Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

REPLIX: UM MECANISMO PARA A REPLICAÇÃO DE DADOS XML

Flávio Rubens de Carvalho Sousa

DISSERTAÇÃO DE MESTRADO

Fortaleza
Março - 2007

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação

Flávio Rubens de Carvalho Sousa

REPLIX: UM MECANISMO PARA A REPLICAÇÃO DE DADOS XML

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Javam de Castro Machado, DSc

Fortaleza
Março - 2007

REPLIX: UM MECANISMO PARA A REPLICAÇÃO DE DADOS XML

Flávio Rubens de Carvalho Sousa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Javam Machado, DSc
Universidade Federal do Ceará

Profa. Rossana Andrade, PhD
Universidade Federal do Ceará

Prof. Altigran Silva, DSc
Universidade Federal do Amazonas

Aprovada em 09 de Março de 2007

Aos Meus Pais.

AGRADECIMENTOS

Ao Prof. Javam Machado, meu orientador. Por sempre ter acreditado no meu potencial e compartilhar seu conhecimento e experiência, fundamentais ao bom desenvolvimento deste trabalho. Obrigado pela confiança e paciência nos momentos mais delicados.

Aos Professores Maxim Grinev, Peter Pleshachkov, Andrey Fomichev e Maria Rekouts do ISPRAS (Institute for System Programming - Russian Academy of Sciences) pela colaboração e atenção dedicada a este trabalho.

A Heraldo Carneiro pela ajuda constante no desenvolvimento e revisão deste trabalho.

A John Schultz da Spread Concepts, a Profa. Bettina Kemme da McGill University, a Jérôme Siméon e ao Prof. Aldri Santos pelos conhecimentos e visões que me transmitiram.

Ao Prof. Altigran Silva e a Profa. Rossana Andrade pelas valiosas sugestões no aprimoramento desta dissertação.

Aos meus pais, Francisco de Sousa Izidório e Maria Edileusa de C. O. Sousa, que jamais pouparam esforços na abençoada tarefa de me fazer feliz.

A minha namorada, Kaluce Gonçalves, a cujo amor e dedicação, devo alto percentual de minhas realizações.

A Bringel Filho, Lincoln Rocha e Antoine Bouhours, por compartilhar comigo suas alegrias.

A todos aqueles que me apoiaram durante esse período. Obrigado a todos que, de alguma forma ou de outra, deixaram algo em mim.

Ao CNPq pelo apoio financeiro sem o qual esse trabalho não teria sido possível.

A Deus, por estar sempre ao meu lado, dando-me coragem para enfrentar todos os obstáculos da vida.

If I have seen further it is by
standing on the shoulders of Giants
—ISAAC NEWTON

RESUMO

XML tem se tornado um padrão amplamente utilizado na representação e troca de dados em aplicações. Devido a essa crescente utilização do XML, torna-se necessária a existência de sistemas eficientes de armazenamento e recuperação de dados XML. Estão sendo desenvolvidos para este fim Bancos de Dados XML Nativos (BDXNs). Estes bancos implementam muitas das características presentes em Bancos de Dados tradicionais, tais como armazenamento, indexação, processamento de consultas, transações e replicação.

Tratando-se especificamente de replicação, a maioria das soluções existentes resolve essa questão apenas utilizando técnicas tradicionais. Todavia, a flexibilidade dos dados XML impõe novos desafios, de modo que novas técnicas de replicação devem ser desenvolvidas. Para melhorar o desempenho e a disponibilidade dos BDXNs, esta dissertação propõe o RepliX, um mecanismo para replicação de dados XML que considera as principais características desses dados. Dessa forma, é possível melhorar o tempo de resposta no processamento de consultas e tornar esses sistemas mais tolerantes a falhas.

Dentre vários tipos de protocolos de replicação, a utilização da abstração de comunicação em grupos como estratégia de comunicação e detecção de falhas mostra-se uma solução eficaz, visto que essa abstração possui técnicas eficientes para troca de mensagens e provê garantias de confiabilidade. Essa estratégia é utilizada no RepliX, que organiza os sites em dois grupos: de atualização e de leitura, permitindo assim balanceamento de carga entre os sites, além de tornar o sistema menos sensível a falhas, já que não há um ponto de falha único em cada grupo.

Para validar o RepliX, uma nova camada de replicação foi implementada em um BDXN, a fim de introduzir as características e os comportamentos descritos no mecanismo proposto. Experimentos foram feitos usando essa camada e os resultados obtidos atestam a sua eficácia considerando diferentes aspectos de um banco de dados replicado, melhorando o desempenho desses banco de dados consideravelmente bem como sua disponibilidade.

Palavras-chave: BDs XML Nativos, Comunicação em Grupo, Replicação

ABSTRACT

XML has become a widely used standard for data representation and exchange in applications. The growing usage of XML creates a need for efficient storage and recovery systems for XML data. Native XML DBs (NXDBs) are being developed to target this demand. NXDBs implement many characteristics that are common to traditional DBs, such as storage, indexing, query processing, transactions and replication.

Most existing solutions solve the replication issue through traditional techniques. However, the flexibility of XML data imposes new challenges, so new replication techniques ought to be developed. To improve the performance and availability of NXDBs, this thesis proposes RepliX, a mechanism for XML data replication that takes into account the main characteristics of this data type, making it possible to reduce the response time in query processing and improving the fault-tolerance property of such systems.

Although there are several replication protocols, using the group communication abstraction for communication and fault detection has proven to be a good solution, since this abstraction provides efficient message exchanging techniques and confiability guarantees. RepliX uses this strategy, organizing the sites into an update group and a read-only group in such a way that allows for the use of load balancing among the sites, and makes the system less susceptible to faults, since there is no single point of failure in each group.

In order to evaluate RepliX, a new replication layer was implemented on top of an existing NXDB to introduce the characteristics of the proposed mechanism. Several experiments using this layer were conducted, and their results confirm the mechanism's efficiency considering the different aspects of a replicated database, improving its performance considerably, as well as its availability.

Keywords: Native XML Databases, Group Communication, Replication

SUMÁRIO

LISTA DE ABREVIATURAS

API - Application Programming Interface

ACID - Atomicidade, Consistência, Isolamento e Durabilidade

BDXN - Banco de Dados XML Nativo

CG - Comunicação em Grupo

DOM - Document Object Model

DTD - Document Type Definition

FIFO - First In First Out

HTTP - Hyper Text Transfer Protocol

JDBC - Java Database Connectivity

LAN - Local Area Network

OSI - Open System Interconnection

RMI - Remote Method Invocation

RPC - Remote Procedure Call

SOAP - Simple Object Access Protocol

2PC - Two-Phase Commit

2PL - Two-Phase Locking

XML - eXtensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Esta dissertação apresenta o RepliX, um mecanismo para replicação em sistemas de banco de dados XML nativos. Esse mecanismo melhora a disponibilidade destes sistemas, quando redireciona as requisições dos clientes para réplicas operacionais assim, como o desempenho, através de acesso concorrente às réplicas.

Neste capítulo serão apresentadas a justificativa e a motivação para o desenvolvimento deste trabalho, assim como os objetivos e as contribuições que se pretende alcançar. Ao final do capítulo, será descrito como está organizada o restante desta dissertação.

1.1 MOTIVAÇÃO E CARACTERIZAÇÃO DO PROBLEMA

Nos últimos anos, o XML (*Extensible Markup Language*) [1] tem se tornado um padrão amplamente utilizado na representação e troca de dados em aplicações, tais como comércio eletrônico [2] e cadastro bibliográfico [3]. Devido a essa crescente utilização do XML, torna-se necessária a existência de sistemas eficientes de armazenamento e recuperação de documentos XML.

Para isto, estão sendo propostos e implementados BDs XML Nativos (BDXNs) [4]. BDXNs são sistemas que armazenam documentos XML segundo uma estrutura lógica de grafo, onde os nós representam elementos e atributos e as arestas definem os relacionamentos elemento/sub-elemento ou elemento/atributo [5]. Esses bancos implementam muitas das características presentes em sistemas tradicionais, tais como armazenamento, indexação, processamento de consultas, transações e replicação.

Técnicas de replicação de dados têm sido usadas para melhorar a disponibilidade, o desempenho e a escalabilidade em BDs tradicionais, tais como os relacionais, e aqueles orientados a objetos [6][7]. Todavia, a flexibilidade do modelo XML dificulta a aplicação de técnicas existentes a BDXNs.

Propostas atuais para replicação de dados XML tentam adaptar os conceitos existentes ao modelo XML [8][9][10]. Contudo, poucos BDXNs fornecem mecanismos de replicação e não existem estudos que avaliem explicitamente aspectos de desempenho, escalabilidade e disponibilidade desses mecanismos.

Dentre vários tipos de protocolos de replicação, a abstração de comunicação em grupos (CG) é uma tecnologia eficiente para implementar estes protocolos, pois provê garantias de confiabilidade que simplificam a aplicação de técnicas de tolerância a falhas [11]. Recentemente, primitivas de comunicação em grupo têm sido aplicadas com eficiência nesses protocolos, tanto em abordagens síncronas [12][13] como assíncronas [14][15]. Dessa forma, acredita-se que CG é uma solução viável para implementar protocolos de replicação em BDXNs.

1.2 OBJETIVO E CONTRIBUIÇÃO

Este trabalho propõe o RepliX, um mecanismo para solucionar o problema de replicação de dados em BDXNs que utiliza formas de propagação síncronas e assíncronas e contempla as características dos dados XML. O RepliX faz uso de primitivas de CG para garantir a consistência entre as réplicas. O objetivo principal consiste em melhorar o desempenho, a escalabilidade e a disponibilidade dos BDXNs. Além disso, pretende-se que o RepliX possa ser integrado aos principais BDXNs.

A principal contribuição desta dissertação é o mecanismo para replicação de dados XML, baseado em técnicas síncronas e assíncronas e CG. Técnicas síncronas são utilizadas em transações de atualização e assíncronas em transações de leitura. Essas técnicas, associadas às primitivas de CG, permitem contemplar as principais características de um ambiente replicado. Outra contribuição é uma extensão do *benchmark* XMark.

1.3 ESTRUTURA DA DISSERTAÇÃO

Os próximos capítulos desta dissertação estão estruturados da seguinte forma:

- Capítulo 2: introduz os Bancos de Dados XML Nativos, destacando suas principais características, e apresenta alguns destes bancos.

- Capítulo 3: apresenta os principais protocolos de replicação e os conceitos de comunicação em grupos, além de apresentar os trabalhos relacionados.
- Capítulo 4: descreve detalhes sobre o RepliX, a especificação, a arquitetura e os algoritmos desenvolvidos.
- Capítulo 5: apresenta a implementação e a avaliação do RepliX.
- Capítulo 6: apresenta as conclusões e os possíveis trabalhos futuros.

CAPÍTULO 2

BANCO DE DADOS XML NATIVOS

Este capítulo discute as principais características dos Bancos de Dados XML Nativos, destacando a sua arquitetura e organização típica. Nele também são abordadas características dos dados XML que influenciam na definição do RepliX, tais como o processamento de consultas e o controle de concorrência. Por fim, apresentamos os principais BDXNs encontrados na literatura.

2.1 CARACTERÍSTICAS

Segundo [4], BDXNs 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 árvore, caso este não possua ciclos direcionados, onde os nós representam elementos e atributos e onde uma aresta de um nó v para um nó v' representa uma relação pai/filho (elemento/sub-elemento ou elemento/atributo) entre v e v' [5].

2.1.1 Arquitetura

Assim como os BDs tradicionais, os BDXNs têm sido projetados de acordo com o modelo de três camadas, a seguir: externa, conceitual e física [16], como ilustra a Figura 2.1.

A camada externa exibe diferentes graus de funcionalidades de um sistema para os grupos de usuários, permitindo encapsular aspectos específicos do BD que, por razões de segurança, não devem ser expostos ou visíveis para usuários sem privilégios de acesso ou não autorizados. Já a camada conceitual mostra a estrutura sobre a qual os documentos são gerenciados. Nessa camada, a unidade fundamental é o documento XML, que

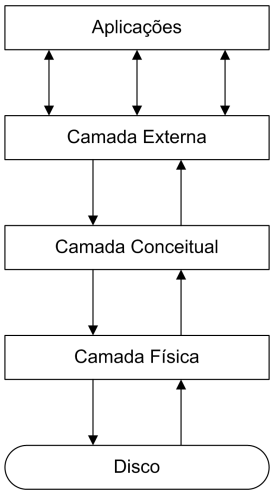

Figura 2.1 Arquitetura genérica de um BDXN

constitui a forma de armazenamento lógico. Para facilitar a organização, os dados XML são agrupados, formando coleções de dados.

Finalmente, a camada física transforma a representação lógica dos dados em estruturas físicas, que, por sua vez, são armazenadas no disco. Durante essa transformação, o sistema não deve perder nenhum tipo de informações do documento de entrada, tais como *namespaces*, *IDs* e *IDREFs*, úteis na reconstrução dos documentos.

Organização

Em virtude da flexibilidade dos dados XML, os BDXNs geralmente utilizam, na camada conceitual, uma estratégia que permita ao banco de dados gerenciar esses dados. Uma organização bastante utilizada no gerenciamento XML é o conceito de coleções de documentos XML.

Uma coleção é um agrupamento de documentos de acordo com sua relevância semântica. Em geral, cada documento armazenado no banco é associado a uma coleção. As coleções têm papel similar às tabelas em bancos relacionais ou diretórios em um sistema de arquivos, também sendo usados nos bancos orientados a objetos. A quantidade dessas coleções em um BDXN pode variar bastante, dependendo da granularidade com a qual as aplicações utilizam os documentos XML.

De acordo com o tipo de acesso dessas aplicações, uma coleção pode ser dividida em subcoleções. Por exemplo, uma coleção de compras pode ser particionada de acordo

com a localização dos clientes. Essa coleção, por sua vez, pode ser dividida por preço ou mercadoria, criando assim, subcoleções. A Figura 2.2 mostra um exemplo de coleção de documentos XML. A coleção P é uma coleção simples. Já a coleção Q é uma coleção composta, pois contém diversas coleções simples. A união de todas as coleções é chamada de *root*, representada na figura abaixo por R.

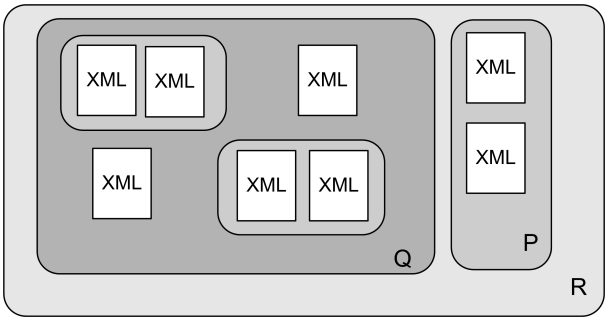

Figura 2.2 Coleção de documentos XML

2.1.2 Processamento de Consultas

O desenvolvimento de técnicas de processamento de consultas sobre dados XML é uma tarefa desafiadora [17]. Isso se deve principalmente às seguintes características:

- Modelo de dados: documentos XML são representados por um modelo de dados baseado em grafo, o que adiciona maior complexidade à sua estrutura.
- Heterogeneidade: um documento XML pode ter um mesmo sub-elemento omitido ou repetido várias vezes.

Sob o ponto de vista de BDs, o modelo de dados XML é um modelo semi-estruturado, onde a estrutura dos dados de um documento é variável e nem sempre é conhecida previamente, diferentemente dos modelos estruturados, cuja estrutura é explicitamente declarada através de um esquema.

A execução de consultas em um BDXN é similar a dos bancos relacionais [18]. Uma diferença é a falta de tipificação. Segundo a especificação da linguagem XQuery, o processamento de consultas em BDXNs é feito em duas fases:

- análise da consulta: prepara o plano de execução da consulta através da verificação sintática de tipos e otimização;
- execução da consulta: executa o plano e faz a verificação dinâmica de tipos.

Nessas fases é importante o conhecimento da informação dos esquemas para que possa ser feita a verificação de tipos. Por isso, antes de implementar essas fases em um BDXN, deve-se conhecer a organização física e lógica dos dados, tais como definição dos esquemas e dos índices dos documentos armazenados no banco [18].

Uma questão importante diz respeito à flexibilidade do sistema para permitir a consulta sobre documentos cujos esquemas e índices não são conhecidos previamente. O uso de um esquema auxilia no processamento de consultas, eliminando expressões de caminho incompatíveis, como expressões que sempre são vazias, na remoção de condições redundantes [19] e na detecção certos tipos de erros em tempo de compilação.

Diferentemente dos modelos de dados tradicionais, o modelo XML ainda não dispõe de uma álgebra padrão. Álgebras são usadas para dar semântica a linguagens de consulta e auxiliar sua otimização. Muitos trabalhos têm sido propostos na definição de uma álgebra. Em [20] é apresentado um álgebra que captura a semântica de muitas linguagens XML e possui várias operações tais como junção, projeção, seleção, agregação, funções, entre outras. Jagadish et al. [21] propõem uma álgebra que utiliza as características dos dados XML e possui regras análogas à álgebra relacional, mas bastante limitada.

Devido à dificuldade da implementação de uma álgebra padrão e à complexidade das linguagens de consultas, o W3C tomou como base a álgebra proposta em [20] e desenvolveu uma semântica formal [22]. Essa semântica possibilita a identificação de ambigüidades na linguagem, assim como auxilia na verificação formal [23].

Linguagens como a XPath e a XQuery contêm uma semântica formal em seus núcleos para a qual as consultas submetidas em alto nível são convertidas [19]. Essa conversão proporciona uma série de benefícios que são inerentes a muitas linguagens funcionais, tais como verificação forte de tipos e facilidades na utilização de técnicas de otimização, como, por exemplo, prevenção de nós duplicados. Michiels [19] discute o *status* dessa semântica formal como uma álgebra e apresenta um comparativo entre elas.

Essas linguagens não possuem suporte a operações de atualização de documentos, tais como inserção ou remoção [24]. Alguns trabalhos propõem soluções para esse

problema. Em [25] é apresentada uma linguagem para atualização de documentos XML denominada XUpdate. Já [26] discute alterações no núcleo da linguagem XQuery através da adição de extensões.

2.1.3 Controle de Concorrência

A estrutura em grafo dos dados XML dificulta o desenvolvimento de protocolos de controle de concorrência a esses dados. Em [27] é apresentado um protocolo baseado em bloqueios hierárquicos em árvores. Apesar de aplicar os bloqueios sobre as árvores, estrutura semelhante a dos documentos XML, esse protocolo possui um baixo nível de concorrência, pois os bloqueios ocorrem de forma *top-down*, ou seja, os nós desde o ponto inicial da consulta até o final do documento também são bloqueados. Assim sendo, consultas XPath não podem ser aplicadas eficientemente, pois podem ser executadas sobre partes distintas de um documento, que, por sua vez, podem estar bloqueadas desnecessariamente.

Protocolos baseados no modelo DOM também foram propostos [28] [29]. Esses protocolos usam diferentes tipos de bloqueio para agrupar nós de níveis distintos, ocasionando um elevado número de conflitos. Para reduzir esses conflitos, [29] utiliza a estrutura de DTD, melhorando o acesso aos dados. Esses protocolos são utilizados por vários sistemas e apresentam resultados satisfatórios com operações sobre o modelo DOM. Contudo, não existem estudos que comprovem a eficiência desses protocolos quando linguagens de consultas XML, como a XQuery são submetidas.

Os protocolos desenvolvidos em [29][30] contemplam as características das linguagens XML, utilizando bloqueios de caminho para aumentar a concorrência e melhorar o desempenho das consultas. Entretanto, esses protocolos apresentam algumas desvantagens, tais como a utilização de um subconjunto muito limitado da linguagem XPath e o uso de métodos dispendiosos para determinar conflitos entre as consultas mais complexas, que inviabilizam sua aplicação em sistemas práticos.

Em [31] é apresentado o protocolo DGLOCK, que contempla as principais características de concorrência a dados XML. Nesse trabalho, são apresentados resultados que comprovam sua eficiência. Uma desvantagem do DGLOCK é não garantir o critério da seriabilidade [32], característica fundamental a protocolos de concorrência e, por isso, pode gerar inconsistências. Pleshachkov et al. [33] apresentam uma melhoria do DGLOCK que garante a seriabilidade, mas não apresentam experimentos que comprovem sua eficiência.

2.1.4 Fragmentação

A fragmentação de dados consiste em determinar alternativas para divisão dos dados em unidades menores e verificar critérios como *completude*, *reconstrução* e *disjunção* [34]. No modelo relacional, tem-se basicamente duas formas: a fragmentação horizontal, que divide uma tabela em função das suas linhas, e a vertical, que fragmenta as colunas. A maioria dos trabalhos no contexto XML tenta adaptar as técnicas dos sistemas tradicionais para solucionar esse problema.

Em [35] são apresentadas técnicas de fragmentação de objetos adaptadas ao modelo XML. São definidas as fragmentações horizontal, vertical e *split* (partição de um documento em outros, cada um destes com um identificador). A ausência dos critérios descritos anteriormente e a utilização de DTDs no processo de fragmentação dificultam a aplicação dessas técnicas a dados XML.

Buneman et al. [36] adaptaram a técnica de vetorização, que consiste na divisão dos dados em colunas, a documentos XML. Esses documentos são decompostos em um conjunto de vetores, cada um contendo os dados desde a raiz até as folhas. Para permitir a execução de consultas distribuídas, um subconjunto de consultas XQuery foi decomposta. Resultados experimentais demostraram melhorias no processamento das consultas, mas o subconjunto da XQuery utilizado e a verificação apenas do critério de *reconstrução* nesse trabalho não viabilizam sua utilização em bases XML.

Bremer e Gertz [37] apresentam técnicas relacionais adaptadas ao modelo XML. Esse trabalho utiliza um esquema denominado *Repository Guide* para auxiliar na divisão dos dados e na verificação dos critérios de correção. São apresentados as fragmentações vertical, que consiste no particionamento do esquema *Repository Guide* e horizontal, que realiza a divisão do documento aplicando condições aos atributos do documento. Apesar de contemplar os critérios de fragmentação, as técnicas desenvolvidas não são bem definidas, limitam-se a linguagem XPath e não são apresentados resultados que validem a proposta.

Em [38] é apresentado o PartiX, 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 documentos XML e sim em coleções XML. 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 sub-consultas foram executadas em paralelo. Apesar dos resultados satisfatórios apresentados, o PartiX considera coleções XML, não podendo ser aplicado a documentos XML separadamente e a decomposição das consultas XQuery ainda é bastante limitada.

2.2 BDS XML NATIVOS

Nesta seção serão apresentados alguns dos Bancos de Dados XML Nativos em evidência, destacando suas principais características.

Timber

O Timber é um BDXN que vem sendo desenvolvido pela Universidade de Michigan [4]. Seu sistema é construído sobre o gerenciador *Shore*, que é responsável pelo armazenamento, gerenciamento de memória e controle de concorrência. O Timber armazena os dados XML, os índices e os metadados através dos módulos *Data Manager*, *Index Manager* e *Metadata Manager* respectivamente, mostrados na Figura 2.3.

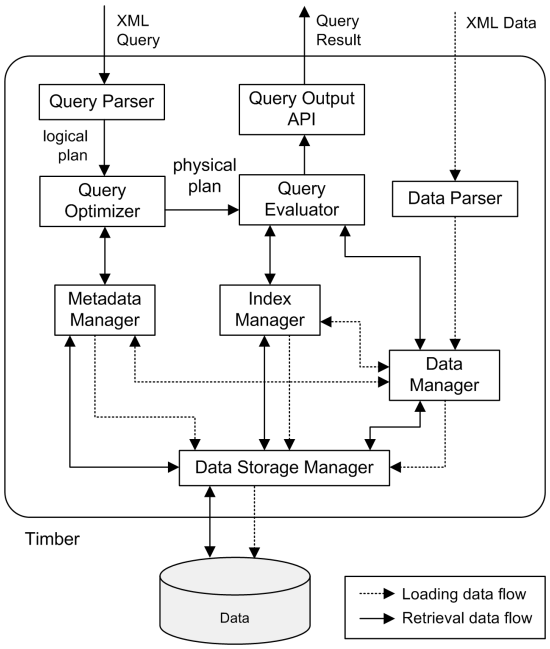

Figura 2.3 Arquitetura do Timber [4]

O *Data Storage Manager* armazena os documentos associando para um cada de seus nós uma tupla (*start*, *end*, *level*), que pode ser usada como identificador do nó, onde *start* e *end* definem o intervalo entre os rótulos de um dado nó, tal que cada nó descendente tem um intervalo que está estritamente incluído no intervalo de seus antecedentes, determinando o relacionamento antecedente-descendente. *Level* representa o nível de aninhamento de um dado nó no documento, determinando, juntamente com *start* e *end*, o relacionamento pai-filho.

O Timber utiliza a álgebra TAX [21] no seu processamento de consulta e uma extensão da linguagem XQuery para prover operações de atualização sobre os documentos. As consultas submetidas ao Timber são reconhecidas e transformadas em uma árvore de operadores algébricos pelo *Query Parser*. O *Query Optimizer* reconhece essa árvore e realiza um mapeamento dos operadores lógicos para os físicos. O plano de consulta resultante é avaliado pelo *Query Evaluator*. Cada operador é processado em *pipeline*, um de cada vez, por meio de um conjunto de chamadas ao *Data Manager* e *Index Manager* que, por sua vez, requisita o *Data Storage Manager*.

O Timber utiliza as funcionalidades do *Shore* no gerenciamento de suas transações. Visto que o Shore foi desenvolvido para gerenciar objetos, isso dificulta o gerenciamento dos dados XML e ocasiona um baixo nível de controle de concorrência.

Natix

O Natix é um BDXN desenvolvido pela Universidade de Mannhein [5]. Seu foco consiste em um gerenciamento eficiente de dados estruturados em árvore a nível de páginas do sistema operacional. O modelo de armazenamento do Natix leva em consideração a semântica dos relacionamentos entre os nós. Quanto maior o grau de relacionamento entre dois nós, mais próximos, fisicamente, são armazenados. Em seu modelo, dados XML são armazenados em forma de árvores hierárquicas, assim como as árvores de arquivos em sistemas operacionais.

Teoricamente, o Natix pode ser classificado como um sistema híbrido, onde estruturas cuja granularidade é maior que um nível especificado são armazenadas numa parte estruturada do banco de dados e estruturas cuja granularidade é menor são armazenadas como objetos planos [39]. Os objetos planos armazenados são, na verdade, grupos de nós clusterizados tratados como registros atômicos pelos níveis mais baixos do sistema. A Figura 2.4 mostra a arquitetura do Natix, que é dividida em três camadas:

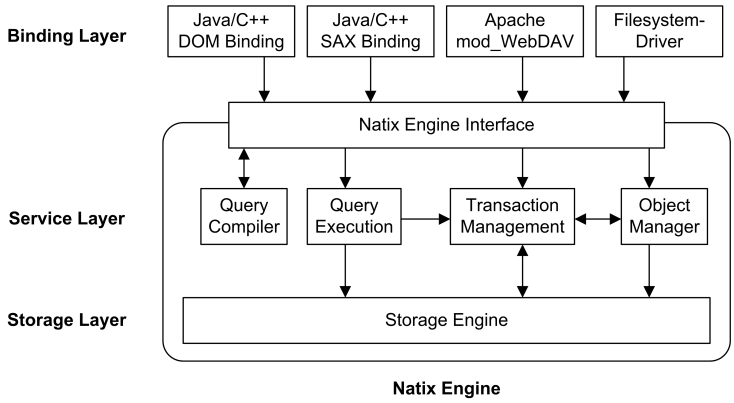

Figura 2.4 Arquitetura do Natix [5]

- Camada de armazenamento: responsável por gerenciar todas as estruturas de dados, possui classes para armazenamento eficiente de XML, bem como índices e metadados, além disso, gerencia o *log* de recuperação e controla a transferência de dados entre armazenamento principal e secundário;

- Camada de serviços: fornece todas as funcionalidades de um BD, sendo composta pela NQE (*Natix Query Engine*) - que contém a *Natix Physical Algebra* (NPA) e a *Natix Virtual Machine* (NVM), pelo *Query Compiler*, pelo *Transaction Management* e pelo *Object Manager*;

- Camada de controle: consiste de todos os módulos que mapeiam dados e requisições de uma aplicação para a interface do Natix e vice-versa. Com essa camada, documentos e resultados de consultas podem ser acessados como arquivos regulares. A estrutura em árvore do documento é mapeada em um diretório hierárquico, podendo, assim, ser manipulado por qualquer *software* que saiba operar com o sistema de arquivos.

A álgebra física do Natix opera sobre seqüências de tuplas, onde uma tupla consiste de uma seqüência de valores de atributos, e estes podem ser números, *strings* ou *handlers* de nós (para qualquer tipo de nó) [40]. As operações de atualização são baseadas no modelo DOM.

O gerenciador de transações possui classes que garantem as propriedades ACID e componentes que auxiliam na recuperação em caso de falhas. Para garantir a seriabilidade das transações, o Natix usa o protocolo 2PL estrito. O nível de granularidade segue a forma como os dados foram armazenados, ou seja, são blocos de nós agrupados.

Tamino

O Tamino é um BDXN desenvolvido pela Software AG [41], 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*. A Figura 2.5 mostra a arquitetura desse banco.

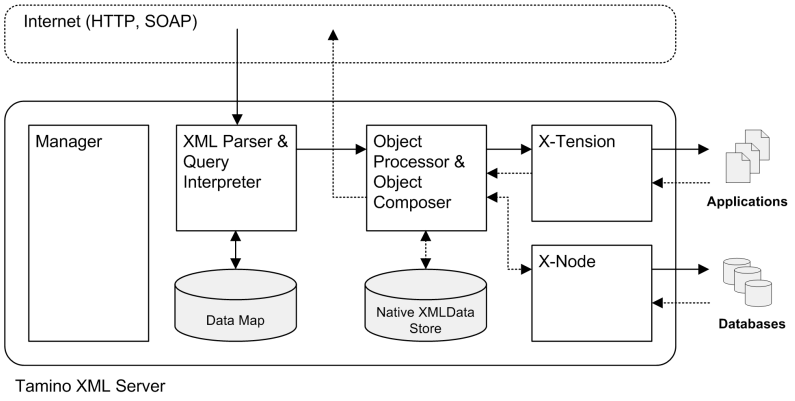

Figura 2.5 Arquitetura do Tamino [8]

O Tamino pode acessar dados externos através do componente *X-Node*. O componente *X-Tension* permite ao usuário passar documentos XML para funções definidas por ele ou ler dados dessas funções para inclusão nesses documentos. O módulo *Object Processor* é responsável por enviar documentos e comandos ao servidor. O módulo *XML Parser* recebe os comandos e os avalia de acordo com os metadados contidos no repositório *Data Map*.

Como linguagem de consulta, o Tamino aceita a linguagem XQuery, tendo seu motor de execução formado por operações padrões, tais como junção, ordenação e *scanning*, dentre outras. Os documentos XML armazenados no Tamino são agrupados em coleções. Para cada documento a ser armazenado, o Tamino determina qual o seu esquema correspondente, baseado no tipo do elemento raiz do documento. Como o esquema possui um modelo de conteúdo aberto, os documentos podem ter elementos e atributos que não estão declarados no esquema. Caso nenhum esquema apropriado seja encontrado, o documento é armazenado sem nenhuma verificação de esquema.

A álgebra utilizada opera sobre listas de tuplas ordenadas e, para isso, seu mod-

elo de dados provê estruturas de tabela, que são coleções ordenadas de tuplas com um conjunto ordenado de variáveis com tipos e nomes definidos. Neste banco, as operações de atualização são baseadas em uma extensão da linguagem XQuery.

O Tamino possui suporte completo a transações e seu controle de concorrência é baseado no modelo DOM. Como já discutimos, essa abordagen não é adequada, pois o número de conflitos é elevado e, em geral, a execução de consultas, tais como XQuery apresenta baixo desempenho.

X-Hive

O X-Hive/DB [9] é um BDXN 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 *Web-Service*, possuindo também diversas outras interfaces de acesso. A arquitetura do banco é ilustrada na Figura 2.6.

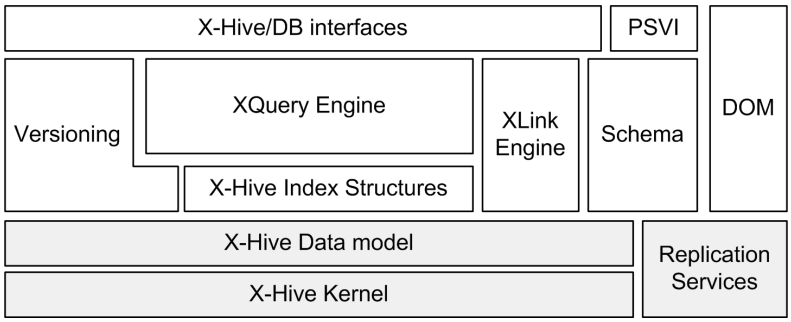

Figura 2.6 Arquitetura do X-Hive [9]

O X-Hive/DB pode ser usado para armazenar não somente documentos XML, independente da existência de um DTD ou Schema, mas também objetos multimídia como imagens, sons e outros documentos. Ele suporta vários métodos de indexação, como índices por atributos, por nomes de elementos, por valores, *full-text*, entre outros. Um mecanismo de versão permite acompanhar alterações nos documentos XML.

O modelo de transações e controle de concorrência do X-Hive é similar ao BDXN Tamino. Uma diferença importante é que o X-Hive bloqueia todo o documento, ocasionando um baixo desempenho. Para solucionar isso, o X-Hive utiliza um gerenciamento eficiente das requisições, mantendo-as em filas para um processamento posterior.

eXist

eXist [42] é um BDXN 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 armazenados de acordo com o modelo DOM.

O eXist possui diversas formas de gerenciamento e pode ser embutido em aplicações Java, integrado a servidores de aplicações, além de possuir diversas interfaces de acesso, tais com SOAP e XML:RPC. A Figura 2.7 ilustra a arquitetura deste banco.

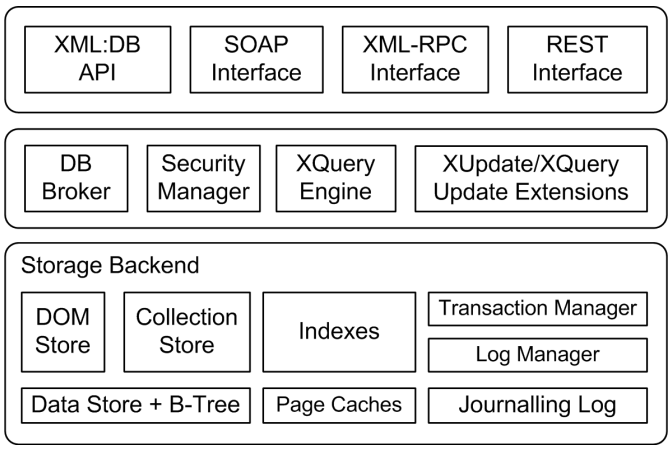

Figura 2.7 Arquitetura do eXist [10]

O eXist usa quatro arquivos de índice em seu núcleo de armazenamento de dados. Todos os índices são baseados em árvores B+ e os elementos, atributos e palavras-chave são organizados por coleção e não por documento, o que traz um ganho de desempenho considerável, visto que usuários normalmente consultam coleções inteiras e diferentes de uma vez. O *Collection Store* gerencia a hierarquia de coleções e mapeia nomes de coleções para os objetos correspondentes. O *DOM Store* representa o componente central da arquitetura de armazenamento nativo do eXist e consiste em um arquivo no qual todos os nós dos documentos são armazenados. Um índice mapeia nomes de elementos e atributos para identificadores de nós. Há também um índice invertido para registrar ocorrências de palavras nos documentos e possibilitar buscas textuais.

O eXist possui suporte a transações e um controle de concorrência simples, que bloqueia todo o documento a cada consulta submetida. Esse suporte é limitado a funcionalidades utilizadas em recuperação de *crash*. Isso significa que transações não são

visíveis nem utilizáveis no código das aplicações, o que dificulta seu uso, já que as aplicações devem gerenciar suas próprias transações.

Sedna

O Sedna [43] é um BDXN desenvolvido pelo Institute for System Programming da Rússia. Sedna tem suporte a todos os serviços dos bancos tradicionais, tais como gerenciamento de memória externa, processamento de consultas e atualização, transações e controle de concorrência, otimização de consultas, acesso através de linguagens de programação, entre outras. A Figura 2.8 mostra a arquitetura do Sedna.

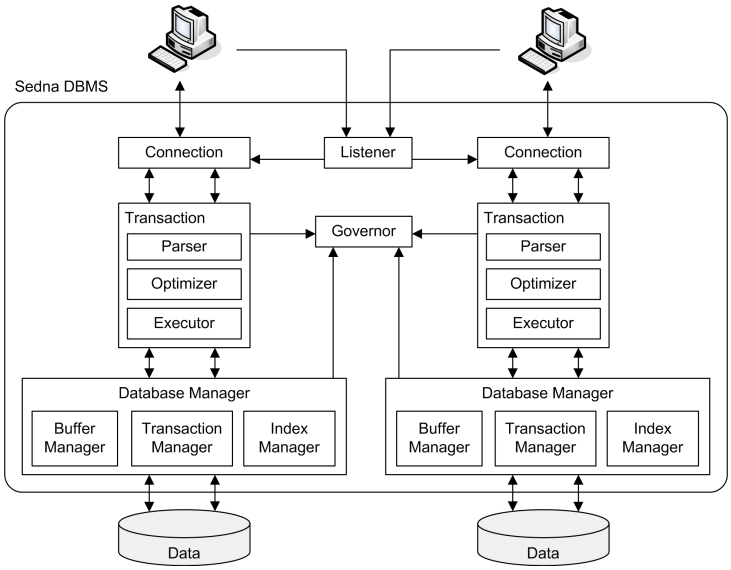

Figura 2.8 Arquitetura do Sedna [43]

O *Governor* é o controlador do sistema, gerenciando os demais componentes e controlando as bases de dados e as transações em execução. O *Listener* cria uma instância do componente *Connection* para cada cliente e inicia uma conexão direta entre o cliente e esse componente. O *Connection* encapsula a sessão do cliente e cria uma instância do componente *Transaction* para cada requisição de início de transação. O *Transaction* engloba os seguintes componentes de execução de consulta: *Parser*, *Optimizer* e *Executor*. O *Parser* traduz a consulta para sua representação lógica, que é uma árvore de operações similar ao núcleo da XQuery. O *Optimizer* recebe a representação lógica da consulta e produz um plano de execução otimizado, que é uma árvore de operações de baixo nível sobre estruturas de dados físicas. O plano de execução é interpretado pelo *Executor*. Cada instância do *Database Manager* encapsula uma única base de dados e consiste de serviços

de gerenciamento como o *Index Manager*, que gerencia os índices, o *Buffer Manager*, que é responsável pela interação entre o disco e a memória principal, e o *Transaction Manager*, que provê o controle de concorrência.

O armazenamento de dados é baseado em um esquema descritivo, que consiste em nós agrupados de documentos XML de acordo com suas posições nesse esquema. Ponteiros diretos são usados para representar a relação entre os nós de um documento, tais como relacionamento entre pai, filhos e irmãos. Para prover suporte a operações de atualização, o Sedna implementa uma extensão da linguagem XQuery.

O Sedna utiliza o protocolo 2PL estrito para garantir a seriabilidade das transações. O controle de concorrência baseia-se no esquema descritivo e o bloqueio dos dados é efetuado sobre um conjunto de nós, onde cada um desses conjuntos recebe um tipo de bloqueio (compartilhado ou exclusivo) [44].

2.3 CONCLUSÃO

Este capítulo apresentou os Bancos de Dados XML Nativos, destacando suas principais características e alguns desses sistemas. Também foram investigados aspectos que influenciaram na definição do RepliX, tais como a complexidade das linguagens de consultas, fragmentação e controle de concorrência a dados XML. A observação desses aspectos é extremamente importante e possibilita fazer escolhas adequadas na implementação dos protocolos de replicação para dados XML.

CAPÍTULO 3

REPLICAÇÃO

Este capítulo discute a replicação de dados, destacando seus principais protocolos. Em seguida, são abordados os conceitos de comunicação em grupo e, por fim, são descritos e analisados os trabalhos relacionados a esta dissertação.

3.1 REPLICAÇÃO DE DADOS

Replicação é um tópico cada vez mais importante no contexto de banco de dados e serve sobretudo para aumentar a disponibilidade do sistema em caso de falha, permitindo redirecionar os clientes para as réplicas operacionais [45]. Por outro lado, oferece também melhorias na escalabilidade, ao permitir a execução paralela de requisições de clientes nas diferentes réplicas. Finalmente, pode permitir uma menor latência no acesso, explorando a localidade dos dados.

Embora replicação seja um conceito intuitivo, sua implementação requer técnicas sofisticadas. Isso ocorre pela dificuldade de manutenção da consistência de réplicas: quando um dado é alterado, suas réplicas também precisam ser atualizadas para manter um estado distribuído consistente [46]. Para manter a consistência das réplicas, são necessários protocolos específicos ou protocolos de replicação.

Gray et al. [6] classificou os protocolos de replicação de bancos de dados usando dois parâmetros. O primeiro parâmetro estabelece a forma de propagação das modificações, que pode ser de forma síncrona ou assíncrona. O segundo parâmetro indica quem pode propagar as atualizações: uma réplica específica, chamada de *cópia primária* ou qualquer uma das réplicas, onde cada uma destas é denominada *réplica ativa*.

3.1.1 Sincronismo entre Réplicas

Várias estratégias de sincronização entre as réplicas podem ser adotadas. A escolha da melhor estratégia depende da disponibilidade de comunicação, do grau de atualização e do volume das informações requisitadas pelos usuários. As principais formas de sincronismo são a síncrona e a assíncrona.

Na forma síncrona, quando uma réplica é alterada, essa alteração é imediatamente aplicada às demais réplicas dentro de uma transação [32]. Nesse caso, as réplicas cooperam usando estratégias para manter a consistência das réplicas. Sistemas de banco de dados síncronos tradicionalmente utilizam o protocolo de *two-phase commit* (2PC) [32]. Neste protocolo, uma réplica é encarregada de coordenar (fase 1) e confirmar (fase 2) a difusão das modificações para as demais. Esse tipo de consistência é muito dispendiosa, principalmente quando o número de participantes é grande, pois o grau de comunicação na rede é alto, e todos os participantes devem estar conectados. Em [6] foi provado que o protocolo 2PC é impraticável quando a quantidade de réplicas é grande, pois o número de *aborts*, *deadlocks* e mensagens trocadas cresce de maneira exponencialmente proporcional ao número de réplicas.

Na forma assíncrona, a alteração de uma réplica não é propagada imediatamente, sendo realizada em um momento posterior, dentro de uma transação separada. A propagação das atualizações pode ser realizada de forma linear ou constante. A primeira consiste em enviar as atualizações a cada transação. A segunda consiste em definir intervalos de tempo configuráveis 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.

3.1.2 Protocolos de Replicação

No protocolo de cópia primária, uma das réplicas é escolhida como cópia ou réplica primária e as outras cópias são réplicas secundárias. Essa cópia primária gerencia as demais e envia a resposta da operação para o cliente. Este protocolo apresenta um baixo poder de processamento, já que apenas a cópia primária realiza o processamento [32]. Além disso, possui algumas desvantagens como a necessidade de escolha de um nova primária, no caso de falha, e tempos de respostas inaceitáveis, quando a primária

torna-se um gargalo, pois ela centraliza todas as operações de atualização [47], afetando a escalabilidade.

No protocolo de réplicas ativas, qualquer uma das réplicas pode executar operações de atualização [32]. Essas operações são executadas na mesma seqüência por todas as réplicas, produzindo resultados idênticos. Não há uma réplica centralizadora, como no protocolo de cópia primária. Esse protocolo apresenta a vantagem de ser tolerante a falhas, já que não existe uma cópia primária, e de apresentar melhor desempenho, pois várias réplicas podem ser acessadas de forma concorrente. A principal desvantagem desse protocolo é a necessidade de um mecanismo que assegure a consistência entre as réplicas quando atualizações são executadas.

3.1.3 Transações em Banco de Dados Replicados

Uma transação é uma seqüência de operações sobre itens de dados que satisfaz as propriedades ACID [48]. Em geral, em um ambiente replicado, as transações de leitura são resolvidas localmente e as de atualização sobre as réplicas resultarão em uma transação distribuída, que atualiza automaticamente as réplicas de um item de dado, mantendo a consistência [34].

As dependências entre transações representam um importante problema em bancos de dados replicados, já que clientes podem acessar o mesmo item de dado em diferentes réplicas [34]. Tratar dependências entre transações requer detectar e impedir situações de execução de transações que possam gerar conflitos no estado das réplicas. Duas ou mais operações executadas por diferentes transações estão em conflito se ambas acessam concorrentemente o mesmo objeto e pelo menos uma delas executa uma operação de atualização [48]. Bancos centralizados usam o protocolo *two-phase locking* (2PL) para sincronizar as transações localmente, garantindo que clientes concorrentes não acessam nem alteram os mesmos dados.

Em bancos de dados replicados, para que um banco b_i confirme localmente uma transação distribuída t_a, b_i precisa verificar se a transação distribuída t_a não está em conflito com as transações executadas sobre as demais réplicas. Segundo Pedone et al. [13], uma transação t_b está em conflito com uma transação t_a, se t_a e t_b têm operações em conflito e se t_b não precede t_a. Observe que t_a representa uma transação distribuída a ser confirmada no banco de dados e, t_b representa qualquer transação (local ou distribuída)

que possa estar sendo executada.

3.2 COMUNICAÇÃO EM GRUPOS

A abstração de grupos [11] tem provado ser um mecanismo eficiente para implementar protocolos de consistência de réplicas, facilitando o desenvolvimento de aplicações distribuídas confiáveis. Essa abstração tem por objetivo resolver problemas básicos de inconsistências na comunicação entre processos distribuídos que cooperam para a execução de uma tarefa. Nesse sentido, um grupo é apenas um conjunto de processos que cooperam. Na forma mais geral de comunicação em grupos, um determinado processo pode pertencer a diferentes grupos, ou seja, os grupos podem se sobrepor.

A existência de múltiplos membros em um grupo é invisível para o cliente. Um cliente que deseja se comunicar com um grupo, ao invés de enviar uma mensagem individual para cada membro, envia apenas uma mensagem para o endereço do grupo. A abstração de grupos evita que um processo externo tenha que conhecer individualmente cada um dos membros.

O serviço baseado em grupos é composto de duas partes: *group membership* e *group communication* [11]. O serviço de *membership* provê aos processos a composição do grupo. Tal composição evolui de acordo com o interesse dos processos de se unirem (*join*) ou sairem (*leave*) do grupo e com a verificação da ocorrência de falhas, sejam de processos pertencentes ao grupo ou de canais de comunicação. Esse serviço abstrai os eventos que provocam mudanças na composição do grupo, através da entrega a cada membro de uma "visão", ou seja, a composição do grupo em um determinado instante, atualizada e composta por todos os processos considerados ativos. Essas visões são atualizadas de forma coerente de tal maneira que os processos que as instalam concordam com a sua composição.

O objetivo do serviço de *communication* é prover aos processos primitivas de comunicação necessárias à troca eficiente de mensagens. A principal primitiva oferecida por esse serviço é a de *multicast* ou difusão. Quando uma entidade externa invoca essa operação a um grupo, essa primitiva garante a entrega da mensagens a todos os membros do grupo. Uma mensagem enviada pode ser confiável ou não-confiável. No primeiro caso, garante-se que a mensagem será recebida por todos os processos não-falhos ou por nenhum deles, garantindo a propriedade da atomicidade. Com essa propriedade, o processo que

envia a mensagem para um grupo irá receber uma mensagem de erro se um ou mais integrantes do grupo tiver problema no recebimento.

O serviço de grupos também deve garantir alguma forma de integração entre a entrega de invocações regulares (*multicast*) e a entrega de eventos de mudança de visão. Dessa forma, tais serviços fazem uso do paradigma de *View Synchronous Communication* [11]. Esse paradigma garante que as mudanças de visão e *multicasts* sejam entregues na mesma ordem para todos os processos do grupo que recebem o mesmo conjunto de mensagens.

Classificação

Grupos podem ser classificados segundo suas características. A principal classificação é quanto à facilidade em refletir ou não as alterações do sistema distribuído [47]. Em grupos estáticos, não ocorre qualquer alteração na quantidade de membros durante a vida útil do sistema, não sendo necessário um serviço para controlar a entrada e saída de membros. Grupos dinâmicos sofrem alterações, refletindo entradas e saídas de membros, ou seja, qualquer alteração no número de membros de um grupo é indicada na visão do grupo.

Os grupos também podem ser classificados quanto à forma na qual o *multicast* é implementado [11]. Se todos os membros estão aptos a fazer o *multicast*, sem necessitar de um membro coordenador, o grupo é não-hierárquico ou simétrico. Quando o grupo precisa de um membro coordenador para realizar o seqüenciamento de mensagens, o grupo é assimétrico ou hierárquico. Neste caso, apenas o coordenador precisa estar apto a fazer o *multicast*. Os demais membros que desejam realizar um *multicast* enviam uma mensagem ponto-a-ponto para o coordenador. Além dessas classificações, os grupos podem ser fechados, onde os membros recebem somente mensagens de outros membros, ou abertos, no qual os membros recebem mensagens de qualquer processo.

Ordenação de Mensagens

Uma mensagem recebida por um membro do grupo que está funcionando corretamente passa por um processo de entrega que eventualmente precisa ser controlado para que a seqüência de entrega seja a mesma seqüência de envio [11]. Os tipos de ordenação que normalmente estão implementadas nos sistemas de CG são:

- FIFO: mensagens enviadas a partir de um único membro são entregues na ordem em que são enviadas a todos os membros do grupo. Por exemplo, para todas as mensagens M_1 e M_2 e todos os processos P_i e P_j, se P_i envia M_1 antes de enviar M_2, então M_2 não é recebida em P_j antes que M_1 seja recebida.

- Causal: mensagens enviadas a partir de múltiplos membros que possuam uma relação de causalidade são entregues a todos os membros do grupo respeitando sua ordem de causalidade. Duas mensagens possuem relação de causalidade caso uma delas tenha sido gerada depois do recebimento da outra. Pode ser necessário preservar essa ordem em todos os participantes, já que o conteúdo da segunda mensagem pode ser afetado pelo processamento da primeira.

- Total: todas as mensagens enviadas ao grupo são entregues a todos os membros na mesma ordem. Por exemplo, para todas as mensagens M_1 e M_2 e todos os processos P_i e P_j, se M_1 é recebida em P_i antes que $M2$ seja, então M_2 não é recebida em P_j antes que M_1 seja.

3.2.1 Sistemas de Comunicação em Grupos

Ensemble

O Ensemble [11] é uma sistema de CG desenvolvido na Universidade de Cornell, usando a linguagem de programação *Objective Caml*. Este sistema proporciona um ambiente para composição de camadas de micro-protocolos. Camadas adjacentes comunicam-se através da troca de eventos (certos tipos de eventos são descendentes enquanto outros são ascendentes) e cada uma delas processa apenas os eventos que lhe são relevantes, entregando-os depois à camada seguinte. Ao contrário de outros modelos de pilhas de protocolos como, por exemplo, o OSI, a aplicação não reside no topo da pilha. Os micro-protocolos podem interagir diretamente com a aplicação disponibilizando interfaces laterais. A Figura 3.1 mostra a arquitetura do Ensemble.

As *interfaces* são concretizadas através da definição de funções que permitem o estabelecimento de comunicação bidirecional entre o Ensemble e a aplicação: o Ensemble invoca a função para indicar novos eventos, e a aplicação retorna as ações que pretende executar. A mudança de configuração da pilha em tempo de execução integra o conjunto de protocolos da plataforma. A nova configuração é proposta pela aplicação, assegurando

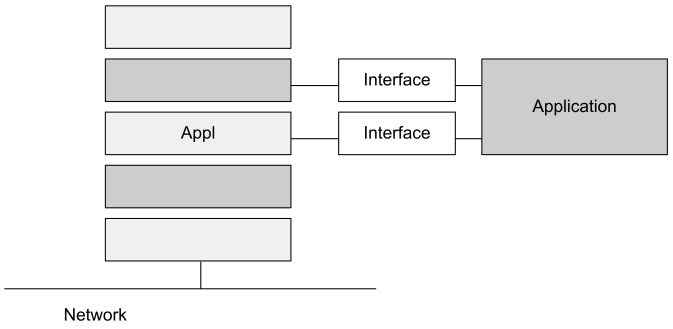

Figura 3.1 Arquitetura do sistema Ensemble [11]

ao Ensemble a simultaneidade da sua execução por todos os membros do grupo. O processo de entrada de um ponto de acesso em um grupo é executado pela invocação da função de criação da pilha e o abandono é executado através do envio de uma mensagem à pilha.

Spread

O Spread [49] é um sistema de CG desenvolvido pela Universidade de Johns Hopkins, implementado na linguagem C e portado para vários sistemas operacionais. Esse sistema é baseado no modelo *daemon*, onde os *daemons* definem a rede de disseminação de mensagens e provêem serviços de gerência de membros e ordenação. As aplicações clientes, que são ligadas a uma biblioteca, podem residir em qualquer local da rede e conectam-se ao *daemon* mais próximo para ter acesso aos serviços de comunicação em grupos.

A arquitetura do Spread é apresentada na Figura 3.2. As aplicações utilizam a biblioteca *SP-lib* ou um conjunto de classes Java para fazer uso da interface de comunicação do Spread. A conexão entre a *SP-lib* e o *daemon* é feita através de uma conexão ponto-a-ponto confiável, empregando a rede de comunicação ou comunicação inter-processos. Os módulos *Session* e *Groups* são responsáveis por gerenciar as conexões e controlar os grupos e seus membros.

A área sombreada da arquitetura mostra os protocolos internos do *daemon*. Entre o módulo *Session* e o módulo *Transport*, existe um fila para cada sessão, o que permite um controle de prioridades. O módulo *Routing* computa árvores de roteamento para determinar quais enlaces, representados pelos módulos *Hop* e *Ring*, serão utilizados no envio de cada mensagem.

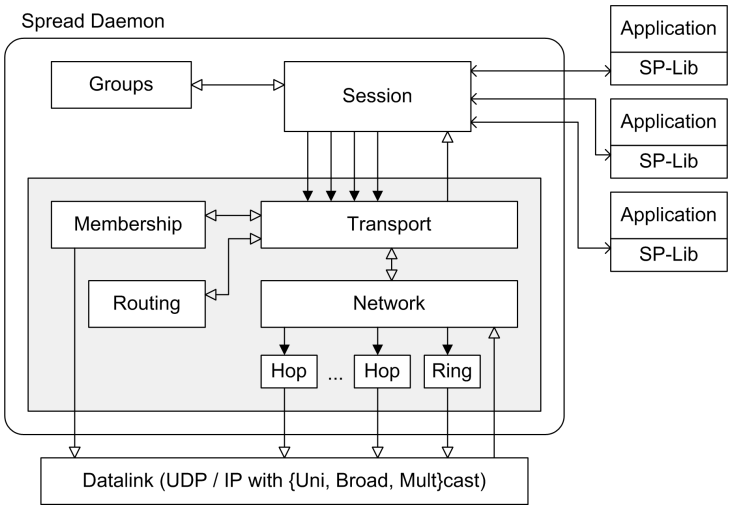

Figura 3.2 Arquitetura do sistema Spread [49]

Podem existir diversas instâncias do módulo *Hop*, cada uma representando um enlace em uma ou mais árvores de roteamento. O módulo *Ring*, por sua vez, é instanciado, no máximo, uma vez e provê disseminação e confiabilidade no grupo local se houver mais de um *daemon* ativo nesse grupo. Instâncias de *Hop* e *Ring* podem ser destruídas e instanciadas à medida que os membros dos grupos entram e saem.

JavaGroups

O JavaGroups é um sistema de CG [50] desenvolvido na Universidade de Cornell e baseado no Ensemble. O JavaGroups implementa entrega confiável de mensagens para um grupo usando IP *multicast* em *hardware*. O IP *multicast* não necessita *software* especial para realizar a difusão de mensagens sobre uma rede de comunicação ponto-a-ponto.

Para enviar e receber mensagens usando o serviço de comunicação do JavaGroups, um processo precisa ser conectado a um canal lógico confiável. Um canal é gerado quando o primeiro processo se conecta a esse canal. Os demais processos conectam-se a esse mesmo canal para formar o grupo. Assim sendo, os processos conectados a um mesmo canal recebem todas as mensagens enviadas para o grupo.

Quando um canal é criado, as propriedades do canal são especificadas para todos os membros. Essas propriedades são implementadas individualmente por protocolos que compõem uma pilha de protocolos semelhante à arquitetura de camadas usadas no sistema Ensemble. Cada protocolo implementa uma propriedade específica, relacionada com

entrega confiável de mensagens ou com o serviço de composição de grupos. A Figura 3.3 ilustra como é feita a interação entre uma aplicação e o JavaGroups.

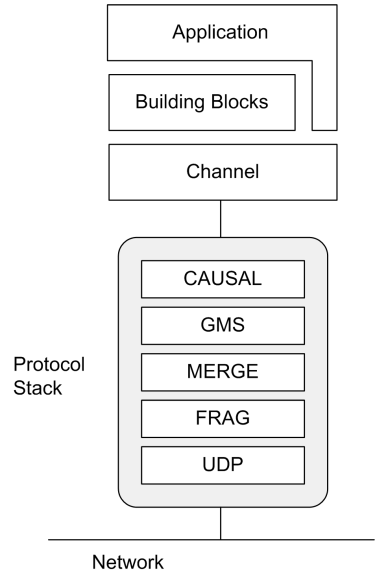

Figura 3.3 Arquitetura do sistema Java Groups [50]

Quando uma mensagem é enviada, essa atravessa todos os micro-protocolos da pilha com a finalidade de obter um serviço confiável. Por exemplo, a camada GMS implementa o serviço de composição de grupos, e a FRAG é um serviço que fragmenta mensagens grandes. A camada UDP representa o final dessa pilha, enviando mensagens entre os membros usando IP e UDP não-confiável. Essa lista de micro-protocolos é flexível, ou seja, outros protocolos podem ser acrescidos ou retirados.

3.2.2 Protocolos de Replicação Baseados em Primitivas de Grupo

Primitivas de comunicação com garantias de confiabilidade e ordenação facilitam a construção de protocolos de replicação. Protocolos baseados nessas primitivas estão sendo desenvolvidos e têm apresentados resultados promissores.

Em [12] é apresentado o Postgres-R, uma extensão do banco Postgres, que baseia-se no protocolo *Database State Machine* [13]. Este protocolo trabalha de forma síncrona e utiliza o protocolo 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 executa localmente a transação sem qualquer sincronização com as demais, evitando assim a sobrecarga associada ao controle de concorrência distribuído [6]. Em seguida, o estado associado à execução da transação é difundido com garantias de atomicidade e ordem total na entrega, utilizando primitivas de 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. [7] apresentam o protocolo RepDB, que baseia-se no protocolo 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 cargas, 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 efetivadas (*commit*) antes de T, seguindo a ordem de *timestamp* cronológico (ordenação total). O uso do *timestamp* associado às primitivas de ordenação previne os conflitos e mantém a consistência.

Já o PDBREP [15] classifica as réplicas em de atualização e de leitura e possui quatro tipos de transações: atualização, leitura, propagação e *refresh*. As réplicas de atualização processam transações de atualização, que são aquelas que contêm pelo menos uma operação de escrita. As réplicas do grupo de leitura recebem apenas transações de leitura, isto é, não realizam quaisquer operações de escrita. Tanto as réplicas de atualização como as de leitura são gerenciadas de forma assíncrona. As réplicas de leitura possuem filas locais onde são armazenadas 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, ou quando uma transação de leitura requer dados atualizados, o que causa a execução de uma transação de *refresh*.

3.3 TRABALHOS RELACIONADOS

O eXist [10] possui um mecanismo de replicação baseada no sistema de CG JavaGroups para sincronizar as réplicas. O eXist 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 do eXist, este envia a atualização para a cópia primária e bloqueia as cópias secundárias. Quando a cópia primária executa uma atualização, esta é propagada para as cópias secundárias, que posteriormente são desbloqueadas para executar as novas requisições.

O X-Hive [9] apresenta um mecanismo 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 [8] 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.

Apesar do eXist utilizar primitivas de CG, essas primitivas são utilizada 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 [34]. Nenhum dos trabalhos relacionados estudados apresenta resultados que comprovem a eficiência de seus mecanismos. Além disso, as soluções por eles apresentadas não possuem portabilidade, já que são implementadas no núcleo do banco de dados. A Tabela 3.1 mostra uma comparação entre os trabalhos relacionados.

	eXist	**X-Hive**	**Tamino**
Protocolo	Cópia Primária	Cópia Primária	Cópia Primária e Réplica Ativa
Propagação	Assíncrona	Assíncrona	Assíncrona e Síncrona
Granulosidade	Total	Total	Total

Tabela 3.1 Comparação entre os trabalhos relacionados

3.4 CONCLUSÃO

Este capítulo apresentou os conceitos e protocolos relacionados a replicação de dados, bem como os sistemas de comunicação em grupos. Além disso, foram abordadas as principais vantagens e desvantagens desses protocolos e os principais trabalhos relacionados. Nenhum dos trabalhos relacionados apresentam resultados experimentais que comprovem a eficiência dos mecanismos desenvolvidos, ponto este previsto nesta dissertação.

Acreditamos que a combinação de protocolos de replicação, juntamente com conceitos de CG é uma solução eficaz para o problema de replicação em BDXNs, pois contempla as principais características de um ambiente replicado. No próximo capítulo serão explanadas as características do RepliX, sua especificação e os algoritmos desenvolvidos.

CAPÍTULO 4

REPLIX: UM MECANISMO PARA REPLICAÇÃO DE DADOS XML

Este capítulo descreve o RepliX, um mecanismo para a replicação de dados em BDXNs, cujo propósito é melhorar a disponibilidade e o desempenho desses sistemas. Também são discutidos sua arquitetura e os algoritmos desenvolvidos.

4.1 CARACTERÍSTICAS DO REPLIX

O objetivo deste trabalho foi desenvolver um mecanismo para replicação em BDXNs, demoninado RepliX, visando melhorar a disponibilidade e o desempenho desses sistemas. O RepliX combina protocolos síncronos, assíncronos e primitivas de CG, visando obter uma estrutura eficiente para a replicação de dados XML.

O RepliX considera as principais limitações no gerenciamento de dados XML, tais como a complexidade de decomposição das linguagens de consulta, controle de concorrência e fragmentação de dados XML. O RepliX não possui os problemas dos protocolos tradicionais, tais como cópia primária e réplica ativa. Estendemos o protocolo PDBREP de forma a contemplar as características do modelo XML e adicionamos técnicas para torná-lo tolerante a falhas.

O critério *one-copy serializability* [32] é usado neste trabalho como modelo de corretude. Esse modelo garante que a execução de transações concorrentes produza 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 RepliX utiliza a seqüência de execução de transações clássica, na qual clientes iniciam uma transação enviando uma requisição *begin* para, em seguida, fazer requisições

de escrita e/ou leitura [48]. A transação é finalizada por uma requisição *commit* ou *abort*. Se a requisição *commit* for enviada e executada com sucesso, as atualizações persistem, garantindo sua sobrevivência na ocorrência de uma falha no sistema. Caso contrário, se for enviada uma requisição *abort* ou no caso de falha no sistema, as modificações da transação são canceladas. As transações concorrentes não vêem as alterações pendentes umas das outras.

4.2 ARQUITETURA DO REPLIX

O RepliX é dividido em alguns componentes de *software* que comunicam entre si, implementando o mecanismo para replicação em BDXNs. Uma visão geral da arquitetura do RepliX pode ser observada na Figura 4.1.

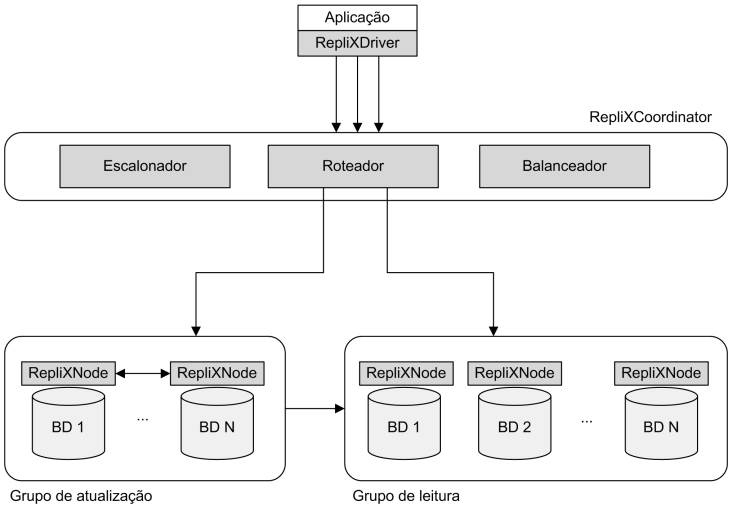

Figura 4.1 Arquitetura do RepliX

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 RepliXCoordinator 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.

4.3 ESPECIFICAÇÃO

Seja um sistema composto por N sites (nós de rede com recursos computacionais), 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. Considerando que a quantidade de sites pode ser alterada, já que sites podem ser removidos ou adicionados, o conjunto N de sites não é fixo.

A estratégia de particionar os sites 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. Com essa estratégia, apenas uma parte dos sites precisa ser atualizada a cada modificação.

O RepliX utiliza primitivas de comunicação de grupos confiável para a troca de mensagens entre os sites. Adotamos a replicação total dos dados devido às características do modelo XML: complexidade da decomposição de linguagens de manipulação de dados XML, como a linguagem XQuery, e a fragmentação de dados nesse formato. De acordo com Mattoso et al. [51], "embora a replicação total possa ser apontada como ponto fraco, é importante observar que o custo da memória secundária vem caindo drasticamente, o que torna a abordagem hoje economicamente atraente mesmo para grandes bases de dados. Além disso, esse tipo de replicação favorece a disponibilidade dos dados".

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 site deste grupo a recebe. Esse site 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 sites desse grupo. Esses sites 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.

A Figura 4.2 exibe um exemplo do grupo de atualização. O site s_2 é o site *primário* para o cliente c_1 e os sites s_1 e s_3 são os *secundários*. Já o site s_3 é o site *primário* para os clientes c_2 e c_3 e os demais sites são *secundários*. Nesse exemplo, o site s_1 atua como *secundário* em relação os demais sites.

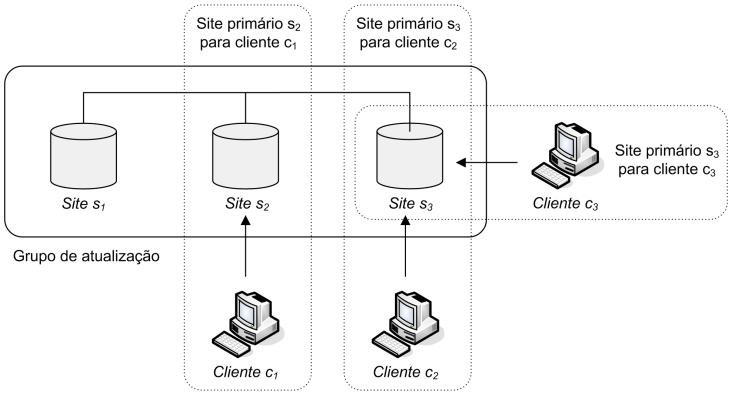

Figura 4.2 Grupo de Atualização

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

Durante a execução das transações de leitura em um determinado site, 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 site 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 site, o RepliX realiza as seguintes verificações: (i) se a nova transação requisita dados que foram atualizados, um transação de *refresh* é executada, (ii) caso contrário, a transação é executada sem a necessidade de transações de *refresh* ou propagação.

A Figura 4.3 mostra um exemplo do grupo de leitura. O site s_6 está atendendo a requisição dos clientes c_4 e c_6 , que iniciou-se antes da transação T_1 enviada pelo grupo de atualização. O site s_5 está tratando a requisição do cliente c_5, que se iniciou antes da transação T_2 e que recebeu a transação de propagação T_1. Como o site s_4 estava ocioso, ele já aplicou as transações T_1 e T_2 e encontra-se atualizado. Para evitar conflito durante a execução das transações de leitura e *refresh*, as de *refresh* bloqueiam os dados a serem modificados, utilizando um bloqueio compatível, chamado de *bloqueio de refresh*. Apesar de utilizar protocolos assíncronos, da perspectiva do usuário, o RepliX funciona de forma síncrona, pois os usuários sempre acessam dados atualizados.

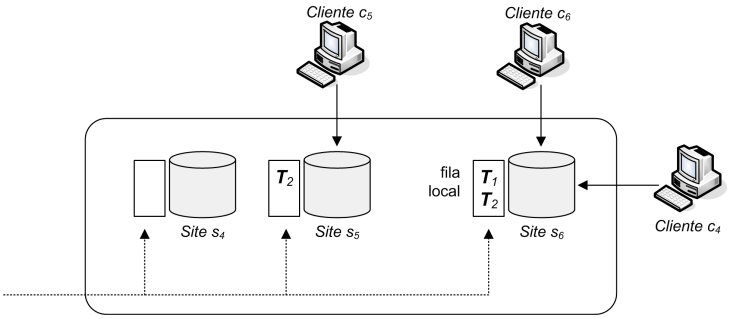

Figura 4.3 Grupo de Leitura

4.4 ALGORITMOS PARA REPLICAÇÃO DE DADOS XML

Esta seção descreve os principais algoritmos do RepliX. O Algoritmo 1 é executado pelo componente RepliXCoordinator. Ao receber uma transação (linha 3), este componente analisa suas operações (linha 4) para, em seguida, encaminhá-la a um dos sites do grupo apropriado. Essa análise da transação é feita verificando o conteúdo de cada uma das suas operações. O coordenador gerencia os sites de cada um dos dois grupos e utiliza um algoritmo *round-robin* para escolher o site que receberá a transação, com o objetivo de distribuir a carga entre os sites. Esse algoritmo está implementado para o grupo de atualização (linhas 5 a 8) e para o de leitura (linhas 10 a 13). Se a transação contém operações de atualização, o próximo site do grupo de atualização é escolhido (linha 5) e

a variável que indica o número do próximo site de escrita é incrementada (linha 6). Em seguida, o coordenador envia a transação ao site escolhido que a executa localmente e retorna um objeto remoto correspondente a transação iniciada (linha 7). O coordenador, então, retorna esse objeto ao cliente (linha 8) que pode utilizá-lo para examinar os resultados da transação e fazer um *commit* ou *abort*. No caso de uma transação de leitura (linha 9), o processo ocorre de maneira semelhante (linhas 10 a 13).

Algoritmo 1 - Procedimento executado pelo Coordenador

```
 1: procedure processar_transacoes
 2:     loop
 3:         transacao ← coordenador.recebe_transacao;
 4:         if transacao contem operacao de atualizacao then
 5:             site ← coordenador.get_site(coordenador.numero_site_escrita);
 6:             coordenador.numero_site_escrita ← coordenador.numero_site_escrita + 1;
 7:             transacao_remota ← site.begin(transacao);
 8:             return transacao_remota;
 9:         else
10:             site ← coordenador.get_site(numero_site_leitura);
11:             coordenador.numero_site_leitura ← coordenador.numero_site_leitura + 1;
12:             transacao_remota ← site.begin(transacao);
13:             return transacao_remota;
14:         end if
15:     end loop
16: end procedure
```

As operações realizadas pelo site *primário* são mostradas no Algoritmo 2. O site *primário* recebe continuamente mensagens enviadas por seus clientes e pelo sistema de CG (linha 3). Se a mensagem indica a existência de uma nova visão (linha 4), esta é prontamente instalada, substituindo a visão anterior (linha 5). Essa mensagem é proveniente do sistema de CG, enquanto as mensagens de *begin* e *commit* de transações são originadas dos clientes do site *primário*. No caso da mensagem de *begin* (linha 6), a transação é iniciada no banco de dados local (linha 7), sendo executadas suas operações e os resultados retornados ao cliente (linha 8). Quando o cliente requisita o *commit* da transação (linha 9), as operações de atualização (*write set*) da transação são enviados aos outros sites do grupo de atualização (sites *secundários*) utilizando uma primitiva de ordenação total do sistema de CG (linha 10). Em seguida, o site *primário* aguarda as confirmações dos *testes de certificação* executados nos sites *secundários* da visão atual (linha 11). Se ocorrerem conflitos no teste de certificação (linha 12), o site *primário* envia um *multicast* com a mensagem de *abort* para os sites secundários (linha 13). Caso não ocorram conflitos, é feito um *multicast* aos sites secundários para que estes realizem o *commit* da transação nos bancos de dados locais (linha 15). Na seqüência, o site *primário*

faz o *commit* localmente (linha 17) e envia o *write set* da transação aos sites do grupo de leitura (linha 18). Posteriormente, os sites de leitura efetivarão essas modificações em seus bancos de dados locais.

Algoritmo 2 - Procedimento executado pelo Site Primário

```
 1: procedure processar_transacoes
 2:     loop
 3:         msg ← site_primario.recebe_mensagem();
 4:         if msn.tipo=nova_visao then
 5:             site_primario.instala_nova_visao;
 6:         else if msn.tipo=begin then
 7:             site_primario.begin;
 8:             resultados ← site_primario.execute(msg.transacao, msg.operacoes);
 9:         else if msn.tipo=commit then
10:             site_primario.multicast(grupo_atualizacao, msg.transacao, write_set);
11:             site_primario.espera_confirmacao_certificacao;
12:             if encontrou conflito nas certificacoes then
13:                 site_primario.multicast(grupo_atualizacao, msg.transacao, abort);
14:             else
15:                 site_primario.multicast(grupo_atualizacao, msg.transacao, commit);
16:             end if
17:             site_primario.commit;
18:             primario.multicast(grupo_leitura, msg.transacao, write_set);
19:         end if
20:     end loop
21: end procedure
```

O Algoritmo 3 descreve o comportamento dos sites *secundários*. Ao receber uma mensagem enviada pelo *primário* (linha 3), caso a mensagem contenha um *write set* (linha 4), é executado um *teste de certificação* (linha 5) para verificar se as atualizações conflitam com alguma transação em execução e, em caso negativo, é enviada uma mensagem de confirmação ao site *primário* (linha 6). Os sites que não responderam, por motivos de falha no site ou de comunicação, são excluídos do grupo, através do envio de uma nova visão. Esses sites podem ser reintegrados posteriormente ao grupo. Se a mensagem recebida for de *commit* (linha 7), uma transação é iniciada no banco de dados local (linha 8), e as operações do *write set* recebido anteriormente são executadas (linha 9). Em seguida, o *commit* da transação é feito (linha 10). Se a mensagem recebida for de *abort* (linha 12), o site secundário aborta a transação (linha 12).

O Algoritmo 4 mostra o *teste de certificação* utilizado no site *secundário*. Esse teste verifica conflitos entre as transações, comparando seus *read sets* e *write sets* (linhas 3 a 5). Depois do *teste de certificação*, a transação é confirmada, abortando transações locais (nos sites *secundários*) em execução que estão em conflito com a transação enviada

pelo *primário*.

Algoritmo 3 - Procedimento executado pelos Sites Secundários

```
 1: procedure rececе_transacoes
 2:     loop
 3:         msg ← site_secundario.recebe_mensagem();
 4:         if msg.tipo =write_set then
 5:             resultado ← site_secundario.teste_de_certificao(msg.operacoes);
 6:             return   resultado;
 7:         else if msg =commit then
 8:             site_secundario.begin;
 9:             site_secundario.execute(msg.transacao, msg.write_set);
10:             site_secundario.commit;
11:         else if msg =abort then
12:             site_secundario.abort(msg.transacao);
13:         end if
14:     end loop
15: end procedure
```

Algoritmo 4 - Procedimento do Teste de Certificação

```
 1: procedure teste_de_certificacao
 2:     for transacoes em site_secundario do
 3:         if (operacoes.read_set= transações.operacoes.write_set) or
 4: (operacoes.write_set= transações.operacoes.read_set) or
 5: (operacoes.write_set= transações.operacoes.write_set) then
 6:             return   falso;
 7:         end if
 8:     end for
 9: end procedure
```

O Algoritmo 5 descreve o comportamento dos sites de leitura. O site de leitura recebe mensagens (linha 3) de clientes ou do grupo de atualização. Ao receber mensagens de *begin* (linha 4), provenientes de clientes, a fila é inspecionada, verificando se existem modificações pendentes recebidas do grupo de atualização a serem efetivadas (linha 5). Em caso positivo, os itens de dados do site a serem atualizados são bloqueados (linha 6), ou seja, estes itens de dados não podem ser acessados por outras transações até que sejam desbloqueados, para a execução de uma transação de *refresh* (linha 7) com as atualizações presentes na fila. Esse é o bloqueio de *refresh*. Ao final da execução, os dados do site são desbloqueados (linha 8). Em seguida, a transação é iniciada no banco de dados local (linha 10), suas operações são executadas dentro da transação iniciada (linha 11) e os resultados retornados ao cliente (linha 12). Caso a mensagem seja de atualização (linha 13), originada de um dos sites do grupo de atualização, o *write set* recebido é colocado no fim da fila do site de leitura (linha 14) para, posteriormente, ser efetivado através de uma transação de *refresh* ou de propagação.

Algoritmo 5 - Procedimento executado pelos Sites de Leitura

```
 1: procedure processar_transacoes
 2:     loop
 3:         msg ← site_leitura.recebe_mensagem;
 4:         if msg.tipo=begin then
 5:             if fila.esta_vazia()=false then
 6:                 site_leitura.bloquear(msg.item_de_dados);
 7:                 site_leitura.executa_transacao_refresh(fila);
 8:                 site_leitura.desbloquear(msg.item_de_dados);
 9:             end if
10:             site_leitura.begin;
11:             resultado ← site_leitura.execute(msg.operacoes);
12:             return   resultados;
13:         else if msg.tipo=atualizacao then
14:             fila.mensagem.(msg.write_set);
15:         end if
16:         while site_leitura.esta_ocioso()=true do
17:             if fila.esta_vazio=false then
18:                 site_leitura.bloquear(msg.item_de_dados);
19:                 site_leitura.executar(transacao_propagacao_fila);
20:                 site_leitura.desbloquear(msg.item_de_dados);
21:             end if
22:         end while
23:     end loop
24: end procedure
```

Nos momentos em que o site se encontra ocioso (linha 16), isto é, não está executando nenhuma transação, é verificado se existem atualizações pendentes na fila (linha 17). Caso existam, os itens de dados a serem atualizados são bloqueados (linha 18), e uma transação de propagação com as alterações contidas na fila é executada (linha 19). Os itens de dados são bloqueados para impedir que novas transações sejam executadas antes da transação de propagação terminar, impedindo que aquelas vejam dados desatualizados. Ao final da execução, os itens de dados do site são desbloqueados (linha 20) e este fica disponível para executar as transações pendentes.

4.5 CONCLUSÃO

Este capítulo apresentou o RepliX, um mecanismo de replicação para Banco de Dados XML Nativos, cujo objetivo é melhorar a disponibilidade, o desempenho e a escalabilidade destes bancos. Foram elencados quais pressupostos são considerados pelo RepliX e sua arquitetura foi descrita. A especificação e os algoritmos desenvolvidos também foram apresentados.

CAPÍTULO 5

IMPLEMENTAÇÃO E AVALIAÇÃO

Este capítulo descreve os componentes do RepliX e a avaliação realizada. São apresentados os detalhes de implementação e, ao final, os resultados obtidos considerando diversas características de bancos de dados replicados.

5.1 ASPECTOS DE IMPLEMENTAÇÃO

Antes de implementar o RepliX, realizamos uma análise dos BDXNs estudados com o objetivo de identificar o banco mais adequado no contexto deste trabalho. Devido à forma como o Timber foi desenvolvido (sobre o sistema *Shore*), este banco não possui suporte nativo a transações e alterá-lo é uma tarefa muito difícil. O Natix apresenta-se como um banco interessante, mas possui uma estrutura complexa, dificultando sua modificação. O Tamino e o X-Hive possuem suporte completo a transações, mas, por tratarem-se de sistemas proprietários, não podem ser alterados. Já o eXist é um banco de código aberto e relativamente simples de alteração, mas não possui suporte completo a transações, o que dificulta sua utilização. O Sedna apresentou-se como o melhor banco, pois possui suporte completo a transações, além de ser um banco de código livre, podendo ser alterado e distribuído com as modificações.

Com relação ao sistema de comunicação em grupo, escolhemos o sistema Spread, baseado no estudo comparativo realizado por [52]. Neste comparativo, o Spread apresentou melhor desempenho que os demais sistemas avaliados, JavaGroups e Ensemble. Isso deve-se principalmente ao modelo de *daemons* utilizado pelo Spread e a linguagem na qual este sistema foi implementado. Assim com o Sedna, o Spread tem código aberto, o que facilita seu uso principalmente no ambiente acadêmico.

Inicialmente, a implementação do RepliX acompanhou os inúmeros *upgrades* do Sedna. Contudo, a cada *upgrade* alterações significativas eram necessárias no RepliX, ocasionando sensível atraso na finalização da implementação. Portanto, optou-se por

estacionar no *upgrade* da versão 1.0.150 do Sedna, que se mostrou estável. Posteriormente, poderá ser realizado o *upgrade* para uma versão mais recente. O Sedna 1.0.150, assim como suas outras versões, possui um ambiente complexo para desenvolvimento e adição de extensões. A documentação da implementação está dispersa no código, o que representou um desafio adicional no desenvolvimento do RepliX. A versão atual do Spread é a 4.0 e esta foi usada na implementação do RepliX.

Para facilitar o *upgrade*, procurou-se fazer o mínimo de modificações no Sedna, visando não comprometer o desempenho do banco. O RepliX integra classes do Sedna, do Spread e classes adicionais.

RepliXDriver

RepliXDriver é uma reimplementação do *driver* Java do banco Sedna, que possui métodos similares a API JDBC [53], adaptada ao modelo XML. A Figura 5.1 mostra o diagrama de classes do RepliXDriver. A classe `RepliX` é utilizada para obtenção de conexões e execução de transações. Essa classe comunica-se com o RepliXCoordinator, através da interface remota `RemoteRepliX` usando RMI. A classe `RepliX` também pode funcionar para um mecanismo mais simples de replicação, com a ausência do RepliXCoordinator, no qual os clientes conectam-se diretamente a um dos sites. Nesse caso, classe `RepliX` comunica-se diretamente com o RepliXNode. Essa abstração é possível, pois tanto as classes remotas do RepliXCoordinator quanto as do RepliXNode implementam as mesmas interfaces remotas.

As interfaces `RemoteRepliX`, `RemoteConnection` e `RemoteTransaction` estendem a interface `Remote` do pacote para invocação remota de métodos (RMI), java.rmi. Todo objeto remoto deve implementar direta ou indiretamente essa interface. A interface `RemoteRepliX` define os métodos utilizados como pontos de entrada para utilização do RepliX.

O método `connect` representa a maneira tradicional como desenvolvedores de aplicação trabalham com BDs. Além dos tradicionais parâmetros como *host*, porta, nome do banco de dados, usuário e senha, esse método também recebe um parâmetro *booleano* indicando se será necessária permissão para escrita, ou seja, permissão para executar transações de atualização. Esse parâmetro é utilizado pelo RepliXCoordinator para direcionar as transações para o grupo de atualização ou leitura. O método retorna um objeto remoto que implementa a interface `RemoteConnection`. Essa interface representa

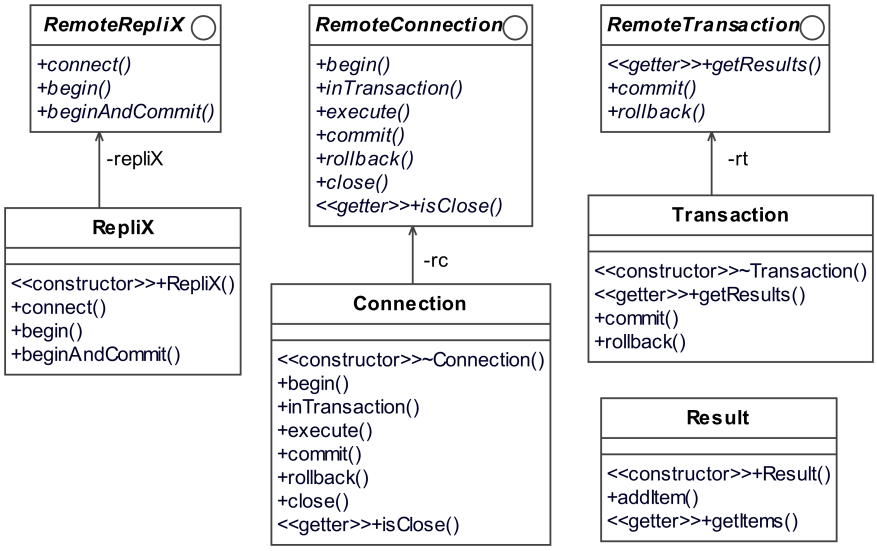

Figura 5.1 Diagrama de classe do RepliXDriver

uma conexão com o BD em um dos sites.

O método `begin` de `RemoteConnection` inicia uma transação e o `inTransacion` retorna um `booleano` indicando se a conexão está em uma transação ou não, ou seja, se uma transação foi iniciada, mas ainda não terminou. O método `execute` executa operações dentro da transação iniciada no BDXN e retorna os resultados representados por um objeto da classe `Result`. Os itens contidos no resultado podem ser obtidos pelo método `getItems` dessa classe. Os métodos `commit` e `rollback` de `RemoteConnection`, respectivamente, efetivam e abortam a transação corrente. O método `close` fecha a conexão. O estado da conexão pode ser inspecionado através do método `isClose`. Uma vez fechada a conexão, a execução de quaisquer operações, exceto de `isClose`, causará uma exceção.

O principal método disponibilizado pela interface `RemoteRepliX` é o método `begin`. Esse método apresenta uma nova interface para execução de transações em ambientes replicados aos desenvolvedores de aplicações. O conceito de conexão, na perspectiva do cliente, deixa de existir e é utilizada apenas a abstração de transações. O método `begin` recebe, além dos tradicionais parâmetros, as operações a serem executadas e inicia a transação em um dos sites, retornando um objeto remoto que implementa a interface `RemoteTransaction`. O site onde será iniciada a transação fica a cargo da implementação do método `begin` na classe que implementa a interface `RemoteRepliX`. Na implementação

do RepliXCoordinator, por exemplo, além de ser feito um balanceamento de carga, as operações são analisadas e a transação é direcionada a um site do grupo de atualização ou de leitura dependendo do seu conteúdo. Os resultados da execução da transação podem ser inspecionados através do método `getResults` de `RemoteTransaction`. Esse método retorna uma coleção de objetos da classe `Result` representando o resultado de cada operação executada. O método `begin` de `RemoteRepliX` inicia a transação e executa suas operações, mas não as efetiva. A transação deve ser efetivada através do método `commit` de `RemoteTransaction` ou abortada através do método `rollback`.

O terceiro método de `RemoteRepliX`, `beginAndCommit`, é similar ao método `begin` e representa uma maneira rápida de iniciar uma transação em um site, executar suas operações e efetivá-la no BD através de uma única chamada remota. Seus parâmetros são idênticos aos do método `begin`, mas, em vez de retornar um objeto remoto representando a transação, retorna apenas uma coleção de objetos `Result` com os resultados das operações executadas. As classes e interfaces que acabamos de descrever estão incluídas em uma biblioteca do RepliX compartilhada por todos os módulos, portanto estão presentes nos diagramas de classes que comentaremos a seguir.

As únicas classes definidas e utilizadas exclusivamente pelo RepliXDriver são `RepliX`, `Connection` e `Transaction`. Essas classes apenas encapsulam as interfaces remotas `RemoteRepliX`, `RemoteConnection` e `RemoteTransaction`, abstraindo do cliente a maneira como é feita a comunicação entre os módulos do RepliX. Essa abstração permite que o mecanismo de comunicação, que atualmente é o RMI, possa, futuramente, ser modificado sem que as aplicações clientes precisem ter seu código alterado. Além de encapsular as chamadas aos respectivos objetos remotos, as classes do RepliXDriver também tratam as exceções referentes a falhas no RMI, levantando exceções próprias da classe `DriverException` (não está presente no diagrama).

RepliXCoordinator

O RepliXCoordinator é o componente que gerencia a execução das transações em cada um dos grupos. A Figura 5.2 mostra o diagrama de classes desse componente. A classe `RemoteRepliXImpl` implementa a interface remota `RemoteRepliX` e é acessada via RMI pelo RepliXDriver. Essa classe é responsável por analisar as operações de cada transação, identificá-la e direcioná-la para um dos sites. A classe comunica-se com o componente RepliXNode, que executa em cada uma das réplicas, usando RMI para chamar métodos

da classe remota `RemoteRepliXImpl` do RepliXNode, que também implementa a interface remota `RemoteRepliX`.

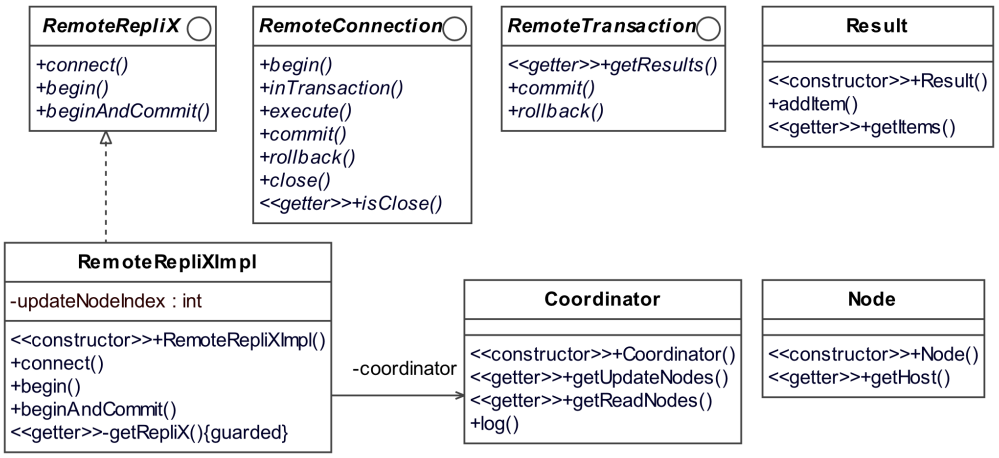

Figura 5.2 Diagrama de classe do RepliXCoordinator

A classe `Coordinator` centraliza a manutenção do RepliXCoordinator. Essa classe inicializa o RepliXCoordinator carregando as configurações do RepliX de um arquivo de configuração no formato XML. A classe também mantém a lista de sites de cada grupo, que podem ser consultadas através dos métodos `getUpdateNodes` e `getReadNodes`. Além disso, seu método `log` é utilizado para geração de um *log* de eventos relevantes do RepliXCoordinator.

A classe `Node` é utilizada para representar os sites e traz apenas um método público para obtenção do endereço do *host*. A classe `RemoteRepliXImpl` é uma implementação concreta da interface remota `RemoteRepliX`. O RepliXCoordinator mantém apenas uma instância das classes `Coordinator` e `RemoteRepliXImpl`, sendo que esta última atua como uma ponte entre os clientes e os BDXNs nos sites, analisando as conexões requeridas e as transações executadas e direcionando-as aos diversos sites. O principal método dessa classe é o `getRepliX`, responsável por escolher um dos sites do grupo de atualização ou de leitura para receber as chamadas remotas. Esse método recebe um parâmetro indicando se a transação é de atualização ou de leitura. As variáveis `updateNodeIndex` e `readNodeIndex` são utilizadas para indicar o próximo site a ser escolhido pelo algoritmo. Com o site escolhido, o RMI é utilizado para instanciar um objeto remoto que implementa `RemoteRepliX` para onde as chamadas a `connect`, `begin` e `beginAndCommit` serão redirecionadas.

A implementação do método `connect` utiliza o `getRepliX` para obter a referência ao `RemoteRepliX` do site escolhido e redireciona a chamada ao método correspondente nesse objeto remoto. Essa chamada retorna um objeto `RemoteConnection`, referente a conexão iniciada no site, que é retornado do RepliXCoordinator ao RepliXDriver, o qual invocou a chamada inicial.

A implementação dos métodos `begin` e `beginAndCommit` funcionam de maneira similar. Ambas inspecionam as operações da transação para identificar seu tipo e, em seguida, utilizam o `getRepliX` para obter a instância remota de `RemoteRepliX` do site escolhido. Essa inspeção é feita verificando a existência de palavras-chave usadas na sintaxe das operações de atualização (essa sintaxe pode ser observada no Apêndice A). Então, é feita uma chamada ao método correspondente de `RemoteRepliX` no site remoto que, no caso do método `begin`, retorna um objeto remoto `RemoteTransaction` representando a transação iniciada, e, no caso do `beginAndCommit`, retorna uma coleção de objetos `Result` com os resultados das operações executadas.

RepliXNode

O RepliXNode realiza a interação entre os demais componentes do RepliX e cada base de dados. Para facilitar a extensão do RepliX, foram criadas abstrações para o acesso ao banco e, para o envio e recebimento de mensagens através das primitivas de CG, a classe abstrata `DatabaseConnection` e a interface `GroupMessenger`, respectivamente. No RepliX foram criadas as classes concretas `SednaConnection`, que acessa o driver Java do Sedna, e `SpreadGroupMessenger`, que encapsula e simplifica a utilização das classes do sistema Spread. Essas abstrações podem ser visualizadas na Figura 5.3.

A classe central do RepliXNode é a classe abstrata `Node` e suas implementações concretas `ReadNode` e `UpdateNode`, correspondentes, respectivamente, aos sites de leitura e de atualização. O método estático `getInstance` processa o arquivo de configuração do site, inicializando algumas variáveis, como o máximo de conexões mantidas pelo *pool*, criando essas conexões e instanciando a subclasse de `Node` de acordo com o tipo do site, atualização ou leitura.

O *pool* de conexões é mantido para diminuir o *overhead* de ter de conectar ao BDXN repetidamente. Os métodos `getConnection` e `releaseConnection` são disponibilizados para obtenção de uma conexão e para liberação de uma conexão em uso, respectivamente. No caso do `getConnection`, se não há uma conexão disponível, o método

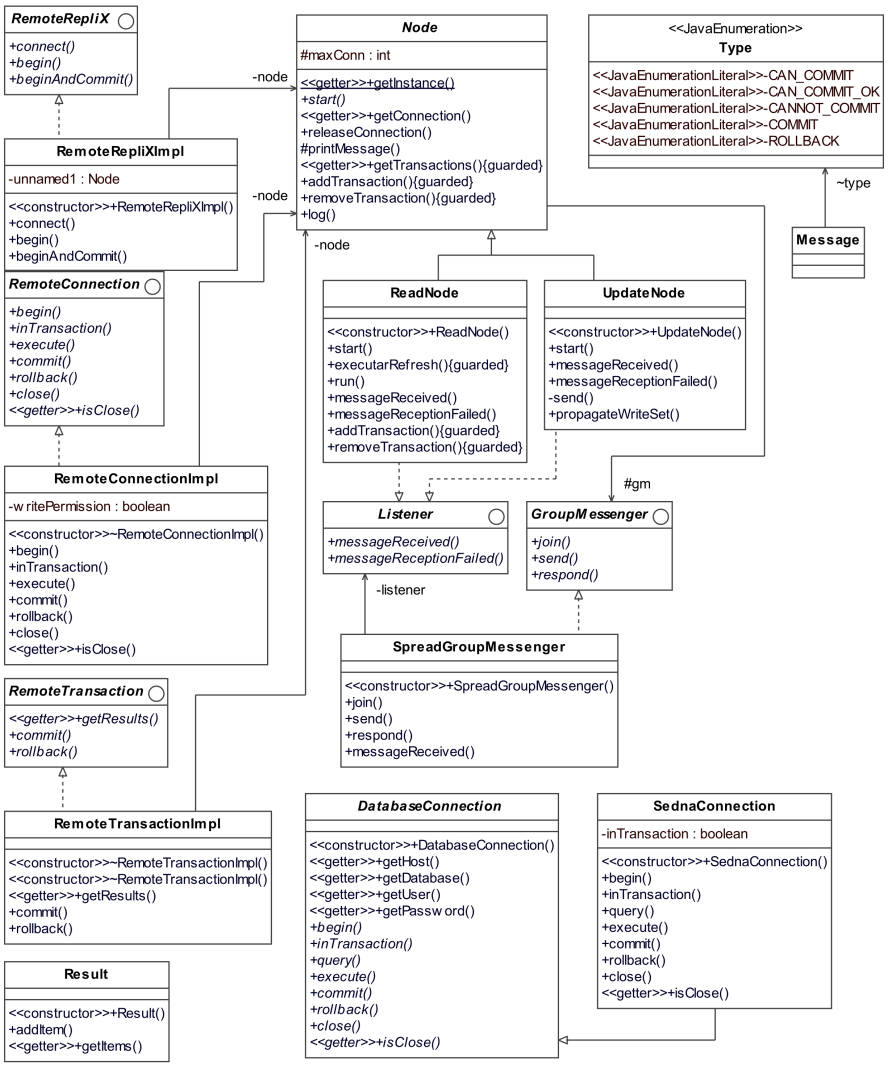

Figura 5.3 Diagrama de classe do RepliXNode

bloqueia até que uma conexão seja liberada.

Para a conexão com o BDXN foi criada uma classe abstrata `DatabaseConnection` que define uma interface para operações relacionadas. Essa abstração permite a utilização de qualquer BDXN, bastando implementar uma subclasse concreta que realize as operações abstratas de `DatabaseConnection`, encapsulando o driver do BDXN em questão. Nesta implementação foi criada a classe `SednaConnection` que escapsula os detalhes da utilização do driver do Sedna e adapta suas operações e exceções ao padrão criado para o RepliX. Os métodos `getXXX` retornam informações básicas da conexão. O método `begin` inicia uma transação na conexão com o BDXN. O método `inTransaction` verifica se há uma transação aberta na conexão. Os métodos `query` e `execute` executam

operações dentro da transação iniciada, com a diferença de que o método `query` retorna os resultados da operação, enquanto o `execute` nada retorna. Os métodos `commit` e `rollback`, respectivamente, efetivam e abortam a transação iniciada. O método `close` fecha a conexão e o método `isClose` verifica se a conexão está fechada ou não.

O método `start` de `Node` e suas subclasses inicia a operação do site, instanciando `RemoteRepliXImpl` e inicializando o sistema de comunicação em grupo. No caso do `UpdateNode`, o site entra no grupo de atualização. Se o site é do tipo `ReadNode`, ele entra no grupo de leitura e, em seguida, inicia a *thread* de propagação, responsável por executar as transações de propagação, nos momentos em que o site se encontra ocioso, e as transações de *refresh*.

Assim como para as conexões com os BDXNs, também foi criada uma abstração para o sistema de comunicação em grupo, possibilitando, futuramente, a utilização de um outro sistema sem mudanças no código do RepliXNode. A interface `GroupMessenger` define os métodos básicos para comunicação em grupo. O método `join` permite que o site entre em um determinado grupo. O método `send` envia mensagens para um ou mais grupos. O método `respond` é utilizado para responder mensagens recebidas diretamente ao remetente. A interface `Listener` de `GroupMessenger` define os eventos de recepção de mensagens. Uma classe interessada em receber mensagens do sistema de comunicação deve implementar essa interface. O método `messageReceived` é chamado quando uma mensagem é recebida, e o método `messageReceptionFailed` quando ocorreu algum problema durante a recepção.

Para esta implementação foi criada a classe `SpreadGroupMessenger` que encapsula e simplifica o uso das classes do sistema Spread, implementando as operações definidas na interface `GroupMessenger`. A classe `Node` também possui o método `log` para geração de um registro dos eventos mais importantes do `RepliXNode` e o método `printMessage` para depuração das mensagens recebidas. Os métodos `getTransactions`, `addTransaction` e `removeTransaction` são utilizados para a manutenção da lista de transações ativas no site e têm particular importância para a subclasse `ReadNode`.

A classe `RemoteRepliXImpl` é a implementação do `RepliXNode` da interface remota `RemoteRepliX`. Essa classe é o ponto de entrada para chamadas ao `RepliXNode`. O método `connect` utiliza o método `getConnection` do `Node` para obter uma conexão do *pool* e encapsula a instância de `DatabaseConnection` obtida em um objeto remoto `RemoteConnectionImpl`, que terá sua referência retornada ao cliente que o invocou. Os

métodos `begin` e `beginAndCommit`, por sua vez, encapsulam a conexão obtida em um objeto remoto `RemoteTransactionImpl` que, no caso do método `begin`, também terá sua referência retornada ao cliente que o invocou. No caso do método `beginAndCommit`, é retornado uma coleção de objetos `Result`, obtida através do método `getResults` de `RemoteTransactionImpl`, contendo os resultados das operações executadas.

A classe `RemoteConnectionImpl` implementa a interface `RemoteConnection` e encapsula a conexão, representada por uma instância de `DatabaseConnection`. O método `begin` chama o método `addTransaction` de `Node` para indicar o início de uma transação no site e o método `begin` de `DatabaseConnection` para iniciar a transação no BDXN local. O método `execute` analisa se a conexão tem direito de escrita e executa as operações requisitadas através do método `query` de `DatabaseConnection` retornando os resultados em um objeto `Result`. Se a transação for de atualização, as operações de escrita são acumuladas na lista `writeSet` para posterior propagação aos outros sites do grupo. O método `commit` verifica o *write-set* acumulado, chamando o método `propagateWriteSet` de `Node`, caso existam operações de escrita, para a propagação das alterações aos outros sites ativos do grupo de atualização. Em seguida, efetiva a transação no BDXN local através do método `commit` de `DatabaseConnection` e chama o método `removeTransaction` de `Node` para avisá-lo da finalização da transação. O método `rollback` aborta a transação iniciada usando o método `rollback` de `DatabaseConnection` e também chama o método `removeTransaction` de `Node` para avisá-lo da finalização da transação. O método `close`, em vez de fechar a conexão com o BDXN, apenas devolve a conexão ao *pool* através do método `releaseConnection` de `Node` e bloqueia quaisquer operações posteriores em cima do objeto.

A classe `RemoteTransactionImpl` é uma implementação da interface remota `RemoteTransaction` e encapsula uma transação em uma conexão no BDXN local. Durante a construção do objeto, a transação é iniciada e suas operações são executadas, respectivamente através dos métodos `begin` e `query` de `DatabaseConnection`. Se a transação for de atualização as operações de escrita são acumuladas na lista `writeSet`. Os resultados da execução das operações são guardadas em uma coleção de objetos `Result` que podem ser consultados através do método `getResults`. Os métodos `commit` e `rollback` funcionam de modo análogo à implementação de `RemoteConnectionImpl`, diferenciando-se apenas por liberar a conexão utilizada para execução da transação logo que esta finaliza, através do método `releaseConnection` de `Node`.

O `UpdateNode` é a subclasse de `Node` instanciada nos sites de atualização. Seus

métodos principais são `propagateWriteSet` e `messageReceived`, o evento acionado pelo `GroupMessenger` ao receber uma mensagem do sistema de comunicação em grupo. O método `propagateWriteSet` cria uma mensagem do tipo `write-set` e a preenche com os dados da transação que o invocou. Em seguida, envia a mensagem ao grupo de atualização através do método `send`, que encapsula e centraliza as chamadas ao método `send` do `GroupMessenger`, e aguarda a confirmação dos *testes de certificação* dos sites secundários. O método `messageReceived` é responsável pela recepção de mensagens advindas do sistema de comunicação em grupo. Quando os sites *secundários* recebem uma mensagem do tipo `write-set`, realizam o *teste de certificação* e retornam uma resposta ao site *primário* através do método `respond` do `GroupMessenger`. Quando o site *primário* recebe as confirmações dos sites *secundários*, uma mensagem do tipo `COMMIT` é enviada através do método `send` ao grupo de atualização e uma outra mensagem é enviada ao grupo de leitura com as atualizações da transação, então, o método `propagateWriteSet` retorna. Quando a mensagem do tipo `COMMIT` é recebida nos sites *secundários*, uma conexão do *pool* é obtida através do método `getConnection` e a transação é iniciada, suas operações executadas e efetivadas pelos métodos `begin`, `execute` e `commit`, respectivamente, de `DatabaseConnection`.

O `ReadNode` é a subclasse de `Node` instanciada nos sites de leitura. A classe `ReadNode` mantém uma fila onde são armazenadas as transações recebidas do grupo de atualização. O método `messageReceived` recebe as mensagens do sistema de comunicação em grupo, no caso do site de leitura, provinientes de um dos sites de atualização. Ao receber uma mensagem contendo um *write-set*, este é posto na fila para posterior efetivação no BDXN local. Os métodos `addTransaction` e `removeTransaction` são sobrescritos na subclasse `ReadNode` para detectar a necessidade da execução de transações de *refresh*, bem como implementar o bloqueio dos itens de dados durante a execução de transações de propagação e *refresh*, impedindo que novas transações iniciem antes que aquelas finalizem e acabem por acessar dados desatualizados.

Antes do início da transação no BDXN local, o método `addTransaction` é chamado pelas classes `RemoteConnectionImpl` e `RemoteTransactionImpl`. Se existirem alterações pendentes na fila, os dados a serem atualizados no site são bloqueados para execução de novas transações de clientes e o método `executarRefresh` é executado. O método `executarRefresh` obtém uma conexão do *pool*, usando o método `getConnection`, esvazia a fila executando todas as alterações pendentes em uma única transação e libera a conexão obtida. Ao fim desse processo, os itens de dados são desbloqueados, liberando-os para

as transações bloqueadas em espera para iniciar. A classe `ReadNode` também mantém uma *thread* de propagação que é notificada pelo método `removeTransaction` quando a última transação em execução é finalizada. Ao ser notificada, verifica a fila e, se houver alterações pendentes, executa uma transação de propagação de maneira análoga ao método `executarRefresh`.

Para facilitar a compreensão das tarefas realizadas pelo RepliX, a Figura 5.4 apresenta um diagrama de seqüência com os componentes do RepliX e a interação entre eles.

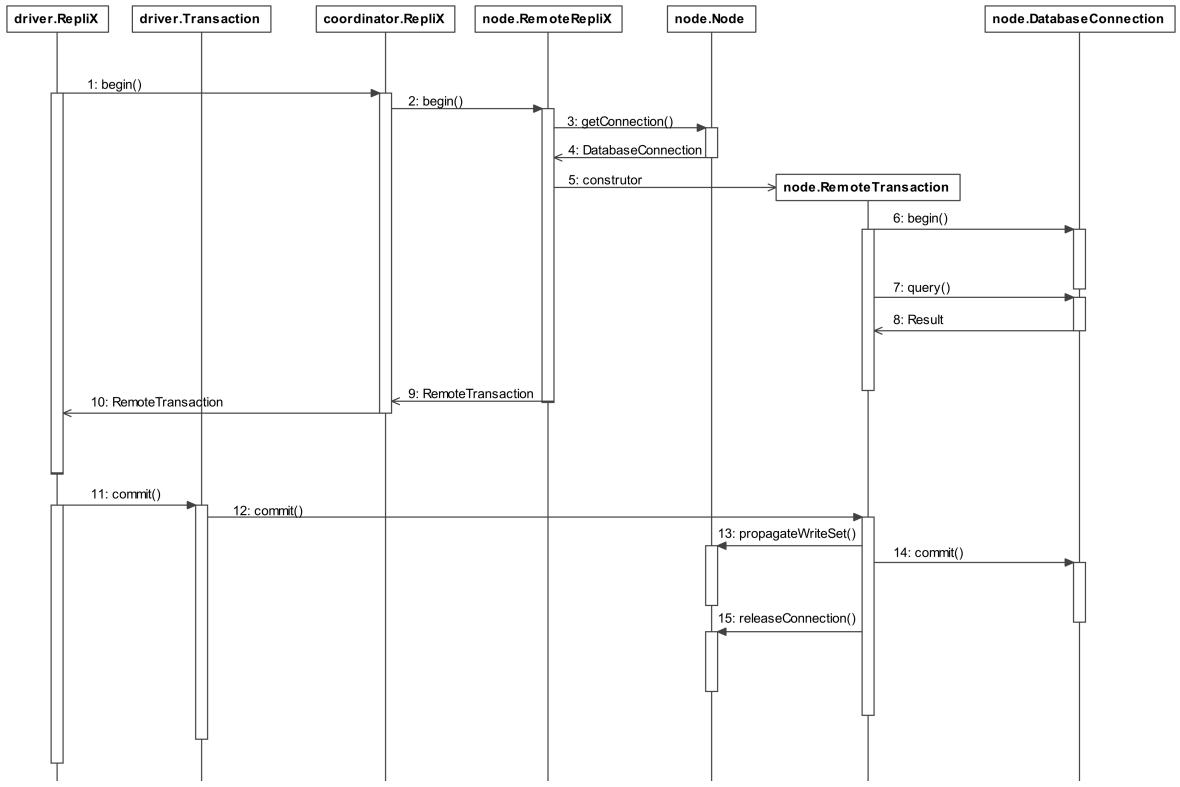

Figura 5.4 Diagrama de sequência do RepliX

O funcionamento do RepliX no diagrama acima pode ser exemplificado seguindo a seqüência de ações e eventos que ocorrem durante a execução de uma transação de atualização. O processo se inicia quando a aplicação cliente invoca o método `begin` na classe `RepliX` do RepliXDriver. Esse método se encarrega de encontrar o RepliXCoordinator e instanciar sua classe remota `RemoteRepliX`. Essa classe é responsável por inspecionar as operações da transação de modo a direcioná-la para o grupo apropriado.

Na execução do método, o coordenador instancia a classe remota `RemoteRepliX`

do RepliXNode e invoca seu método `begin`. É nesse método que o pedido de início de transação feito pelo usuário será realmente atendido no banco de dados local. Um pedido de conexão é feito ao *pool* de conexões da classe `Node`, e a conexão obtida, uma implementação da classe abstrata `DatabaseConnection` correspondente ao BDXN utilizado, é passada à classe `RemoteTransaction` instanciada logo depois. Essa classe remota encapsula a transação que é iniciada no banco de dados local utilizando o método `begin` de `DatabaseConnection` e suas operações são executadas com o método `query`.

Em seguida, a referência para a instância remota de `RemoteTransaction` faz o caminho inverso da chamada `begin` que desencadeou o processo, sendo retornada do `RepliXNode` para o RepliXCoordinator e, finalmente, para o RepliXDriver, onde é encapsulada na classe `Transaction`. Com esse objeto, a aplicação cliente pode inspecionar os resultados das operações, bem como realizar o *commit* ou *abort*.

Quando o cliente executa o método `commit` de `Transaction`, é invocado o método `commit` da instância remota `RemoteTransaction` no site onde a transação foi iniciada. Ele, então, chama o método `propagateWriteSet` de `Node` que é responsável por sincronizar os sites do grupo de atualização e propagar as alterações aos sites do grupo de leitura, utilizando o sistema de CG. Finalmente, é feito o *commit* no banco de dados local utilizando o método `commit` de `DatabaseConnection`, e a conexão é liberada utilizando o `releaseConnection` de `Node`.

5.2 AVALIAÇÃO

À medida que os sistemas computacionais se tornam mais complexos, a análise de desempenho torna-se uma atividade cada vez mais relevante e indispensável. No âmbito desses sistemas, uma ferramenta para execução de testes padrão (*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 respectivos consumos de recursos, submetido a um determinado sistema para execução durante um determinado intervalo de tempo. Por sua vez, uma tarefa é a unidade de execução do sistema computacional. Por exemplo, num cenário de banco de dados, uma tarefa pode corresponder a uma consulta XQuery.

A avaliação no contexto deste trabalho busca analisar o desempenho e a disponi-

bilidade proporcionada pelo RepliX. Devido às interfaces e às linguagens de acesso aos BDXNs, torna-se complexo desenvolver experimentos apropriados para verificar o desempenho desses sistemas [54]. Nesse sentido, vários *benchmarks* para dados XML foram propostos, tais como os apresentados em [55][56]. Lu et al.[57] reportam que nenhum estudo foi encontrado no uso de *benchmarks* que permitam ao usuário identificar o impacto causado pelo tipo de armazenamento no desempenho das consultas e argumentam que a observação da forma como os dados são armazenados, ou seja, com e sem a utilização de um esquema, influenciam muito na avaliação do sistema.

Com a ausência de operações de atualização na linguagem XQuery, os *benchmarks*, em geral, possuem suporte apenas a operações de leitura. Recentemente o W3C publicou um *draft* [58] com uma extensão da XQuery contendo algumas operações de atualização, tais como inserção, deleção e renomeação. O banco Sedna implementa essas operações e possui algumas extensões, baseadas na proposta de Patrick Lehti [59], mostradas no Apêndice A.

Para a avaliação do RepliX, estendemos o *benchmark* XMark [55] adicionando operações de atualização (Apêndice B), de acordo com observações feitas por [60], de forma a viabilizar a execução de experimentos. Também desenvolvemos um modelo de avaliação e um simulador de clientes baseados em [61].

Modelo e Ambiente de Avaliação

Assumimos um conjunto de sites S=$\{S_1...S_N\}$. Cada site S_i possui um banco de dados e este contém uma cópia completa da base de dados, realizando o gerenciamento das transações localmente. O banco assegura as propriedades ACID na execução das transações locais e utiliza o protocolo 2PL para garantir o controle de concorrência. Uma rede provê a comunicação entre os componentes do RepliX. O sistema de CG provê as primitivas de comunicação utilizadas pelo RepliX. Consideramos um conjunto de clientes C=$\{C_1...C_M\}$, que são a origem das transações. Para processar um transação t, um cliente C conecta-se ao RepliX e submete a transação t ao RepliX, que repassa t ao site S_i. Para cada transação t, somente um site S_i a recebe. No grupo de atualização, esse site é o *primário*. A concorrência de transações é simulada usando múltiplos clientes. O simulador de clientes gera as transações de acordo com certos parâmetros, envia para o RepliX e coleta os resultados ao final da cada execução. A Figura 5.5 exibe a interface desse simulador.

Figura 5.5 Simulador de clientes

O ambiente utilizado para a avaliação foi um *cluster* de 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 Tabela 5.1 exibe os valores dos parâmetros utilizados na avaliação.

Parâmetro	**Valor**
Número de sites	11
Tamanho da base de dados	10 MB
Número de clientes	20 - 50
Tamanho das transações	10 operações
Operações de leitura	80%

Tabela 5.1 Paramêtros utilizados na avaliação

5.3 EXPERIMENTOS

Os experimentos foram realizados de acordo com o modelo de avaliação descrito. Cada experimento explora aspectos diferentes de um banco de dados replicado, tais como tempo de resposta, *throughput* ou vazão, escalabilidade, proporção de atualizações e disponibilidade. A comparação foi feita entre o banco Sedna convencional e o RepliX acoplado ao Sedna.

Para evitar que atrasos na inicialização do RepliX viessem a interferir nos resultados, as medidas iniciais obtidas das transações executadas (10%) foram descartadas,

considerando os valores posteriores, o que torna os experimentos mais próximos de um ambiente real.

Desempenho

Para medir o desempenho do RepliX, foram considerados dois índices: o tempo de resposta e o *throughput*. O tempo de resposta foi medido considerando diferentes quantidades de clientes, onde cada cliente submete 100 transações, sendo 80% das transações de leitura. No RepliX foram utilizados 11 sites, sendo que o número de sites do grupo de atualização foi composto por 3 sites.

A Figura 5.6 mostra o tempo médio de resposta. No gráfico, o tempo de resposta aumenta com a adição de mais clientes e o Sedna convencional apresenta um resultado bem inferior ao RepliX. A razão para isso é que o Sedna é sobrecarregado muito rápido, enquanto no RepliX as transações de atualização e leitura são distribuídas entre as réplicas. A troca eficiente de mensagens executada pelo sistema de comunicação em grupo também contribuiu para diminuir o tempo de resposta do RepliX.

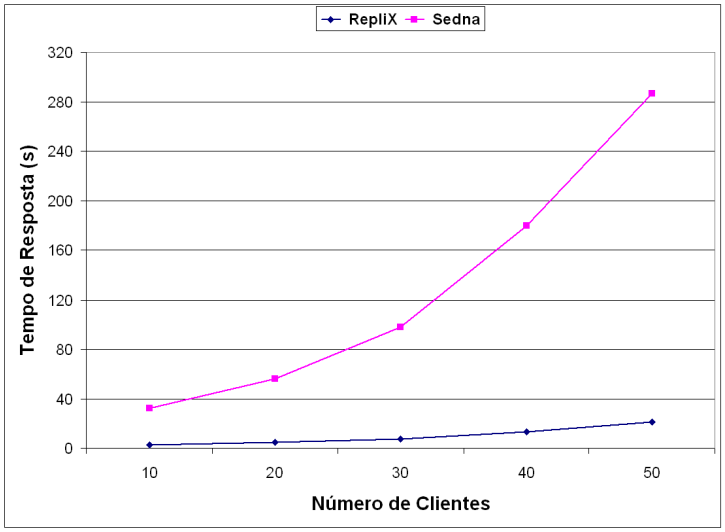

Figura 5.6 Gráfico de tempo de resposta

O *throughput*, ou vazão, que representa a capacidade máxima de processamento por unidade de tempo, foi medido variando a quantidade de clientes. A Figura 5.7 mostra o *throughput* para o RepliX e o Sedna convencional. No gráfico, pode-se observar que as duas curvas diminuem com o aumento no número de clientes. O Sedna convencional apresenta baixos valores de *throughput* e os mantém constantes. Isso ocorre porque o Sedna,

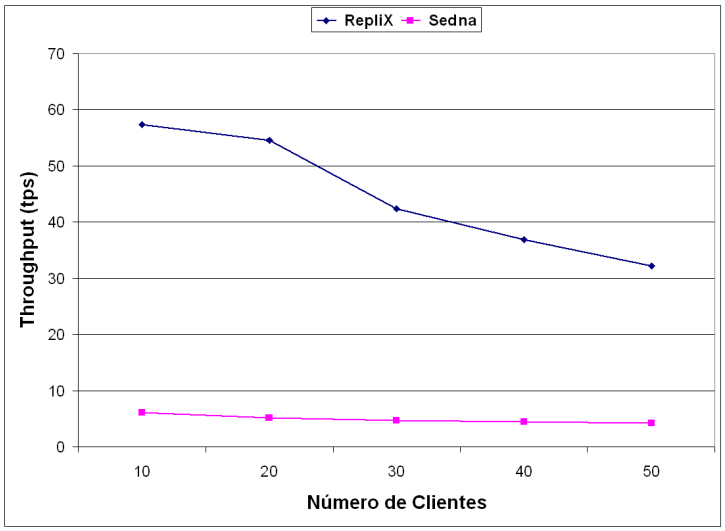

Figura 5.7 Gráfico de *throughput*

assim como sistemas centralizados, apresentam um limite no processamento e gerenciamento de transações. Com o RepliX, a taxa de *throughput* melhorou significativamente, devido à execução concorrente das transações nos sites. Com o aumento para 30 clientes, o *throughput* do RepliX diminue consideravelmente, já que todas as réplicas começam a ficar sobrecarregadas. Já com 40 e 50 clientes, o *throughput* diminui mais lentamente. No geral, o RepliX apresentou melhores resultados de *throughput* do que o Sedna com os números de clientes avaliados.

Proporção de Atualizações

Em sistemas replicados, a proporção de transações de atualizações é um parâmetro importante, podendo afetar o desempenho. Para verificar isso com o RepliX, fixamos o número de sites em 11, sendo 3 destes de atualização, 20 clientes submetendo 100 transações cada um e alteramos a proporção de atualizações, calculando o tempo de resposta. A Figura 5.8 apresenta um gráfico com os resultados obtidos neste teste.

O tempo de resposta do Sedna convencional cresce rapidamente com a adição de transações de atualização. Já no RepliX, o tempo de resposta aumenta gradualmente até 40% de atualizações. Uma razão é que as transações são distribuídas entre os grupos de atualização e leitura, tornando o processamento mais eficaz. A partir de 40% de atualizações, o tempo de resposta aumenta bastante. Isso se deve ao aumento no envio de mensagens do grupo de atualização para o grupo de leitura. Mesmo assim, os resultados

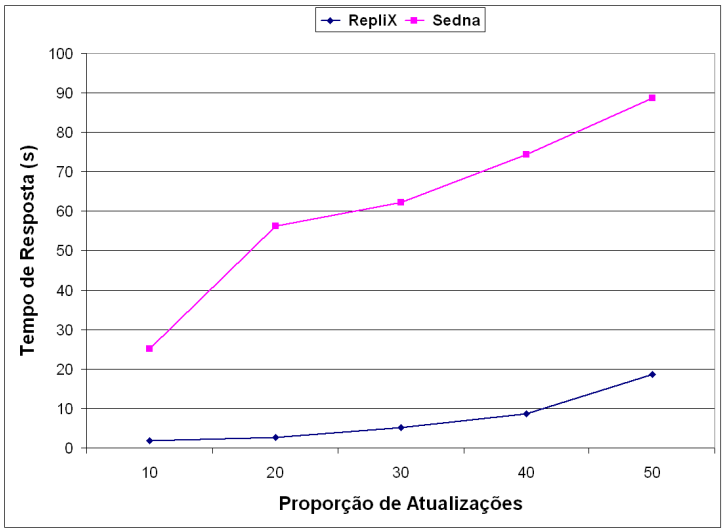

Figura 5.8 Gráfico de proporção de atualizações

do RepliX são satisfatórios.

Escalabilidade

Um sistema replicado deve melhorar seu desempenho quando o número de réplicas é aumentado. Neste experimento, medimos de que modo o RepliX reflete a alteração no número de réplicas. Fixamos o número de clientes em 20, cada um submetendo 100 transações, sendo 80% de leitura. A Figura 5.9 mostra o resultado da avaliação de escalabilidade.

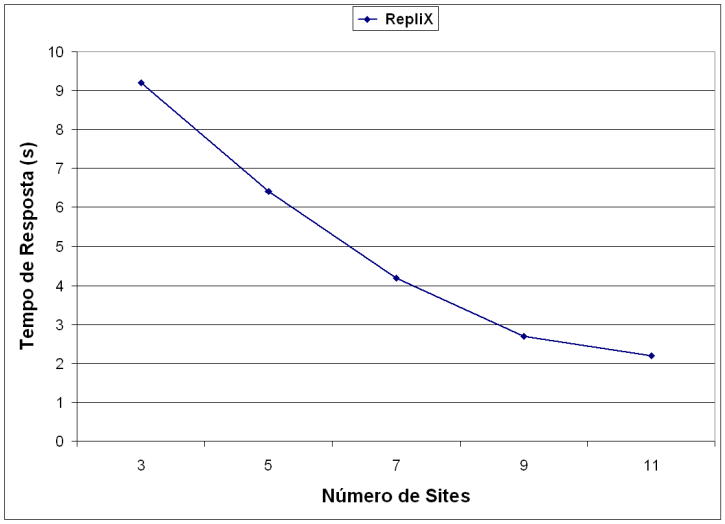

Figura 5.9 Gráfico de escalabilidade

A adição de mais sites melhorou o desempenho do RepliX. Com 3 sites, o RepliX apresentou um tempo de resposta considerável. Já com 7 sites, o resultado melhorou bastante. A partir de 10 sites, o tempo de resposta do RepliX tende a manter-se constante. A estratégia de sincronização do RepliX, atualizando inicialmente os sites do grupo de atualização e difundindo as modificações, diminui o problema de sincronizar todas as réplicas a cada atualização, o que favorece a escalabilidade.

Disponibilidade

Para verificar a disponibilidade do RepliX, foram realizados dois experimentos nos quais foram provocadas sucessivas falhas em uma instância no grupo de replicação, enquanto eram realizadas requisições ao RepliX de forma contínua. Fixamos o número de clientes em 20, cada um submetendo 100 transações, sendo 80% de leitura.

No primeiro experimento, configuramos para que um site funcionasse corretamente por 5 minutos e, decorrido esse tempo, interrompeu-se a execução por um minuto. Este experimento foi realizado por um período de 3 horas para cada quantidade de sites, calculando-se o tempo de resposta. A Figura 5.10 mostra o comportamento do RepliX. Por exemplo, com 3 sites, houve somente 2,4% de aumento no tempo de resposta das requisições. A adição de mais sites melhorou a disponibilidade, já que um número menor de requisições sofre falha e precisa ser direcionado para outro site.

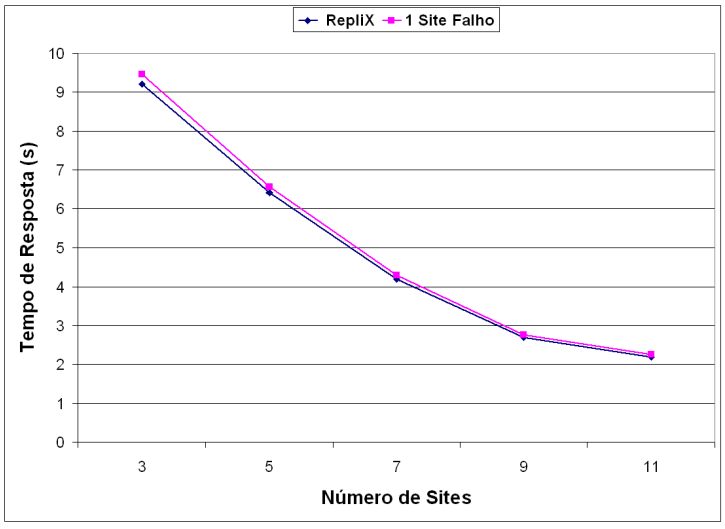

Figura 5.10 Gráfico de disponibilidade

No segundo experimento, mostrado na Figura 5.11, adicionamos uma falha con-

stante em um dos sites, ou seja, este site não responde a requisições e não se recupera, permanecendo com falha durante todo o experimento. O tempo de resposta diminui em comparação com a execução do RepliX sem a presença de falhas, mas manteve-se em valores aceitáveis. No pior caso, na configuração com falha e 3 sites, o RepliX é apenas 6,2% mais lento que na configuração sem falha.

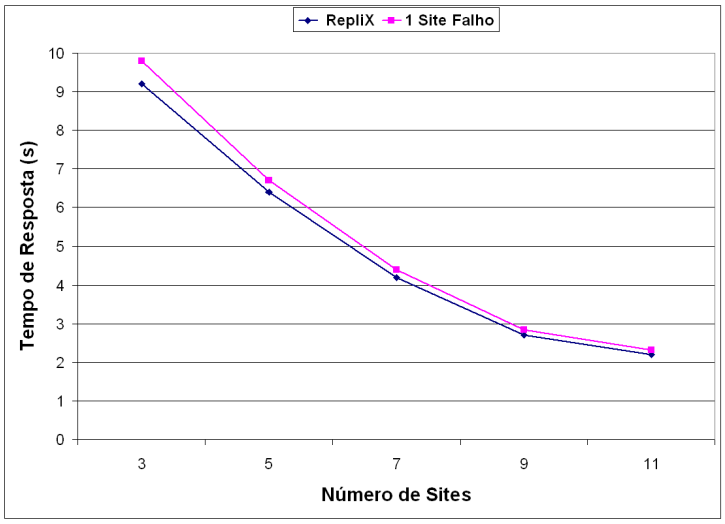

Figura 5.11 Gráfico de disponibilidade com falha constante

Nos dois experimentos, nenhuma das requisições dos clientes deixou de ser respondida pelo RepliX mesmo com falhas sendo provocadas em um dos sites. Entretanto, esses experimentos não são suficientes para validar a dependabilidade total da solução apresentada neste trabalho. Para alcançar uma validação mais completa, experimentos detalhados com injeção de falhas são necessários.

5.4 CONCLUSÃO

Este capítulo descreveu a implementação e avaliação do RepliX. Foi justificada a escolha dos sistemas utilizados, os detalhes da implementação foram descritos, além do modelo e do ambiente de avaliação. Ao final, foram apresentados os resultados obtidos na avaliação do RepliX. O capítulo a seguir apresenta as conclusões deste trabalho.

CAPÍTULO 6

CONCLUSÃO

Este trabalho apresentou o RepliX, um mecanismo para replicação em bancos de dados XML Nativos, que objetiva melhorar a disponibilidade e o desempenho desses sistemas. O RepliX possui uma arquitetura modular e flexível, o que facilita sua integração a qualquer BDXN, além de poder ser estendido, adicionando novas características.

6.1 RESULTADOS ALCANÇADOS

O XML é um padrão amplamente utilizado. Por causa de sua flexibilidade e simplicidade, adotá-lo significa uma boa escolha para, por exemplo, representação de dados na Internet e como formato padrão para troca de informações entre diferentes aplicativos, entre outros.

Com um volume muito grande de documentos XML, surge a necessidade de tratá-los também como bases de dados. Os Bancos de Dados XML Nativos têm se apresentado com uma solução eficiente nesse contexto. Todavia, esses bancos ainda apresentam muitas limitações, como a ausência de mecanismos de replicação eficientes. As soluções que suportam uma replicação eficiente das bases de dados são extremamente importantes, uma vez que aumentam a disponibilidade dos dados na presença de falhas, permitindo explorar a localidade desses dados e dispersar a carga entre as várias réplicas.

Atualmente, existem alguns mecanismos para replicação em BDXNs. No entanto, esses mecanismos 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 RepliX, que combina protocolos de cópia primária e réplicas ativas com características de comunicação em grupos, de forma a permitir a replicação eficiente de dados XML. Em particular, o RepliX contempla as características dos dados XML, fornecendo meios para a replicação de dados nesse formato, melhorando a disponibilidade e o desempenho dos BDXNs. De forma a demonstrar que as opções de desenvolvimento do RepliX foram adequadas, discutiu-se os desafios da sua concretização no BDXN Sedna.

Por fim, avaliou-se o RepliX considerando diversas características de replicação. A avaliação consistiu em comparar o BDXN Sedna convencional com o Sedna utilizando o RepliX, enquanto processavam cargas semelhantes num mesmo cenário de execução. Para isso, foi utilizado o *benchmark* XMark, que gera carga normalizada para bases de dados. Pela análise dos resultados obtidos, foi possível verificar que o RepliX melhorou o desempenho e a disponibilidade do Sedna, mesmo em cenários de replicação com grande proporção de atualizações.

6.2 TRABALHOS FUTUROS

Como trabalhos futuros pretendemos, primeiramente, desenvolver soluções de balanceamento de carga mais eficazes, considerando a carga dos sites e o conteúdo das consultas. Com isso, forneceremos um mecanismo ainda mais robusto para a replicação de dados XML.

Em seguida, pretendemos desenvolver uma estratégia para a decomposição de consultas XQuery que permita ao RepliX trabalhar não apenas com replicação total, mas também com replicação parcial. Para tanto, estenderemos a estratégia apresentada no PartiX, observando características da semântica formal da linguagem XQuery.

O RepliX não fornece uma forma de escolha da estratégia de replicação a ser utilizada. Sendo assim, no momento da inicialização do RepliX, o usuário deve informar apenas os sites que serão utilizadas. Portanto, outro direcionamento a trabalhos futuros seria permitir ao usuário especificar informações sobre a replicação, tais como parâmetros de tempo de propagação de atualização ou novas estratégias de replicação.

O RepliX foi validado com o banco Sedna. Entretanto, pode ser utilizado por qualquer banco. Em trabalhos futuros, poder-se-ia avaliá-lo com outros BDXNs, tais

como Timber, Natix e eXist, verificando a melhoria proporcionada em cada um desses sistemas.

Com relação à avaliação de desempenho e disponibilidade, também é proposto a avaliação do RepliX 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.

Por fim, durante a avaliação do RepliX, foram observadas pequenas quantidades de *aborts* no grupo de atualização, principalmente quando o número de transações de atualização aumenta. Portanto, é necessário um estudo aprofundado desses resultados, bem como da estrutura de funcionamento dos algoritmos visando identificar os fatores que possam ter ocasionado esse problema.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] W3C. *Extensible Markup Language*, 2007. `http://www.w3.org/XML/`.

[2] cXML. *Commerce XML Resources*, 2007. `http://www.cxml.org`.

[3] DBLP. *Digital Bibliography*, 2007. `http://dblp.uni-trier.de`.

[4] 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.

[5] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native xml base management system. *VLDB J.*, 11(4):292–314, 2002.

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

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

[8] Tamino. *Tamino XML Server*, 2007. `http://www.softwareag.com/tamino`.

[9] X-Hive. *X-Hive Database*, 2007. `www.x-hive.com`.

[10] eXist. *eXist: Open Source Native XML Database*, 2007. `http://exist.sf.net`.

[11] Kenneth Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Hardcover, 2005.

[12] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.

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

[14] Khuzaima Daudjee and Kenneth Salem. A pure lazy technique for scalable transaction processing in replicated databases. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 802–808, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 565–576. VLDB Endowment, 2005.

[16] Mark Graves. *Designing XML Databases*. Prentice Hall PTR, 2001.

[17] Michael Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[18] Fausto Ayres. *QEEF - Uma Máquina de Execução de Consultas Extensível*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2005.

[19] Michiels Philippe. Xquery optimization. In *Proc. of the VLDB 2003 PhD Workshop. Co-located with the 29th International Conference on Very Large Data Bases (VLDB 2003)*, number 76 in CEUR Workshop Proceedings, Berlin, Germany, 2003. Morgan Kaufmann.

[20] Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. An algebra for xml query. In *FST TCS 2000: Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 11–45, London, UK, 2000. Springer-Verlag.

[21] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for xml. In *Database Programming Languages, 8th International Workshop, DBPL 2001*, volume 2397 of *Lecture Notes in Computer Science*, pages 149–164, Frascati, Italy, 2001. Springer.

[22] W3C. *XQuery 1.0: An XML Query Language*, 2006. `www.w3.org/XML/Query`.

[23] Mary F. Fernandez and Jérôme Siméon. Growing xquery. In *European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 405–430, Darmstadt, Germany, 2003. Springer Berlin / Heidelberg.

[24] 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.

[25] XUpdate. *XML Update Language*, 2007. `http://xmldb-org.sf.net/xupdate/`.

[26] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity analysis in xml update languages. In *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, 2007, Proceedings*, volume 4353 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2007.

[27] Stijn Dekeyser and Jan Hidders. Conflict scheduling of transactions on xml documents. In *ADC '04: Proceedings of the fifteenth conference on Australasian database*, pages 93–101, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[28] Michael Peter Haustein and Theo Härder. 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.

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

[30] Kuen-Fang Jack Jea, Shih-Ying Chen, and Sheng-Hsien Wang. Concurrency control in xml document databases: Xpath locking protocol. In *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*, pages 551–556. IEEE Computer Society, 2002.

[31] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. Xmltm: efficient transaction management for xml documents. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 142–152, McLean, Virginia, USA, 2002. ACM Press.

[32] Philip Bernstein and Eric Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[33] Peter Pleshachkov, Petr Chardin, and Sergey Kuznetsov. A dataguide-based concurrency control protocol for cooperation on xml data. In *9th East-European Conference*

on Advances in Databases and Information Systems (ADBIS), volume 3631 of *Lecture Notes in Computer Science*, pages 268–282, Tallinn, Estonia, 2005. Springer Berlin / Heidelberg.

[34] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 1999.

[35] 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.

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

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

[38] Alexandre Andrade, Gabriela Ruberg, Fernanda Baião, Vanessa Braganholo, and Marta Mattoso. Efficiently processing xml queries over fragmented repositories with partix. In *EDBT Workshops*, volume 4254 of *Lecture Notes on Computer Science*, pages 150–163, Munich, Germany, 2006. Springer.

[39] Elaine Castro. Xml-pm: Um método eficiente para identificação de padrões no processamento de consultas a dados xml. Master's thesis, Universidade Federal do Ceará, Fortaleza, 2006.

[40] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic xpath processing in natix. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 705–716, Washington, DC, USA, 2005. IEEE Computer Society.

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

[42] 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, Erfurt, Germany, 2002. Springer-Verlag.

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

[44] Peter Pleshachkov. *Transaction Management in XML Database Management Systems*. PhD thesis, Institute for System Programming of Russian Academy of Sciences, Rússia, 2006.

[45] Cristiano Lima. Orpis: Um modelo de consistência de conteúdo replicado em servidores web distribuídos. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

[46] Márcia Pasin. *Réplicas para Alta Disponibilidade em Arquiteturas Orientadas a Componentes com Suporte de Comunicação de Grupo*. PhD thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

[47] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[48] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[49] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. Technical report, Distributed Systems and Networks lab, Johns Hopkins University, 2004.

[50] Bela Ban. Javagroups user's guide. Technical report, Department of Computer Science,Cornell University, 1999.

[51] Marta Mattoso, Geraldo Zimbrão, Alexandre A. B. Lima, Fernanda Baião, Vanessa P. Braganholo, Albino A. Aveleda, Bernardo Miranda, Bruno Kinder Almentero, and Marcelo Nunes Costa. Pargres: uma camada de processamento paralelo de consultas sobre o postgresql. In *WSL - Workshop de Software Livre, FISL 6.0*, pages 259–264, 2005.

[52] Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti. Total order communications: A practical analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference*, volume 3463 of *Lecture Notes in Computer Science*, pages 38–54, Budapest, Hungary, 2005. Springer.

[53] Sun Microsystems. *JDBC 4.0 API Specification*, 2006. `http://java.sun.com/jdbc`.

[54] Francesco Pagnamenta. Design and initial implementation of a distributed xml database. Master's thesis, University of Dublin, Irlanda, 2005.

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

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

[57] 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 Trans. Inter. Tech.*, 5(1):154–194, 2005.

[58] W3C. *XQuery Update Facility*, 2006. `http://www.w3.org/TR/xqupdate/`.

[59] Patrick Lehti. Design and implementation of a data manipulation processor for an xml query language. Technical Report KOM-D-149, Technische University Darmstadt, Germany, 2001.

[60] Zeeshan Sardar and Bettina Kemme. Don't be a pessimist: Use snapshot-based concurrency control for xml. Technical report, School of Computer Science, McGill University, Montreal, Canada, 2005.

[61] Matthias Wiesmann and Andre Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.

APÊNDICE A

LINGUAGEM DE ATUALIZAÇÃO - BANCO DE DADOS SEDNA

A linguagem de atualização do Sedna é uma extensão da XQuery. Nesta linguagem, o resultado de cada operação de atualização não deve violar a consistência das entidades XML armazenadas no banco de dados. Caso isso aconteça, um erro será retornado. As principais operações são: *insert*, *delete*, *delete_undeep*, *replace* e *rename*.

INSERT
A operação INSERT insere dados nas posições identificadas pelos termos *into*, *preceding* e *following*

```
UPDATE insert Expr1 (into|preceding|following) Expr2
```

`Expr1` identifica a seqüência ordenada de nós a serem inseridos. Para cada nó no resultado de `Expr2`, o resultado de `Expr1` é inserido na posição identificada por um dos termos, *into*, *preceding* ou *following*. Se *into* for usado, o resultado de `Expr1` é inserido em uma posição aleatória da seqüência de nós filhos para cada nó do resultado de `Expr2`. Se *preceding* for usado, o resultado de `Expr1` é inserido antes de cada nó do resultado de `Expr2`. Caso *following* seja usado, o resultado de `Expr1` é inserido depois de cada nó do resultado de *Expr2*. Se uma das seguintes condições acontecerem, um erro é retornado.

- Existem nós que não são elementos no resultado de `Expr2` ao usar *into*.
- Existem nós temporários na `Expr2` (um nó é considerado temporário se é criado como resultado da avaliação de um construtor da XQuery).

Exemplo:

```
UPDATE
insert <atencao>Pressão muito alta!</atencao>
preceeding doc("hospital")//presssao[sistole >180]
```

DELETE

A operação DELETE remove nós persistentes do banco de dados e contém uma subexpressão que retorna os nós a serem removidos.

```
UPDATE delete Expr
```

`Expr` identifica os nós a serem removidos do banco de dados. Note que os descendentes dos nós também são removidos. Se a seguinte condição acontecer, um erro é retornado:

- Existem nós no resultado de `Expr` que não estão armazenados em memória externa.

Exemplo:

```
UPDATE
delete doc("hospital")//pressao[sistole>180]
```

DELETE_UNDEEP

```
UPDATE delete_undeep Expr
```

A operação DELETE_UNDEEP remove nós identificados por `Expr`, mas, diferentemente da operação DELETE, não remove seus descendentes. Considerando o seguinte exemplo:

```
UPDATE delete_undeep doc("a.xml")//B
```

O documento a.xml antes da atualização:

<A><B><C/><D/></B></A>

O documento depois da atualização:

<A><C/><D/></A>

REPLACE

A operação REPLACE é usada para substituir nós em um documento XML.

```
UPDATE replace var [as type] in Expr1 with Expr2(var)
```

Cada nó retornado por `Expr1` é substituído pela seqüência de nós retornada por `Expr2` onde *var* está ligada a um nó. Note que `Expr2` é executada sobre o documento original sem levar em consideração atualizações intermediárias realizadas durante a execução dessa operação. Caso uma das seguintes condições acontecerem, um erro é retornado:

- Existem valores atômicos no resultado de `Expr1`.
- Existem nós temporários no resultado de `Expr1`.

A variável *var* pode ter uma declaração opcional de tipo. Se o tipo do valor da variável não corresponder ao tipo declarado, um erro é retornado. No seguinte exemplo, o salário das pessoas chamadas "Felipe"é duplicado.

```
UPDATE
replace \$p in doc("folha.xml")/bd/pessoa[nome="Felipe"]
with
<pessoa>
{(\$p/@*,
  \$p/node()[not(self::salario)],
  for \$s in \$p/salario
  return <salario>{\$s*2}</salario>)}
</pessoa>
```

RENAME

A operação RENAME é usada para modificar o nome de um elemento ou atributo.

```
UPDATE rename Expr on QName
```

A seguinte expressão modifica o nome dos elementos *cargo* sem alterar seu conteúdo.

```
UPDATE
rename //cargo[.="escovador de bits"] on profissao
```

APÊNDICE B

OPERAÇÕES DE ATUALIZAÇÃO - BENCHMARK XMARK

O XMark é um *benchmark* para dados XML desenvolvido pela Universidade de Amsterdam, Holanda. A estrutura do documento modelada é um site de leilões na Internet e os dados são gerados em um único documento. As entidades principais são pessoas, leilões abertos, leilões fechados, item e categoria. O esquema hierárquico é mostrado na Figura B.1.

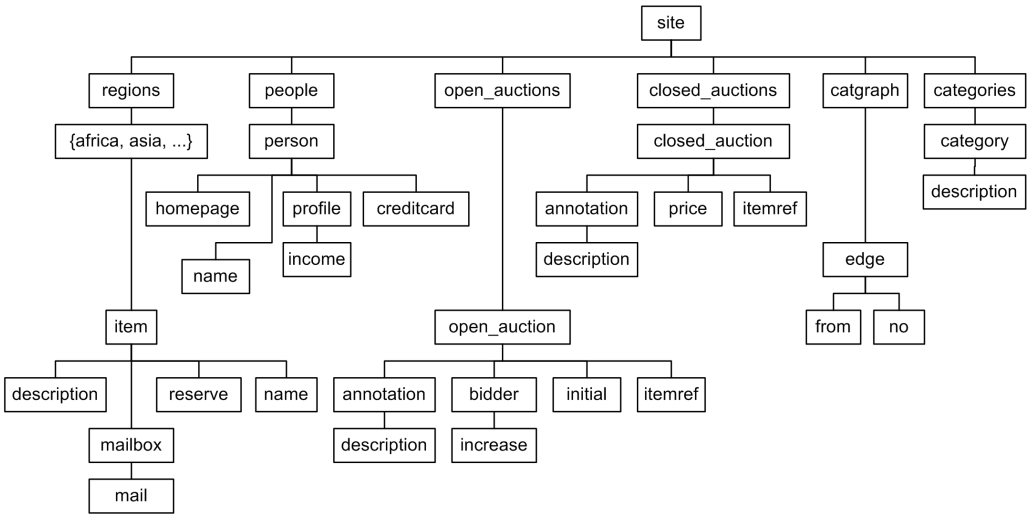

Figura B.1 Estrutura documento gerado pelo benchmark XMark

Os dados são gerados pelo *xmlgen*, um gerador de documentos. O *xmlgen* é independente de plataforma e escalável. As palavras usadas na criação dos documentos são as 17000 palavras mais comuns nas peças de teatro de Shakespeare. Essas palavras são selecionadas randomicamente, gerando um documento, de acordo com um DTD, que pode variar de 10 KB até 100 MB de tamanho.

O XMark especifica 20 consultas, cada uma delas explorando um aspecto particular do sistema. Contudo, o XMark não contém nenhuma operação de atualização.

Assim sendo, criamos seis dessas operações para utilização durante os experimentos de avaliação:

- U1. Cria um novo elemento *person* com informações como nome, telefone e número de cartão de crédito em sua subárvore, adicionando-a ao conjunto de elementos *person* do site (/site/people/).

  Exemplo:

```
UPDATE insert
<person id="person25">
  <name>John Smith</name>
  <phone>3222-2322</phone>
  <creditcard>3454 3656 2344 6767</creditcard>
</person>
into document("auction")/site/people
```

- U2. Insere um lance em um leilão aberto. Essa subárvore contém dados do lance como data, hora, valor de acréscimo e referência para a pessoa que fez o lance e é adicionada em um determinado leilão aberto em /site/open_auctions/.

  Exemplo:

```
UPDATE insert
<bidder>
  <date>04/09/2006</date>
  <time>11:57:28</time>
  <personref person="person11"/>
  <increase>41.50</increase>
</bidder>
into document("auction")/site/open_auctions/open_auction[@id = "open_auction15"]
```

- U3. Remove as pessoas (e suas subárvores) com um determinado nome de /site/people.

  Exemplo:

```
UPDATE delete
document("auction")/site/people/person[name/text()="John Smith"]
```

- U4. Duplica o preço de reserva de todos os leilões abertos em /site/open_auctions.

  Exemplo:

```
UPDATE replace
$a in document("auction")/site/open_auctions
with
<open_auction>
{($p/@*,
  $p/node()[not(self::reserve)],
  for $r in $p/reserve
  return <reserve>{$r * 2}</reserve>)}
</open_auction>
```

- U5. Cria um novo item. Sua subárvore contendo informações como nome do item, a categoria a qual ele pertence, localizaçao, etc. é adicionada em uma das regiões de /site/regions/.

  Exemplo:

```
UPDATE insert
<item id="item29">
<location>United States</location>
<quantity>1</quantity>
<name>balthasar bred breathe </name>
<payment>Cash</payment>
<description>
<text>
mistake treacherous springe <emph> absent lucius </emph> fairly
</text>
</description>
<shipping></shipping>
<incategory category="category39"/>
<mailbox>
<mail>
<from>Kerryn Cooke mailto:Cooke@ntua.gr</from>
<to>Tsunenori Lund mailto:Lund@msn.com</to>
<date>02/01/1999</date>
<text>
speaks indeed pocket her <emph> flight </emph> miserable field
</text>
</mail>
</mailbox>
```

```
</item>
into document("auction")/site/regions/namerica
```

- U6. Fecha um leilão aberto com um determinado ID, removendo sua entrada em /site/open_auctions, bem como sua subárvore.

  Exemplo:

```
UPDATE delete
document("auction")/site/open_auctions/open_auction[@id = "open_auction78"]
```

Para a avaliação do RepliX, utilizamos as seguintes consultas disponíveis no XMark:

- Q1. Retorna o nome da pessoa com o ID "person0"
- Q4. Lista o preço de reserva dos leilões abertos nos quais uma pessoa deu um lance antes de outra.
- Q5. Quantos itens vendidos têm preço maior que 40?
- Q6. Quantos itens estão listados em todos os continentes?
- Q6. Quantos itens estão listados em todos os continentes?
- Q7. Quantas prosas existem na base de dados?
- Q10. Lista todas as pessoas de acordo com seus interesses.
- Q13. Lista os nomes dos itens registrados na Austrália e suas descrições.
- Q14. Retorna os nomes de todos os itens cuja descrição contém a palavra "gold".
- Q15. Retorna as palavras-chave das anotações nos leilões fechados.
- Q16. Retorna os IDs dos leilões que têm uma ou mais palavras-chave.
- Q20. Agrupa os clientes por renda e retorna a cardinalidade de cada grupo.

APÊNDICE C

EXTENSÃO ADICIONADAS AO BANCO DE DADOS SEDNA

A implementação de mecanismos de replicação não raro exige a obtenção de informações sobre o estado interno do BD, por exemplo, os bloqueios e o estado das transações. A maioria dos BDs não provê acesso a esses dados por parte das aplicações, o que dificulta a implementação desses mecanismos. Um solução para esee problema é a alteração código-fonte do banco de dados, permitindo a adição de métodos de acesso as informações necessárias. Entretando, esse processo não é trivial, visto que BDs são aplicações que tendem a possuir códigos-fonte consideravelmente grandes. Qualquer modificação em uma aplicação desse porte requer um minucioso estudo do código-fonte e sua documentação de projeto, caso esta esteja disponível.

O procotolo de certificação no RepliX precisa determinar se duas transações estão em conflito. Esse tipo de situação pode ser identificado observando quais bloqueios cada transação em execução possui. O Sedna não fornece uma maneira de consultar essas informações, mas é um banco de código aberto, o que permitiu que essa funcionalidade fosse implementada. Nesta seção, descreveremos as alterações que foram feitas no código-fonte do Sedna para possibilitar o acesso a esses dados por aplicações clientes remotas.

A Comunicação

O Sedna usa um protocolo baseado em mensagens através de sockets TCP/IP para a comunicação entre os clientes e o servidor do Sedna. O formato da mensagem é o seguinte:

- Os primeiros quatro *bytes* representam um inteiro que identifica o código da instrução.
- Os quatro *bytes* seguintes representam um inteiro que identifica o tamanho do corpo da mensagem em *bytes*.

- Os próximos N *bytes* correspondem ao corpo da mensagem, onde N é o tamanho do corpo da mensagem descrito no item acima. O conteúdo do corpo da mensagem é determinado pela instrução. Uma limitação da versão atual do Sedna é que o tamanho do corpo da mensagem não pode ultrapassar 10240 *bytes*.

A descrição das mensagens trocadas entre o cliente e o servidor está fora do escopo desta seção, entretanto, para que a aplicação cliente possa requisitar e receber um conjunto de dados específicos do servidor do Sedna é preciso definir uma ou mais instruções para que ambas as partes saibam identificar o tipo de mensagem e processá-la corretamente. Na modificação que realizamos, introduzimos uma nova instrução chamada `se_ActiveTransactions` (o código escolhido para instrução foi 119, mas poderíamos escolher qualquer outro não utilizado).

Modificação do driver do Sedna

No lado da aplicação cliente, foi modificado o *driver* do Sedna para que este possa requisitar e receber as mensagens contendo os dados das transações ativas. Os arquivos referentes ao *driver* estão no diretório `driver/java/` a partir da raiz do código-fonte do Sedna. Primeiramente, adicionamos a constante referente à instrução do arquivo `NetOps.java` na linha 80:

```
final static int se_ActiveTransactions= 119;
```

Esse arquivo contém a definição da classe responsável pela comunicação com o servidor do Sedna através de sockets TCP/IP. Em seguida, modificamos o arquivo `SednaConnection.java`, que contém a definição da interface de manipulação da conexão com o Sedna, na linha 22 para inserir a assinatura do método `getActiveTransactions`:

```
public String getActiveTransactions() throws DriverException {
    NetOps.Message msg = new NetOps.Message();
    msg.instruction = NetOps.se_ActiveTransactions;
    msg.length = 0;
    NetOps.writeMsg(msg, outputStream);
    NetOps.readMsg(msg, bufInputStream);
    if (msg.instruction == NetOps.se_ErrorResponse)
        throw new DriverException(NetOps.getErrorInfo(msg.body,msg.length));
     if (msg.instruction != NetOps.se_ActiveTransactions)
        throw new DriverException(DriverException.SE3008);
return new String(msg.body, 0, msg.length);
```

```
}
```

Implementamos o novo método da interface na classe `SednaConnectionImpl`, contida no arquivo `SednaConnectionImpl.java`, na linha 148:

```
public String getActiveTransactions() throws DriverException;
```

Esse método utiliza a classe `NetOps` para enviar uma requisição e receber os dados referentes às transações ativas.

Modificação do servidor do Sedna

No servidor do Sedna, foram modificados os módulos de gerenciamento de transação (TR) e de gerenciamento de banco de dados (SM) para que o Sedna possa receber as requisições e responder com os dados das transações ativas. Os arquivos modificados estão no diretório `kernel/sm/` e `kernel/tr/` a partir da raiz do código-fonte do Sedna. Primeiramente, adicionamos a constante referente à instrução na enumeração `se_sp_instructions` no arquivo driver/c/`sp_defs.h` na linha 66:

```
se_ActiveTransactions = 119
```

Em seguida, modificamos o arquivo `client_code.h`, que contém a classe abstrata `client_core`. Essa classe define a interface de manipulação de clientes do Sedna. Alteramos o arquivo na linha 55 para inserir a assinatura do método `active_transactions`:

```
virtual void active_transactions(SSMMsg* sm_server) = 0;
```

Então, implementamos o método acima na subclasse concreta de `client_core` chamada `socket_client`. Essa classe contém os métodos que tratam as diversas requisições feitas pelo cliente através de sockets TCP/IP. No arquivo `socket_client.h`, linha 79, inserimos a assinatura do método:

```
virtual void active_transactions(SSMMsg* sm_server) = 0;
```

A implementação do método é feita no arquivo `socket_client.cpp`, linha 516:

```
void socket_client::active_transactions(SSMMsg* sm_server) {
  d_printf2("Active transactions\n");
  sp_msg.instruction = se_ActiveTransactions;
  sp_msg.length = 20;
  strcpy(sp_msg.body, active_transactions_from_sm(sm_server));
```

```
  if(sp_send_msg(Sock, &sp_msg)!=0) {Sock = U_INVALID_SOCKET;
     throw USER_EXCEPTION(SE3006);}\\
 }
```

Modificamos o arquivo `tr.cpp` responsável pelo tratamento de cada requisição recebida para reconhecer a instrução `se_ActiveTransactions` e chamar o método implementado acima. No laço principal do método `main`, inserimos o seguinte código na linha 446:

```
else if (client_msg.instruction == se_ActiveTransactions) {
          client->active_transactions(sm_server);
       }
```

Os dados referentes às transações ativas não se encontram no módulo TR e, sim, no módulo SM, responsável pelo gerenciamento do banco de dados. A comunicação entre os módulos do Sedna é feita através de um sistema de mensagens implementado em cima de áreas de memória compartilhada entre os processos. O código do método utiliza a classe `SSMMsg` para enviar uma mensagem ao módulo SM que obtém as informações requisitadas sobre as transações ativas e a retorna ao requerente:

```
 void *active_transactions_from_sm(SSMMsg* sm_server) {
    sm_msg_struct msg;
    msg.cmd = 119;
    if (sm_server->send_msg(&msg) !=0 )
      throw USER_EXCEPTION(SE3034);   }
   return msg.data;
}
```

Finalmente, no módulo SM, modificamos método `sm_server_handler`, no arquivo `sm.cpp`, encarregado de processar as mensagens recebidas dos outros módulos.

```
case 119:  {//get active transactions
              msg->data = get_active_transactions_data();
               break;
           }
```