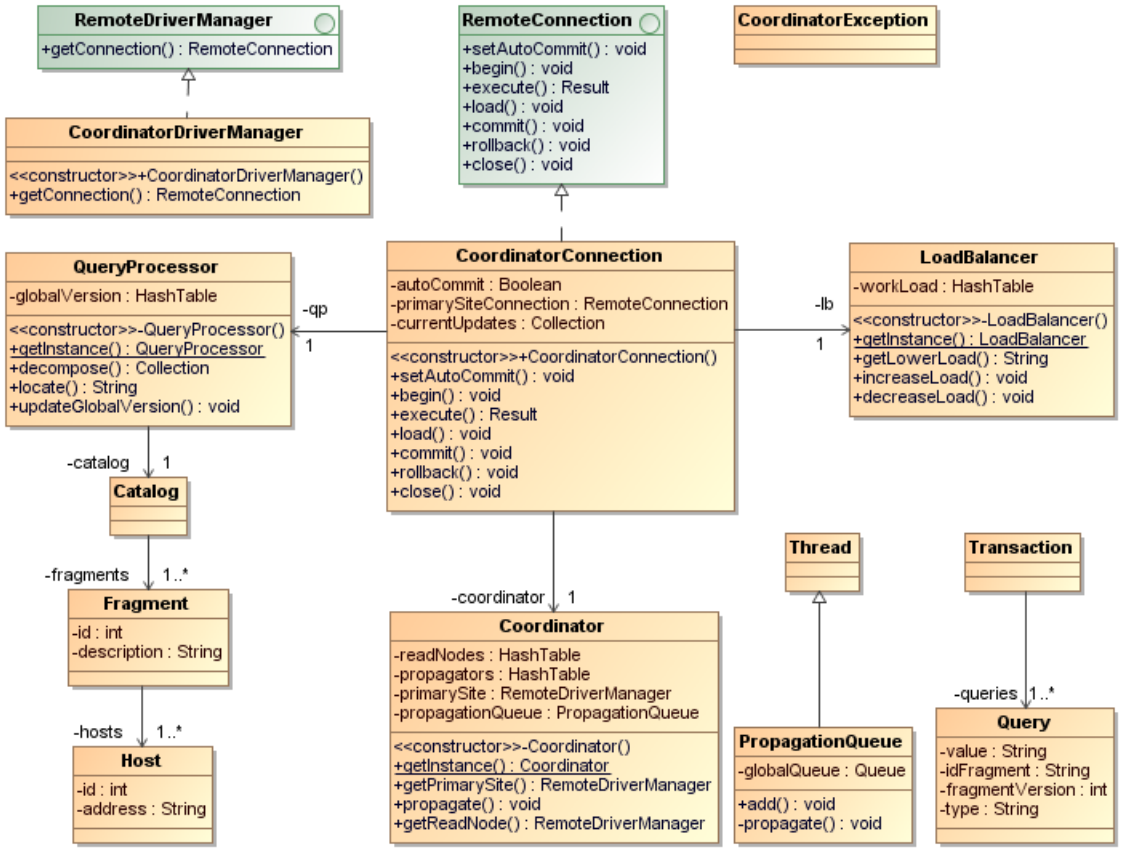


Figura 4.3 Diagrama de classes do RepliXPCoordinator

Este objeto remoto constitui o meio de comunicação inicial entre os clientes e o componente RepliXPCoordinator. Em cada uma das conexões, a instância local da classe `DriverManager` está associada a uma instância remota de `CoordinatorDriverManager`. O método `main` também instancia um novo objeto da classe `Coordinator`, que controla todo o fluxo de execução das transações. Para garantir que apenas uma instância da classe `Coordinator` seja criada, foi aplicado o padrão de projeto Singleton [51]. A classe define um construtor privado e o método estático `getInstance` controla a instanciação do objeto. Se a instância do objeto não existir, este método faz uma chamada ao construtor privado da classe e retorna uma nova instância do tipo `Coordinator`. Caso contrário, é retornado o objeto existente.

A classe `Coordinator` possui os atributos `readNodes` e `propagators`, ambos do tipo `HashTable`. O `HashTable` é uma estrutura Java que representa uma lista de elementos, onde cada elemento corresponde a um valor é associado a uma chave. No caso dos atributos `readNodes` e `propagators`, a chave corresponde a um valor `String` que iden-

tifica unicamente um sítio no sistema e ao valor é atribuída a referência remota daquele sítio. O atributo `readNodes` contém a referência ao objeto remoto `RemoteDriverManager` de cada sítio de leitura. O atributo `propagators` armazena as referências remotas `RemotePropagationManager` de cada sítio de leitura. As informações do identificador e o endereço de cada sítio são passados ao componente por um arquivo XML de configuração. O atributo `primarySite` armazena a referência ao sítio primário de atualização.

O `propagateQueue` é um atributo do tipo `PropagateQueue` e corresponde à fila de propagação global. A classe `PropagateQueue` possui o atributo `globalQueue` do tipo java `Queue`, o qual armazena as transações de propagação. Ao ser instanciado o objeto do tipo `PropagateQueue`, um subprocesso java (*thread*) é iniciado, passando a propagar aos sítios de leitura as transações que são adicionadas à fila `globalQueue`. Uma transação é uma instância da classe `Transaction`.

A classe `Coordinator` também possui os métodos `getPrimarySite` e `propagate`. O método `getPrimarySite` retorna a referência do sítio primário de atualização, que está armazenado na variável privada `primarySite`. O método propagate recebe como parâmetro um objeto do tipo `Transaction`, o qual corresponde à transação de propagação. Este método envia a transação passada por parâmetro a cada sítio de leitura, que é armazenada na sua fila de propagação local. A classe `Transaction` possui apenas o atributo `queries`, que corresponde a uma lista de objetos do tipo `Query`.

Para obter uma conexão remota do componente RepliXPCoordinator, o cliente realiza uma chamada ao método `getConnection` da classe `CoordinatorDriverManager`. Este método retorna uma instância da classe `CoordinatorConnection`. Esta classe é responsável por processar as transações submetidas pelos clientes. O atributo `autoCommit` armazena um valor booleano. O objeto `primarySiteConnection` é um atributo do tipo `RemoteConnection` e representa a conexão corrente com o sítio primário. O atributo `currentUpdates` corresponde à lista de consultas de atualização da transação corrente. A classe `CoordinatorConnection` tem acesso a uma instância das classes `QueryProcessor` e `LoadBalancer`. Ambas as classes adotam o padrão Singleton [51] para garantir a instanciação única de objetos.

A classe `QueryProcessor` realiza o processamento das operações e a manipulação do catálogo de dados. O objeto `globalVersion` é um atributo do tipo `HashTable`, que armazena a versão global de cada fragmento. A classe `QueryProcessor` possui o método `updateGlobalVersion`, que recebe como parâmetro a lista de fragmentos que foram alte-

rados por uma transação de atualização. Este método atualiza o atributo `globalVersion`, incrementando em uma unidade o valor da versão dos fragmentos alterados. Outro atributo da classe `QueryProcessor` é o `catalog`, instância do tipo `Catalog`. Esta classe armazena as informações de definição dos fragmentos e alocação de suas réplicas. Ela possui o atributo `fragments`, que corresponde a uma lista de instâncias da classe `Fragment`. Cada instância de `Fragment` possui um identificador e a definição do fragmento, representados pelos atributos `id` e `description`, respectivamente. Outro atributo de uma instância de `Fragment` é o objeto `hosts`, que consiste de uma lista de instâncias da classe `Host`. Esta lista corresponde aos sítios nos quais o fragmento está armazenado. Cada instância de `Host` possui um identificador e o endereço do sítio, representados pelos atributos `id` e `address`, respectivamente.

A classe `LoadBalancer` é responsável por controlar a carga de trabalho em cada sítio de leitura. O atributo `workLoad` do tipo `HashTable` armazena a carga de trabalho de cada um desses sítios. A carga de trabalho consiste de um valor que representa a quantidade de operações que estão sendo executadas em um determinado sítio. O método `getLowerLoad` recebe um conjunto de sítios e retorna a informação de qual sítio possui a menor carga de trabalho naquele instante. Os métodos `increaseLoad` e `decreaseLoad` recebem como parâmetro o identificador de um sítio. O método `increaseLoad` incrementa em uma unidade o valor da carga do sítio informado enquanto que o método `decreaseLoad` diminui em uma unidade sua carga.

Todas as exceções do componente são encapsuladas por uma exceção específica do componente, instância da classe `CoordinatorException`. As duas figuras a seguir ilustram os diagramas de classes dos fluxos da execução de uma transação de atualização e de leitura. A Figura 4.4 refere-se ao fluxo da transação de atualização.

Para dar início à transação de atualização, o cliente realiza uma chamada ao método `begin` da instância da classe `Connection` (chamada 1). Este método apenas realiza uma chamada ao método `begin` da instância de `CoordinatorConnection` (chamada 2). Este método localiza o sítio primário de atualização ao executar o método `getPrimarySite`, disponível na instância única da classe `Coordinator` (chamada 3). De posse da informação do sítio primário, é criada uma instância remota da classe `UpdateNodeConnection` (chamada 4) e iniciada a transação (chamada 5).

Cada consulta da transação é submetida passada por parâmetro ao método `execute` da classe `Connection` (chamada 6). Este método apenas repassa a consulta como parâmetro

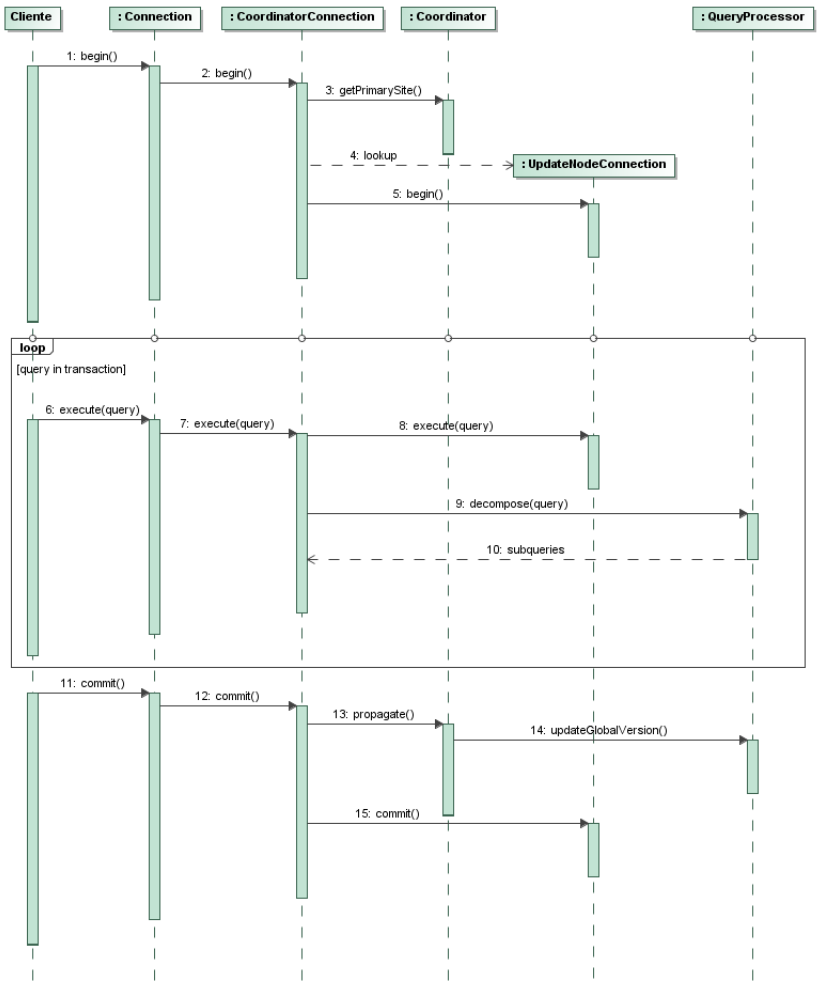


Figura 4.4 Diagrama de Sequência de uma Transação de Atualização no RepliXCoordinator

na chamada do método `execute` da instância da classe `CoordinatorConnection` (chamada 7). Por conseguinte, a consulta é passada por parâmetro ao método remoto `execute` do objeto do tipo `UpdateNodeConnection` (chamada 8). Em seguida, a consulta é decomposta ao ser chamado o método `decompose` da instância única da classe `QueryProcessor` (chamada 9), que retorna um conjunto de subconsultas (chamada 10).

Quando todas as consultas da transação são executadas, o cliente realiza uma chamada ao método `commit` do objeto do tipo `Connection` (chamada 11). Este método realiza uma chamada ao método `commit` do objeto remoto `CoordinatorConnection` (chamada 12). Por sua vez, este método faz uma chamada ao método `propagate` da instância única da classe `Coordinator`, passando como parâmetro uma transação de propagação (chamada 13). O método `propagate` adiciona a transação na fila de propagação global e atualiza o vetor de versão global dos fragmentos por chamar o método

`updateGlobalVersion` (chamada 14). Ao fim da execução do método propagate, o método `commit` é chamado da instância remota de `UpdateNodeConnection`, que consolida as alterações na base do sítio primário e propaga as alterações para os sítios secundários (chamada 15).

A Figura 4.5 ilustra o diagrama de sequência referente às chamadas realizadas entre a aplicação cliente e o RepliXPCoordinator para executar uma transação de leitura. Como os métodos `begin` e `commit` são opcionais para este tipo de transação, o diagrama apresenta apenas a execução de uma consulta de leitura.

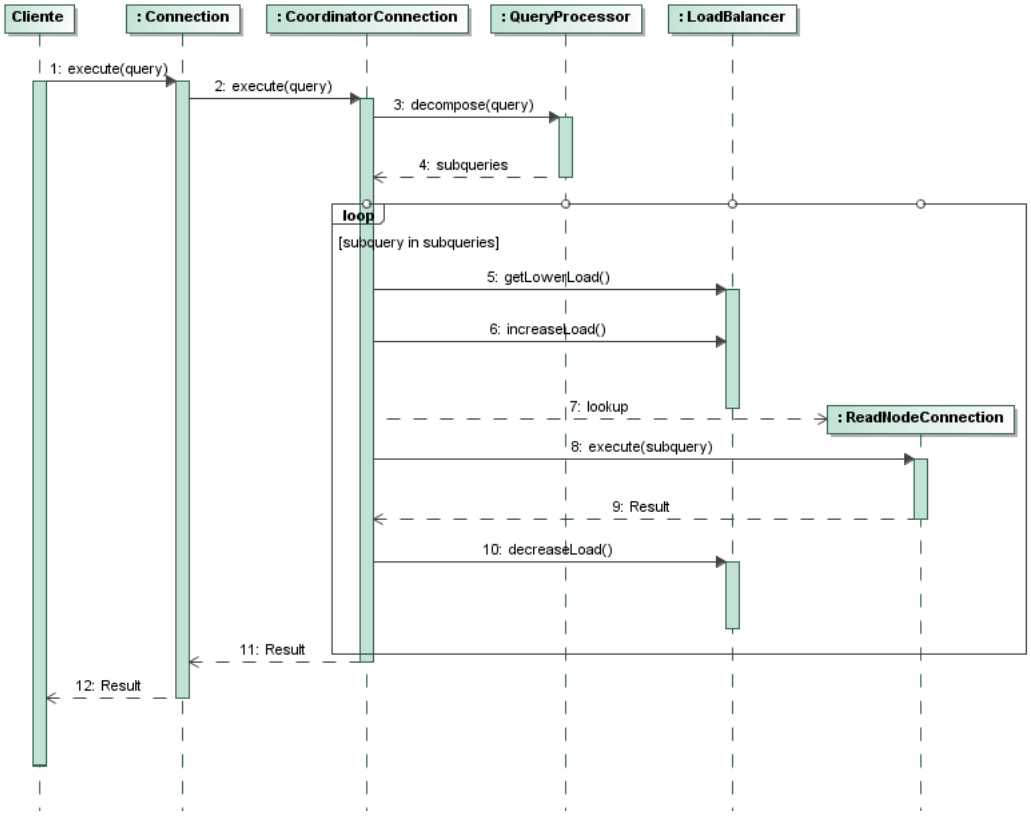


Figura 4.5 Diagrama de Sequência de uma Transação de Leitura no RepliXCoordinator

De início, a consulta é submetida ao componente RepliXPCoordinator pela chamada do método `execute` de um objeto da classe `Connection` (chamada 1). Este método faz uma chamada ao método `execute` da classe `CoordinatorConnection` (chamada 2). Por sua vez, este método faz uma chamada ao método `decompose` da instância única da classe `QueryProcessor` (chamada 3), que retorna um conjunto de subconsultas (chamada 4). Posteriormente, para cada uma das subconsultas retornadas, a seqüência de passos é executada.

Inicialmente, uma chamada ao método `getLowerLoad` da instância única da classe `LoadBalancer` é feita para descobrir qual sítio possui a menor carga (chamada 5). Em sequida, a chamada ao método `increaseLoad` incrementa a carga do sítio que executará a consulta (chamada 6). O método `lookup` cria uma nova instância remota da classe `ReadNodeConnection`. Na seqüência, o método `execute` desta classe é chamado (chamada 7) e a subconsulta é passada como parâmetro para ser executada no sítio de leitura (chamada 8). Ao finalizar a consulta, um objeto do tipo `Result` é retornado (chamada 9) e o método `decreaseLoad` é chamado para decrementar em uma unidade a carga do sítio que executou a consulta (chamada 10). Por fim, os resultados das subconsultas são unidos em um único objeto `Result`, que é retornado ao objeto da classe `Connection` (chamada 11), que, por sua vez, repassa este objeto para o cliente (chamada 12).

RepliXPNode

O RepliXPNode é o componente que gerencia as transações submetidas a um sítio de leitura ou escrita. A Figura 4.6 mostra o diagrama de classes desse componente. O componente RepliXPNode possui uma classe principal (omitida no modelo), que serve para iniciar sua execução pela chamada ao método `main`. Este método instancia um objeto do tipo `NodeDriverManager` e o publica como objeto remoto utilizando RMI.

Este objeto constitui o meio de comunicação inicial entre o coordenador e o sítio. O método `getConnection` é utilizado pela instância da classe `CoordinatorDriverManager` para obter uma instância remota de `NodeDriverManager`. Este método realiza a leitura de um arquivo XML de configuração contendo informações do sítio. Uma destas informações é o tipo do sítio. Caso o sítio seja de atualização, o método retorna uma instância remota da classe `UpdateNodeConnection`. Caso contrário, a instância remota retornada é do tipo `ReadNodeConnection`.

As classes `UpdateNodeConnection` e `ReadNodeConnection` implementam a interface `RemoteConnection`. No entanto, a implementação de cada método é específica para o tipo de transação manipulado. Além disso, a classe `UpdateNodeConnection` está associada a um objeto do tipo `UpdateNode` enquanto que a classe `ReadNodeConnection` referencia uma instância da classe `ReadNode`.

A classe `UpdateNode` é responsável por propagar as modificações de uma transação de atualização. O método `propagate` recebe como parâmetro uma transação, enviando-

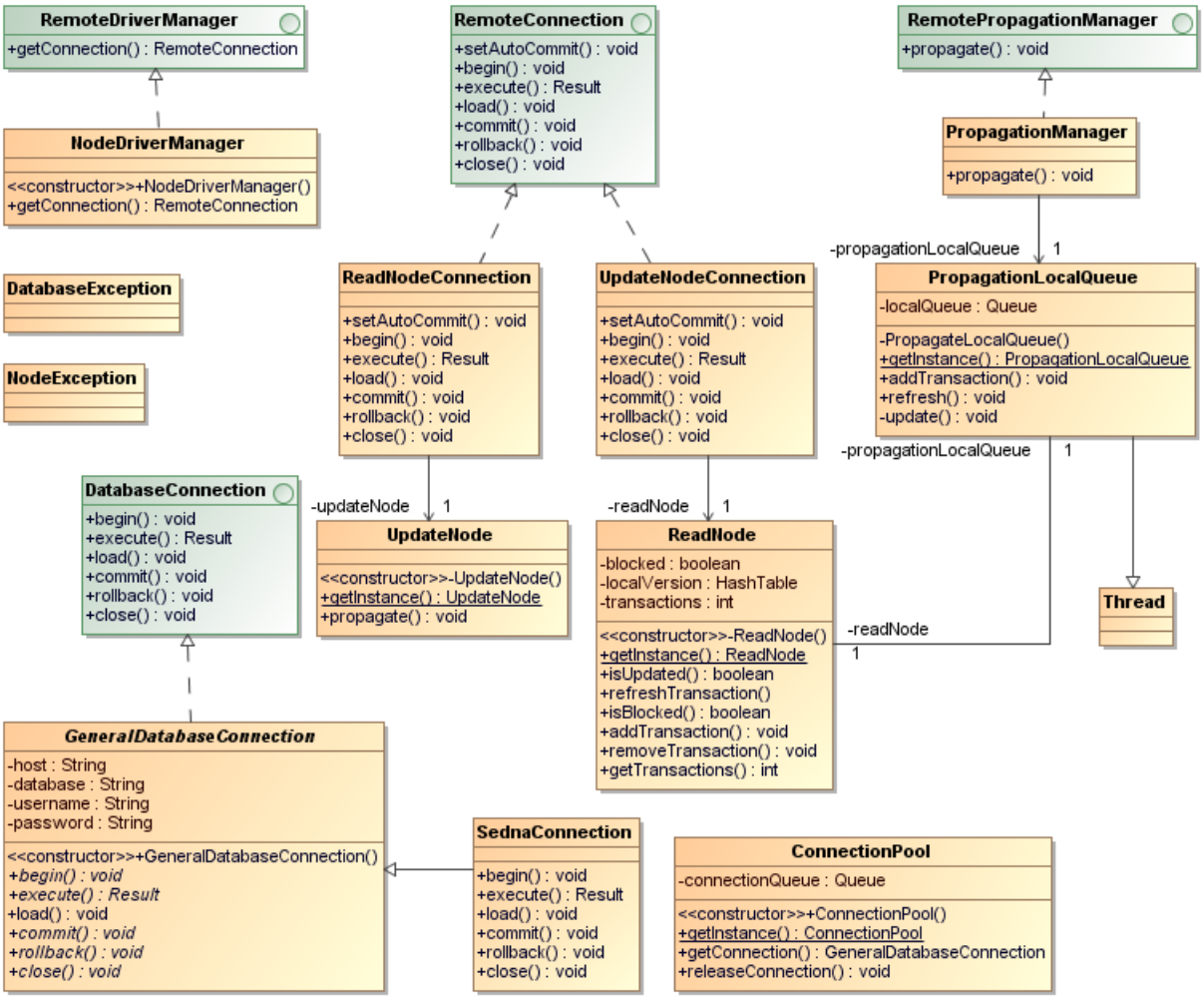


Figura 4.6 Diagrama de classes do RepliXPNode

a para todos os sítios secundários de atualização. São garantidas as propriedades de atomicidade e ordem na entrega da propagação.

A classe `ReadNode` controla todas as informações em um sítio de leitura. Ela possui o atributo `blocked` do tipo `boolean`, o qual indica se o sítio está bloqueado ou desbloqueado. O atributo `transactions` armazena o número de consultas de leitura que estão sendo executadas no sítio. O atributo `localCatalog` é um objeto do tipo `Catalog`, que serve para armazenar as informações dos fragmentos. O atributo `propagationLocalQueue`, instância da classe `PropagateLocalQueue`, referencia a fila de propagação local.

O método `isUpdate` recebe como parâmetro o identificador e a versão de um frag-

mento. Este método retorna o valor verdadeiro caso o fragmento esteja atualizado e falso, caso contrário. O método `isBlocked` retorna o valor da variável `blocked`. O método `refreshTransaction` recebe como parâmetro o identificador e a versão de um fragmento. Este método executa uma transação de *refresh* para atualizar o fragmento. Os métodos `addTransaction` e `removeTransaction` manipulam a valor do atributo `transactions`, incrementando e decrementando este valor a cada início e fim de uma consulta, respectivamente. O método `getTransactions` retorna o valor da variável `transactions`.

O método `main` também cria instâncias únicas para as classes `ConnectionPool`, `PropagationManager` e `PropagateLocalQueue`. A classe `ConnectionPool` serve para controlar um *pool* de conexões com a base de dados local. Ela reduz os custos com a abertura e o fechamento de uma conexão a cada requisição. O atributo `connectionQueue` representa uma fila de conexões com a base de dados. O método `getConnection` retorna uma conexão da fila `connectionQueue`. Caso todas as conexões estejam ocupadas, uma nova conexão é feita e retornada. O método `releaseConnection` recebe uma conexão que não está sendo utilizada, armazenando-a na fila para ser utilizada por outra requisição.

No componente RepliXPNode, a estratégia de conexão com a base de dados adota o padrão de projeto DAO [51]. A conexão é abstraída pela interface `DatabaseConnection`. Esta interface define todos os métodos da interface `RemoteConnection`. A classe abstrata `GeneralDatabaseConnection` implementa a interface `DatabaseConnection`. Ela contém os atributos `host`, `database`, `username` e `password`, que são os parâmetros de conexão com a base de dados. O padrão de projeto DAO proporciona à aplicação uma flexibilidade em relação ao tipo de banco de dados utilizado. Para que um SGBD seja utilizado, basta que seja criada uma subclasse concreta da classe `GeneralDatabaseConnection` que realize os métodos abstratos da classe herdada. Neste trabalho, foi implementada a classe `SednaConnection`, a qual implementa os métodos de conexão específicos para o SGBDXN Sedna [12].

A classe `PropagateLocalQueue` corresponde à fila de propagação local. O atributo `localQueue` consiste de uma fila que armazena as transações que chegam no sítio. O método `addTransaction` recebe como parâmetro uma transação e a adiciona na fila `localQueue`. O método `refresh` recebe como parâmetro o identificador de um fragmento e a versão requerida. Este método aplica uma transação de *refresh* para atualizar o valor dos dados do fragmento informado. A classe `PropagateLocalQueue` herda da classe `Thread` e possui um subprocesso que fica em execução contínua das transações de propagação.

Uma instância da classe `PropagationManager` é publicada remotamente via RMI na execução do método `main`. Ela implementa a interface `RemotePropagationManager`. Esta classe está relacionada com um objeto do tipo `PropagationManager`, o qual gerencia a fila de propagação local. O método `propagate` recebe as transações de propagação do coordenador e as repassa para o `PropagationManager`.

4.2 AVALIAÇÃO

À medida que os sistemas computacionais se tornam mais complexos, a análise de desempenho se torna uma atividade cada vez mais relevante e indispensável. No âmbito desses sistemas, uma ferramenta, para execução de testes-padrão ou *benchmark*, permite realizar um conjunto de testes projetados para comparar o desempenho de um sistema computacional em relação a outros, submetendo-os a uma carga de trabalho semelhante. Uma carga de trabalho corresponde ao conjunto de tarefas e recursos alocados para a execução de um sistema durante um intervalo de tempo. Por sua vez, a tarefa é a unidade de execução do sistema computacional. Por exemplo, no cenário de SGBDXN, uma tarefa pode corresponder a uma operação XQuery.

Durante o desenvolvimento e nos experimentos do RepliXP, optou-se por utilizar o SGBDXN Sedna [12] pelo fato de ser um sistema *open-source* e com as características de armazenamento e processamento de consultas necessárias à execução dos experimentos. Além disso, o Sedna possui uma API que fornece acesso simples para aplicações Java, o que facilitou a implementação do método de conexão do RepliXP com este SGBDXN.

O RepliXP visa a aumentar o desempenho no gerenciamento de bases de dados XML. Por utilizar a replicação parcial combinada com características de fragmentação de dados XML, o RepliXP proporciona melhor tempo de resposta. Assim, a avaliação no contexto deste trabalho busca analisar o tempo de resposta quando o RepliXP é utilizado.

Em razão das interfaces e linguagens de acesso aos SGBDXNs, SGBDs habilitados e às tecnologias para manipulação de documentos XML, torna-se complexo desenvolver experimentos apropriados para verificar o desempenho de sistemas como o RepliXP [52]. Nesse sentido, vários *benchmarks* para dados XML foram propostos, tais como os apresentados em [53] [34]. Lu et al.[54] ressaltam que nenhum estudo foi encontrado no uso de *benchmarks* que permitam ao usuário identificar o impacto causado pelo tipo de armazenamento no desempenho das operações XML. Argumentam que a observação da

forma como os dados são armazenados, ou seja, com ou sem a utilização de um esquema, influencia muito na avaliação do sistema.

Para a avaliação do RepliXP, estendeu-se a estratégia proposta pelo PartiX [16] para avaliar a fragmentação horizontal. Adicionaram-se consultas de atualização seguindo a linguagem de consultas do SGBDXN Sedna. Também se desenvolveu um simulador de clientes, denominado RepliXPSimulator, baseado em [15]. O RepliXPSimulator consiste de uma ferramenta para configurar o ambiente distribuído e realizar experimentos com base no tempo de resposta, criando um conjunto de clientes e submetendo cargas transacionais.

Ambiente de Avaliação

Seja $S = \{s_1, s_2, \ldots, s_n\}$ o conjunto de sítios do sistema. Cada sítio s_i possui um SGBDXN Sedna contendo os documentos XML escolhidos de acordo com a localização dos dados, adequados a cada experimento. Cada sítio possui uma instância do componente RepliXPNode acoplada. Considera-se, ainda, um conjunto de clientes $C = \{c_1, c_2, \ldots, c_m\}$, que contém os aplicativos que dão origem às transações. Cada cliente c_j utiliza o componente RepliXPDriver para realizar a conexão com o mecanismo. Além disso, um sítio adicional s_c contém uma instância do componente RepliXPCoordinator acoplado.

Para processar uma transação T, um cliente de c conecta-se ao sítio s_c e submete a requisição de T. O sítio s_c coordena a execução da transação, direcionando requisições para os sítios de S que devem processar as consultas de T. Os sítios recebem as requisições e executam localmente, retornando o resultado para o sítio s_c, o qual é repassado para o cliente c.

A concorrência de transações é simulada pela utilização de múltiplos clientes. O simulador de clientes RepliXPSimulator, visualizado na Figura 4.7, permite a configuração do ambiente de distribuição dos dados e gera transações de acordo com alguns parâmetros. Ele submete as transações para o sítio s_c e coleta os resultados ao final de cada execução. Os parâmetros usados para a geração das transações especificam o número de clientes, quantidade de transações por cliente, o tamanho da transação, a porcentagem de transações de atualização, o percentual de operação de atualização por transação deste tipo e o intervalo de atraso entre as transações.

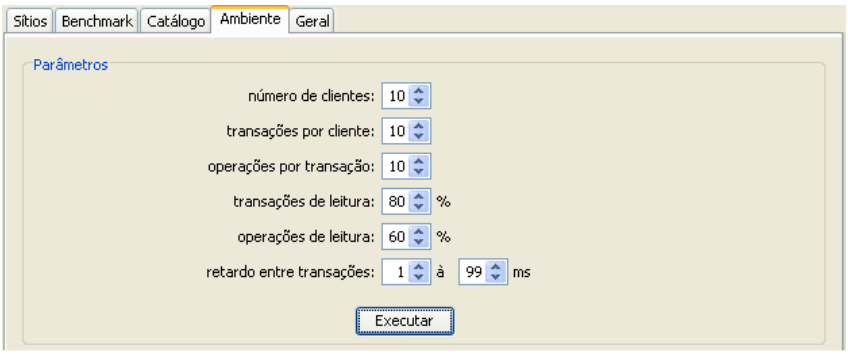


Figura 4.7 RepliXPSimulator

O ambiente utilizado para a avaliação foi um *cluster* de 8 PCs conectados através de um *Hub Ethernet*. Cada PC possui um processador de 3.0 GHz, 1 GB de RAM, sistema operacional Windows XP e interface de rede *full-duplex* de 100 Mbit/s. A base de dados consiste de uma coleção de documentos XML gerada pelo ToXGene [55]. Os testes foram realizados com 10 transações por cliente, onde cada transação era composta por 10 operações. Para cada cliente, 80% de suas transações eram de leitura. No caso das transações de atualização, 60% era composta por operações de leitura e 40% por operações de escrita.

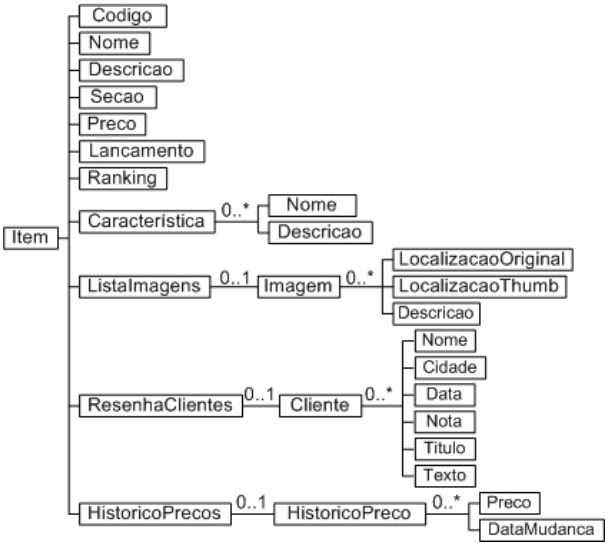


Figura 4.8 Esquema dos Documentos da Base de Dados

O RepliXP adota a estratégia de fragmentação horizontal proposta pelo PartiX [16]. A base `Itens` é formada por uma coleção de documentos que seguem o esquema ilustrado pela Figura 4.8. O foco da fragmentação está no elemento `Secao`, que representa em que seção um item pode ser encontrado. A base `Itens` foi criada seguindo uma

distribuição homogênea dos itens nas seções existentes.

Fragmento	Critério de Seleção	Sítios
$F01$	$/item/secao =' Livro'$	$R01$
		$R02$
		$R03$
$F02$	$/item/secao =' CD'$	$R02$
		$R03$
		$R04$
$F03$	$/item/secao =' DVD'$	$R03$
		$R04$
		$R05$
$F04$	$/item/secao =' Eletronico'$	$R04$
		$R05$
		$R06$

Tabela 4.1 Catálogo do Ambiente de Avaliação

A Tabela 4.1 apresenta o catálogo de dados global referente à avaliação. O ambiente possui 8 sítios, onde dois deles são o coordenador e o sítio primário de atualização. Os demais são sítios de leitura, cujos identificadores são $R01$, $R02$, $R03$, $R04$, $R05$ e $R06$. Na avaliação, foi utilizada a estratégia de espelhamento de dados. Cada fragmento está armazenado em três réplicas da base, as quais são alocadas em sítios distintos. O objetivo desta técnica é garantir uma maior disponibilidade e reduzir a quantidade de bloqueios entre as réplicas.

Resultados da Avaliação

Dentre os critérios de desempenho em sistemas de banco de dados, escolhemos o tempo de resposta da consulta como critério mais relevante para a comparação entre o desempenho das estratégias de replicação total e replicação parcial utilizando o RepliXP.

A Figura 4.9 representa o gráfico da variação do tempo de resposta com o aumento do número de clientes, com a base de dados de tamanho fixo igual a 10MB. De acordo com o gráfico, o RepliXP com 50 clientes resultou em um menor tempo de resposta ao adotar-se a replicação total. Isso ocorre porque, utilizando-se a replicação total com poucos clientes, tem-se uma pequena quantidade de bloqueios em decorrência de poucas operações concorrentes. Adicionalmente, a replicação parcial inclui mecanismos de controle que aumentam o tempo de resposta, relevante para uma pequena quantidade

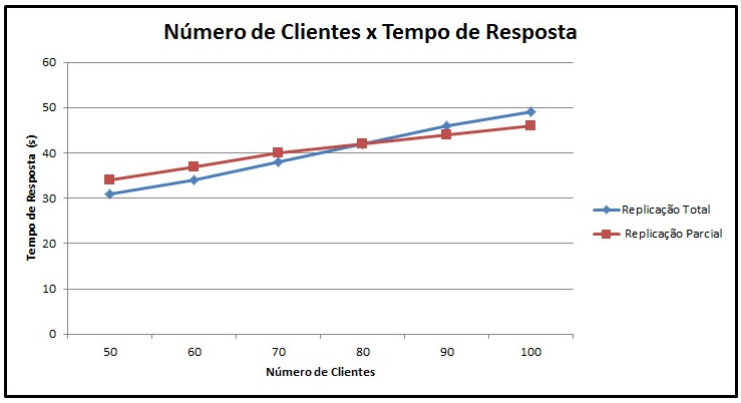


Figura 4.9 Gráfico Tempo de Resposta x Número de Clientes

de clientes.

À medida que aumentamos o número de clientes é aumentado, elevam-se a quantidade de operações concorrentes e, consequentemente, tem-se um número maior de bloqueios sob as bases de dados. Ao executar o RepliXP com 100 clientes, podemos verificar que o tempo de resposta com replicação parcial é menor que o tempo de resposta utilizando-se a estratégia de replicação total. Isso ocorre porque, na replicação parcial, ao executarmos uma operação de atualização, apenas os sítios que possuem os dados atualizados são bloqueados. No caso da replicação total, todos os sítios são bloqueados, uma vez que cada sítio possui uma cópia do recurso. Isso faz com que o tempo de resposta seja menor ao utilizarmos o RepliXP com replicação parcial.

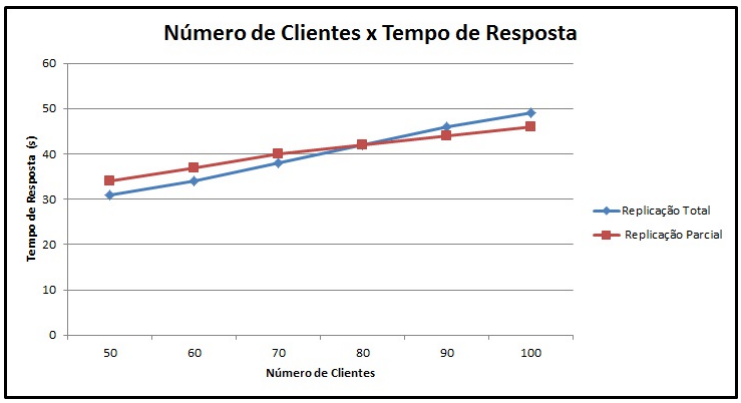


Figura 4.10 Gráfico Tempo de Resposta x Tamanho da Base de Dados

A Figura 4.10 representa o gráfico da variação do tempo de resposta com o aumento do tamanho da base de dados, com a quantidade fixa de 10 clientes. Conforme apresentado no gráfico, o RepliXP com uma base de dados de tamanho 10MB resultou

em um menor tempo de resposta ao adotar-se a replicação total. Isso ocorre porque, utilizando-se a replicação total com uma base dados de pequeno tamanho, pouco tempo é desprendido para a atualização de cada uma das bases. Adicionalmente, os mecanismos de controle da replicação parcial correspondem a um esforço adicional que aumentam o tempo de resposta, tornando-se relevante ao se utilizar bases pequenas.

A medida que aumentamos o tamanho da base distribuída, aumenta-se o tempo necessário para as atualizações. Ao executar o RepliXP com uma base de dados de tamanho 30MB, podemos verificar que o tempo de resposta com replicação parcial é menor que o tempo de resposta utilizando-se a estratégia de replicação total. Isso ocorre porque, na replicação parcial, ao executarmos uma operação de atualização, apenas os sítios que possuem os dados modificados sofre a atualização. No caso da replicação total, todos os sítios são atualizados, uma vez que cada sítio possui uma cópia do recurso. Isso faz com que o tempo de resposta seja menor ao utilizarmos o RepliXP com replicação parcial.

4.3 CONCLUSÃO

Este capítulo descreveu a implementação e avaliação do RepliXP. Foi justificada a escolha dos artefatos utilizados, bem como descritos os detalhes da implementação e os experimentos realizados. De acordo com o resultado da avaliação, concluímos que o RepliXP utilizando a estratégia de replicação parcial de dados XML apresentou uma diminuição no tempo de resposta das transações executadas. No capítulo a seguir, são apresentadas as conclusões deste trabalho.

CAPÍTULO 5

CONCLUSÃO

O presente trabalho apresentou o RepliXP como um mecanismo para prover a replicação parcial de base de dados XML, cujo objetivo é melhorar o desempenho desses sistemas. O RepliXP possui uma arquitetura modular e flexível, o que facilita sua utilização a qualquer SGBDXN ou com suporte a XML. Além do mais, sua arquitetura extensível permite que alterações nos componentes existentes ou desenvolvimento de novas funcionalidades sejam realizadas de forma transparente.

5.1 RESULTADOS ALCANÇADOS

Atualmente, existem alguns mecanismos para replicação em SGBDXNs. No entanto, até a conclusão deste trabalho, nenhum mecanismo propõe a replicação parcial de base de dados XML. Além do mais, as soluções existentes para replicação parcial de dados XML utilizam protocolos tradicionais, tais como cópia primária e réplicas ativas. As soluções de replicação baseadas em cópia primária oferecem um desempenho satisfatório, contudo, não favorecem a disponibilidade e apresentam problemas de escalabilidade. Por sua vez, as soluções baseadas em réplicas ativas melhoram a disponibilidade, permitindo a substituição de uma réplica com falha por uma operacional. Entretanto, por atualizarem todas as réplicas a cada atualização, geram muitos conflitos, o que se traduz em uma queda acentuada do desempenho, afetando a escalabilidade.

Neste trabalho foi apresentado o RepliXP, que define uma estratégia para a replicação parcial de dados utilizando protocolos de cópia primária e propagações de atualização de forma assíncrona para garantir a replicação de forma eficiente. Em particular, o RepliXP contempla a replicação parcial no âmbito de dados XML, possibilitando a aplicação de técnicas de fragmentação a um documento ou uma coleção de documentos XML.

Por fim, avaliou-se o RepliXP considerando características de desempenho no contexto da distribuição de dados. Esta avaliação consistiu em comparar os resultados

do RepliXP com replicação parcial e replicação total, ambos acoplados ao SGBD Sedna. Para isso, foi utilizada a estratégia de *benchmark* proposto por [43] com as devidas alterações para as consultas de atualização de dados. A carga de dados foi gerada pela ferramenta ToxGene [55] seguindo uma distribuição uniforme dos dados. Pela análise dos resultados, foi possível concluir que, para os testes executados, o RepliXP obteve um melhor desempenho em relação à solução de replicação total adotada na avaliação. De acordo com os resultados apresentados, o RepliXP utilizando a estratégia de replicação parcial de dados XML proporcionou uma melhoria no tempo de resposta das transações. Este comportamento foi observado tanto no aumento do número de clientes simultâneos como no aumento do tamanho da base de dados.

5.2 TRABALHOS FUTUROS

A escolha da técnica de fragmentação interfere diretamente no desempenho da replicação parcial. Devido a isto, um dos trabalhos futuros que se pretende desenvolver é a adaptação do RepliXP à outras estratégias de fragmentação não abordadas, como a fragmentação vertical e híbrida. Com isso, o mecanismo pode ser aplicado a outros cenários de fragmentação, podendo apresentar melhores resultados.

As consultas de atualização no modelo XML podem acarretar na alteração do esquema da base de dados. O RepliXP não possui suporte estas consultas, limita a solução à um conjunto reduzido de consultas que podem ser executadas. Portanto, outro direcionamento de trabalho futuro é estender o RepliXP de modo a permitir que consultas de atualização possam modificar a estrutura dos documentos da base. Para isso, é necessário que a solução modifique o catálogo global de dados no sítio coordenador, assim como especificar uma nova definição dos fragmentos existentes ou a criação de novos fragmentos.

A realocação dinâmica de dados em bases parcialmente replicados é uma das soluções adotadas para melhorar o desempenho destes sistemas em decorrência de alterações nos dados ao longo da execução do sistema. Um dos trabalhos futuros propostos ao RepliXP é a adoção de uma estratégia de realocação dinâmica dos dados. Esta solução permitiria que o mecanismo decidisse o momento em que a definição dos fragmentos fosse modificada para melhorar o seu desempenho.

A tolerância a falhas é uma das características de sistemas distribuídos que não é

contemplada pelo RepliXP. A falha é detectada, mas não existe um tratamento posterior. Para resolver essa limitação, um dos trabalhos futuros propostos para o RepliXP é o tratamento pós-falhas das requisições.

Na execução de uma consulta de leitura de dados, a escolha da réplica é feito por base na carga de cada sítio de leitura. A carga de um sítio tem como métrica o número de consultas que estão sendo executadas naquele sítio. No entanto, outros fatores influenciam na escolha do sítio que deve executar a consulta. Um dos trabalhos futuros é desenvolver um mecanismo de maior eficácia para o balanceamento de carga do RepliXP, levando em consideração fatores como o conteúdo das consultas e o estado do sítio.

Com relação à avaliação de desempenho, é proposto a avaliação do RepliXP em ambientes de WAN com o intuito de identificar a variação no desempenho adicionado em decorrência da latência da rede. Para tanto, são necessários identificar parâmetros e desenvolver cenários que contemplem um ambiente mais geral do que o utilizado nos experimentos aqui apresentados.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] W3C. Extensible markup language (xml) 1.0, 2006.

[2] W3C. Xml path language (xpath) version 1.0, 1999.

[3] W3C. Xquery 1.0: An xml query language, 2007.

[4] Oracle. Oracle database 11g xml db technical overview, 2007.

[5] Microsoft. Microsoft sql server 2005, 2005.

[6] Gang Gou and Rada Chirkova. Efficiently querying large xml data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[7] Harald Schoning. Tamino - a dbms designed for xml. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 149–154, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Thorsten Fiebig, Sven Helmer, Carl christian Kanne, Julia Mildenberger, Guido Moerkotte, Robert Schiele, and Till Westmann. Anatomy of a native xml base management system. *The VLDB Journal*, 11(4):292–314, 2002.

[9] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[10] Wolfgang Meier. exist: An open source native xml database. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 169–183, London, UK, 2003. Springer-Verlag.

[11] X-Hive. X-Hive Database. http://www.x-hive.com. Acessado em 01/08/2009.

[12] Andrey Fomichev, Maxim Grinev, and Sergey Kuznetsov. Sedna: A native xml dbms. In *32nd Conference on Current Trends in Theory and Practice of Computer Science*, volume 3831 of *SOFSEM 2006*, pages 272–281, Merin, Czech Republic, 2006. Springer-Verlag.

[13] M. Tamer Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1999.

[14] Fernanda Baião, Marta Mattoso, and Gerson Zaverucha. A distribution design methodology for object dbms. *Distributed and Parallel Databases*, 16(1):45–90, 2004.

[15] Flávio Rubens Carvalho Sousa. Replix: Um mecanismo para a replicação de dados xml. Mestrado em ciência da computação, Departamento de Computação, Universidade Federal do Ceará, 2007.

[16] Alexandre Silva Andrade. Partix: Projeto de fragmentação de dados xml. Mestrado, COOPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2006.

[17] Maria Pazin. Réplicas para alta disponibilidade em arquiteturas orientadas a componentes com suporte de comunicação de grupo. Dissertação de mestrado, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

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

[19] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 38–57, London, UK, 1996. Springer-Verlag.

[20] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM Press.

[21] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[22] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the The 20th*

International Conference on Distributed Computing Systems, ICDCS '00, page 464, Washington, DC, USA, 2000. IEEE Computer Society.

[23] d Can Türker, Hans-Jörg Schek, Yuri Breitbart, and Torsten Grabs aFuat Akal annd Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 565–576. VLDB Endowment, 2005.

[24] Esther Pacitti, Cédric Coulon, Patrick Valduriez, and M. Tamer Özsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 18(3):223–251, 2005.

[25] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, volume 0, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.

[26] J. E. Armendáriz, J. R. Juarez, J. R. G. de Mendivil, H. Decker, and F. D. Muñoz-Escoí. k-bound gsi: a flexible database replication protocol. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 556–560, New York, NY, USA, 2007. ACM Press.

[27] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[28] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[29] Otávio C. Décio. *Guia de Consulta Rápida XML*. Novatec, 2000.

[30] Elaine Castro. Xml-pm: Um método eficiente para identificação de padrões no processamento de consultas a dados xml. Mestrado em ciência da computação, Departamento de Computação, Universidade Federal do Ceará, Fortaleza, 2006.

[31] Daniela Florescu and Donald Kossmann. Storing and querying xml data using a rdbms. In *Bulletin of the Technical Committee on Data Engineering*, volume 22, pages 27–34, Washington, DC, USA, 1999. IEEE Computer Society.

[32] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. A general technique for querying xml documents using a relational database system. *SIGMOD Rec.*, 30(3):20–26, 2001.

[33] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of xml documents. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 137–150, London, UK, 2001. Springer-Verlag.

[34] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. Xbench benchmark and performance testing of xml dbmss. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, page 621, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Michiels Philippe. Xquery optimization. In *Proceedings of the VLDB 2003 PhD Workshop*, number 76 in VLDB '03, Berlin, Germany, 2003. Morgan Kaufmann.

[36] Cynthia P. Santiago and Javam C. Machado. i-fox: Um Índice eficiente e compacto para dados xml. In *XIX Simpósio Brasileiro de Bancos de Dados*, pages 191–203, Brasília, Distrito Federal, Brasil, 2004. UnB.

[37] XUpdate. Xml update language. http://xmldb-org.sf.net/xupdate/. Acessado em 01/08/2009.

[38] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity analysis in xml update languages. In *Proceedings of the 11th international conference on Database Theory*, volume 4353 of *ICDT'07*, pages 374–388. Springer-Verlag, 2007.

[39] Sven Hartmann, Sebastian Link, and Markus Kirchberg. A subgraph-based approach towards functional dependencies for XML. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, volume IX, pages 200–211. IIIS, 2003.

[40] Jan-Marco Bremer and Michael Gertz. On distributing xml repositories. In *International Workshop on the Web and Databases*, WebDB '03, pages 73–78, San Diego, California, 2003.

[41] Hui Ma and Klaus-Dieter Schewe. Fragmentation of xml documents. In *XVIII Simpósio Brasileiro de Bancos de Dados*, pages 200–214, Manaus, Amazonas, Brasil, 2003. UFAM.

[42] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis D. Viglas. Vectorizing and querying large xml repositories. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 261–272, Washington, DC, USA, 2005. IEEE Computer Society.

[43] Alexandre Andrade, Gabriela Ruberg, Fernanda Baião, Vanessa Braganholo, and Marta Mattoso. Efficiently processing xml queries over fragmented repositories with partix. In *Proceedings of the 2006 international conference on Current Trends in Database Technology*, volume 4254 of *EDBT'06*, pages 150–163, Munich, Germany, 2006. Springer-Verlag.

[44] Hiroto Kurita, Kenji Hatano, Jun Miyazaki, and Shunsuke Uemura. Efficient query processing for large xml data in distributed environments. In *Proceedings of the 21st International Conference on Advanced Networking and Applications*, AINA '07, pages 317–322, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Lawrence W. Dowdy and Derrell V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[46] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Transaction Database Systems*, 13(3):263–304, 1988.

[47] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.

[48] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

[49] Maydene Fisher, Jon Ellis, and Jonathan C. Bruce. *JDBC API Tutorial and Reference*. Pearson Education, 2003.

[50] Dom4j, 2009. http://dom4j.sourceforge.net. Acessado em 01/08/2009.

[51] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[52] Leonardo Oliveira Moreira. Dtx: Um mecanismo de controle de concorrência distriuído para dados xml. Master's thesis, Universidade Federal do Ceará, 2008.

[53] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 974–985, Hong Kong, China, 2002. Morgan Kaufmann.

[54] Hongjun Lu, Jeffrey Xu Yu, Guoren Wang, Shihui Zheng, Haifeng Jiang, Ge Yu, and Aoying Zhou. What makes the differences: benchmarking xml database implementations. *ACM Transactions on Internet Technology*, 5(1):154–194, 2005.

[55] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: A template-based data generator for xml. In *Proceedings of the 2002 ACM international conference on Management of data*, SIGMOD '02, page 616, New York, NY, USA, 2002. ACM.

[56] J. E. Armendáriz, J. R. Juárez, J. R. Garitagoitia, J. R. González de Mendívil, and F. D. Muñoz-Escoí. Implementing database replication protocols based on o2pl in a middleware architecture. In *Proceedings of the 24th IASTED International Conference on Database and Applications*, DBA'06, pages 176–181, Anaheim, CA, USA, 2006. ACTA Press.

[57] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys (CSUR)*, 23(3):269–317, 1991.

[58] Stijn Dekeyser and Jan Hidders. Conflict scheduling of transactions on xml documents. In *Proceedings of the 15th Australasian Database Conference*, ADC '04, pages 93–101, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[59] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. Silkroute: A framework for publishing relational data in xml. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[60] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. Xmltm: efficient transaction management for xml documents. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, CIKM '02, pages 142–152, New York, NY, USA, 2002. ACM Press.

[61] Michael Peter Haustein and Theo Harder. A lock manager for collaborative processing of natively stored xml documents. In *XIX Simpósio Brasileiro de Bancos de Dados*, pages 230–244, Brasília, Distrito Federal, Brasil, 2004. UnB.

[62] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Evaluating lock-based protocols for cooperation on xml documents. *SIGMOD Record*, 33(1):58–63, 2004.

[63] Kuen-Fang Jack Jea, Shih-Ying Chen, and Sheng-Hsien Wang. Concurrency control in xml document databases: Xpath locking protocol. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, ICPADS '02, page 551, Washington, DC, USA, 2002. IEEE Computer Society.

[64] Kamalakar Karlapalem and Qing Li. A framework for class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8(3):333–366, 2000.

[65] Meike Klettke and Holger Meyer. Xml and object-relational database systems - enhancing structural mappings based on statistics. In *Selected papers from the 3th International Workshop on The World Wide Web and Databases*, WebDB '00, pages 151–170, London, UK, 2000. Springer-Verlag.

[66] Cristiano Cachapuz Lima. Orpis: Um modelo de consistência de conteúdo replicado em servidores web distribuídos. Mestrado, Universidade Federal do Rio Grande do Sul, Porto Alegre, maio 2003.

[67] Xuemin Lin, Maria E. Orlowska, and Yanchun Zhang. On data allocation with the minimum overall communication costs in distributed database design. In *Proceedings of the 5th International Conference on Computing and Information*, ICCI '93, pages 539–544, Washington, DC, USA, 1993. IEEE Computer Society.

[68] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *ACM SIGMOD Record*, 26(3):54–66, 1997.

[69] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Survey*, 22(4):299–319, 1990.

[70] Humberto Vieira, Gabriela Ruberg, and Marta Mattoso. Xverter: querying xml data with or-dbms. In *Proceedings of the 5th ACM International Workshop on Web Information and Data Management*, WIDM '03, pages 37–44, New York, NY, USA, 2003. ACM Press.