UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA REPLICAÇÃO DE DADOS NO SGBD POSTGRESQL

MALCUS OTÁVIO QUINOTO IMHOF

BLUMENAU
2007

2007/1-27

MALCUS OTÁVIO QUINOTO IMHOF

FERRAMENTA PARA REPLICAÇÃO DE DADOS NO SGBD POSTGRESQL

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Alexander Roberto Valdameri – Mestre, FURB

BLUMENAU
2007

2007/1-27

FERRAMENTA PARA REPLICAÇÃO DE DADOS NO SGBD POSTGRESQL

Por

MALCUS OTÁVIO QUINOTO IMHOF

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Alexander Roberto Valdameri – Mestre, FURB
Membro:	Prof. Adilson Vahldick, Especialista – FURB
Membro:	Prof. Paulo Fernando da Silva, Mestre – UFSC

Blumenau, 29 de junho de 2007

Dedico este trabalho a minha família, principalmente aos meus pais, meus amigos, colegas de classe e meu orientador, que me estimularam na elaboração deste trabalho.

AGRADECIMENTOS

À minha família que sempre me apoiou nas minhas escolhas, principalmente aos meus pais Hélio e Ana.

Aos meus amigos pela compreensão e estímulos dados aos meus estudos.

Ao meu orientador, Alexander Roberto Valdameri, pelo seu empenho demonstrado na orientação.

Ninguém é tão grande que não possa aprender,
nem tão pequeno que não possa ensinar.

RESUMO

O presente trabalho apresenta o desenvolvimento de uma ferramenta para replicação assíncrona de dados, multiplataforma, específica para o SGBD PostgreSQL, com tolerância a falta de comunicação, nodos secundários atualizáveis e com a garantia de consistência dos dados, utilizando para detectar e resolver conflitos as técnicas de versionamento de linhas e propriedade do registro.

Palavras-chave: Banco de dados. Replicação. Sistemas distribuídos. PostgreSQL.

ABSTRACT

This project presents the implementation of an asynchronous replication, multiplatform, specific for PostgreSQL database, with communication fault tolerance, secondary site updatable and data consistency warranties, using to detect e solve conflicts row-version and ownership techniques.

Key-words: Database. Replication. Distributed systems. PostgreSQL.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API - *Application Programming Interface*

RF - Requisito Funcional

RMI – Remote Method Invocation

RNF – Requisito Não Funcional

RPC – *Remote Procedure Calls*

SGBD – Sistema Gerenciador de Banco de Dados

UML - *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Segundo Coulouris, Dollimore e Kindberg (2001, p. 553), replicação de dados é a chave para prover alta disponibilidade e tolerância a falhas em sistemas distribuídos. Atualmente grandes empresas que, por conseqüência, possuem um grande repositório de dados promovem a distribuição dos mesmos em suas unidades geograficamente distribuídas. Neste contexto, aplicam-se os Sistemas Gerenciadores de Bancos de Dados (SGBDs). O PostgreSQL é um dos SGBDs que mais tem despertado o interesse das empresas de software.

Uma das razões pelas quais o PostgreSQL tem sido explorado está relacionada a sua origem, por ser um produto aberto, sem restrições de utilização e de custo zero. Porém, em se tratando de replicação de dados, o PostgreSQL não apresenta recursos que contemplam as necessidades de replicar dados entre bases de dados em sistemas operacionais distintos, citando como exemplo Microsoft Windows e Linux.

Atualmente os replicadores de dados existentes para o SGBD PostgreSQL estão operando exclusivamente sobre o sistema operacional Linux, inviabilizando o uso do mesmo em ambientes Windows. Neste contexto, este trabalho pretende viabilizar uma ferramenta de replicação que opere tanto em Linux como em Microsoft Windows.

A idéia contida neste trabalho é o desenvolvimento de uma ferramenta de replicação de dados assíncrona[1], utilizando para tanto a linguagem de programação Java.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é elaborar uma ferramenta de replicação de dados para o SGBD PostgreSQL.

Os objetivos específicos do trabalho são:

a) operar nos sistemas operacionais Microsoft Windows 2000 Server Family, Microsoft Windows 2003 Server Family, OpenSuse Linux 10.0 e RedHat Enterprise Linux 3;

b) caracterizar uma replicação assíncrona, onde as réplicas podem ficar fora de

[1] Replicação assíncrona: as réplicas podem ficar temporariamente fora de sincronia.

sincronia num primeiro momento e a replicação dos dados é executada num segundo momento;

c) manter consistência entre as instâncias do SGBD PostgreSQL;
d) realizar atualização remota dos dados;
e) resolver conflitos na replicação de dados;
f) controlar o fluxo de informações, obedecendo o sincronismo das bases de dados.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em quatro capítulos que estão descritos a seguir:

O primeiro capítulo contextualiza e justifica o desenvolvimento do trabalho.

No segundo capítulo é disponibilizada a fundamentação teórica necessária para um razoável conhecimento dos assuntos utilizados no desenvolvimento do trabalho.

O terceiro capítulo tem como foco o desenvolvimento do aplicativo, descrevendo os requisitos principais do problema como também a especificação e implementação.

O quarto capítulo apresenta as conclusões finais e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O trabalho engloba os temas sistemas distribuídos, banco de dados, PostgreSQL, replicação de dados e Java RMI, além de trabalhos correlatos.

2.1 SISTEMAS DISTRIBUÍDOS

De acordo com Date (2000, p. 617), sistema distribuído é “qualquer sistema que envolve múltiplas localidades conectadas [...], nas quais o usuário [...] de qualquer localidade pode acessar os dados armazenados em outro local.”

Outra abordagem define sistemas distribuídos como uma rede de computadores que possui vários componentes se comunicando através da coordenação de mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2001, p. 553). Dentro da área de sistemas distribuídos encontram-se os bancos de dados distribuídos.

Um sistema computacional distribuído consiste em uma coleção de computadores autônomos interligados por uma rede de comunicação e equipados com um sistema operacional distribuído. Esse sistema operacional permite que os computadores coordenem suas atividades e compartilhem os diversos recursos do sistema, como hardware, software ou dados (TANENBAUM, p. 02, 1995).

O desenvolvimento de sistemas distribuídos tornou-se possível com o surgimento das redes locais de computadores no início da década de 70. Essas redes permitiram que, computadores que antes trabalhavam de maneira isolada, pudessem ser interligados, trazendo vários benefícios para a computação. Mais recentemente, a disponibilidade de computadores pessoais e estações de trabalho de maiores desempenhos possibilitou que sistemas distribuídos fossem implementados com grandes vantagens em relação aos sistemas centralizados, em termos de desempenho, disponibilidade e custos. Além disso, a confiabilidade, poder de expansão da capacidade computacional do sistema e a natureza de algumas aplicações que são inerentemente distribuídas, tornam os sistemas distribuídos cada vez mais difundidos (TANENBAUM, p. 02 ,1995).

Date (2000, p. 618) afirma que dois bancos de dados conectados a fim de formar um único banco de dados distribuídos global têm como vantagens a eficiência do processamento e

maior acessibilidade.

2.2 BANCO DE DADOS

Um banco de dados é uma coleção de dados relacionados. Os dados são fatos que podem ser gravados e que possuem um significado implícito. As informações são uma coleção de dados com um significado. (ELMARSI, p. 4, 2005).

Segundo Elmarsi (2005, p.4) um sistema gerenciador de banco de dados é uma coleção de programas que permite aos usuários criar e manter um banco de dados, facilitar os processos de definição, construção, manipulação e compartilhamento de banco de dados entre vários usuários e aplicações.

A definição de um banco de dados implica em especificar os tipos de dados, as estruturas e as restrições para os dados a serem armazenados em um banco de dados.

Date (2000 ,p.4) afirma que um SGBD é basicamente um sistema computadorizado de armazenamento de registros cujo propósito geral é o armazenamento de informações e proporcionar a busca e atualização destas informações.

Um banco de dados é uma coleção integrada de dados onde o mesmo envolve os próprios dados, o hardware em que os dados residem, o software que controla o armazenamento e os usuários (DEITEL; DEITEL, p. 812, 2001).

2.3 POSTGRESQL

Segundo PostgreSQL BR (2003), PostgreSQL é “[...] um SGBD (Sistema Gerenciador de Banco de Dados) objeto-relacional de código aberto, com mais de 15 anos de desenvolvimento. É extremamente robusto e confiável, além de ser extremamente flexível e rico em recursos.”

O PostgreSQL também tem como característica seguir as normas estabelecidas no SQL:2003, além de oferecer importantes recursos, tais como: *queries* complexas, chaves estrangeiras, *triggers*, *views* e integridade transacionais. Além de possuir tais recursos, ele é extensível, ou seja, é possível a criação de funções, operações, métodos de índices e

linguagens procedurais (POSTGRESQL, 2005).

Entretanto, como o SGBD PostgreSQL é um produto aberto, não há uma equipe própria que possa fornecer o devido suporte técnico. O seu suporte é totalmente dependente de seus usuários, que formam grupos de discussões e fóruns. Por estes motivos, soluções para problemas que venham ocorrer podem não ser facilmente encontradas.

2.4 REPLICAÇÃO DE DADOS

Replicação de dados é uma técnica para obter uma maior disponibilidade de serviços. Os benefícios para se fazer a replicação são o aumento de desempenho obtido por parte de aplicativos que acessem seus dados e o aumento de disponibilidade das informações (COULOURIS; DOLLIMORE; KINDBERG, 2001, p. 553). A replicação para aumentar o desempenho é importante quando o sistema distribuído necessita obter acessos em distintas áreas geográficas (TANENBAUM, 2002, p. 453).

Em replicação de dados, são encontradas duas classificações, a replicação assíncrona e síncrona.

Na replicação assíncrona, se um banco é alterado, a alteração será propagada e aplicada para outro(s) banco(s) num segundo passo, sendo que esta poderá ocorrer em segundos, minutos, horas ou até dias depois (BEEHIVE, 2006). Uma desvantagem é que este tipo de replicação tende a atrasar a detecção de conflitos entre as operações, o que só acontece no fim da execução das transações.

Na replicação síncrona todas as cópias ou replicações de dados serão feitas no instante da sincronização, mantendo a consistência dos dados. Se alguma cópia do banco é alterada, essa alteração será imediatamente aplicada a todos os outros bancos dentro da transação. Entretanto, este tipo de replicação traz uma série de desvantagens como, por exemplo, uma transação não poderá ser concluída se um dos nodos que possuem réplicas estiver indisponível (DUARTE, 2006).

2.4.1 Técnicas de replicação de dados

A técnica *master to multiple slavers* consiste na existência de um nodo principal e

nodos secundários, onde as atualizações só são permitidas no nodo principal. Os nodos secundários possuem uma réplica das informações, porém somente para leitura (BRAGA. 2001, p.30).

A técnica de replicação ponto a ponto possui a característica de não possuir um nodo primário, permitindo assim que sejam atualizados quaisquer nodos. Contudo, com esta técnica os conflitos devem ser resolvidos posteriormente, no memento de sincronia dos nodos.

Possui como principal vantagem a atualização de todos os dados localmente. Em contrapartida este modelo deve tolerar inconsistências globais quando os nodos não estiverem sincronizados, além da necessidade de resolução dos eventuais conflitos gerados (BRAGA. 2001, p.42).

2.4.2 Conflitos na replicação de dados

Os conflitos na replicação assíncrona podem utilizar a estratégia de versão para as linhas consiste em acrescentar, na tabela, uma nova coluna, que guardará a versão da linha. Assim, ao se fazer a atualização de uma linha, é verificada a versão da linha corrente. Desta forma, pode-se determinar se há ou não conflito na sua atualização (BRAGA. 2001, p.44).

Alguns cenários para resolução de conflitos:

- Prioridade de Nodo: cada site possui sua prioridade. O site que possuir maior prioridade terá efetivada a sua transação. Essa resolução pode ser implementada através de *trigger* ou *stored procedures* que, por exemplo, devem consultar uma tabela que armazene um critério de prioridade entre os sites existentes;
- Tempo Global: é uma resolução um pouco mais complexa, pois pressupõe que todos os sites mantêm em sincronia os seus relógios. Essa resolução também é implementada usando-se *trigger* ou *stored procedures*;
- Atualização Comutativa: a base de sua estrutura é ignorar o conflito. Esse tipo de resolução só é viável em algumas situações onde o valor correto do dado é obtido ignorando o conflito das alterações. Essa situação prevê a mesma prioridade para todos os sites;
- Manual: o conflito não é automaticamente solucionado. O conflito é gravado no *log* e resolvido manualmente. Dessa forma, os dados podem ficar

inconsistentes por um longo período de tempo.

2.5 JAVA RMI

O Java RMI permite que objetos executando no mesmo computador ou em computadores separados se comuniquem entre si via chamadas de método remoto. Essa tecnologia está baseada em uma tecnologia anterior semelhante, utilizada para programação procedural, chamada RPC que foi desenvolvida nos anos de 1980 (DEITEL; DEITEL, p. 891, 2001).

O RMI é a implementação do RPC para o Java, destinada a comunicação distribuída entre objetos Java. Quando um objeto é registrado como remotamente disponível, um outro aplicativo Java pode utilizar este objeto remoto, desde que consiga fazer o *lookup*[2] deste objeto (DEITEL; DEITEL, p. 891, 2001).

O Java RMI permite o programador criar aplicações distribuídas em Java, onde os métodos de objetos remotos podem ser invocados por outras máquinas virtuais Java, possivelmente em *hosts* diferentes. O RMI não trunca os tipos de dados, suportando o polimorfismo existente na orientação a objetos (SUN DEVELOPER NETWORK, 2007).

A seguir mostra-se um exemplo de código fonte de um aplicativo que utiliza o Java RMI. No Quadro 1 é especificada a interface a ser implementada.

```
import java.net.*;
import java.rmi.*;
public interface Hello extends Remote {
    String sayHello() throws RemoteException;}
```

Quadro 1 – Definição da Interface

Quadro 2 demonstra a implementação da interface definida no Quadro1.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException{
        super();
    }
    public String sayHello() {
        return "HelloWorld!";}}
```

Quadro 2 – Implementação da Interface

[2] Processo para receber a referência remota de um objeto.

O Quadro 3 exibe a implementação do aplicativo servidor e seu registro no *rmiregistry*.

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloServer {
    public static void main (String args [ ]) {
        //Cria e instala o security manager
        //System.setSecurityManager(new RMISecurityManager() );
        try {
            //Cria HelloImpl
            HelloImpl obj = new HelloImpl();
            Naming.rebind("HelloServer", obj);
            System.out.println("Hello Server pronto.");
        } catch(Exception e) {
            System.out.println("HelloServer erro"+ e.getMessage());
        }
      }
}
```

Quadro 3 – Implementação do aplicativo servidor

O Quadro 4 mostra a implementação de um aplicativo cliente e a obtenção da referência remota.

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class HelloClient {
    public static void main(String args[ ]) {
        try {
        Hello obj = (Hello)Naming.lookup("rmi://vip03/HelloServer");
            System.out.println(obj.sayHello());
        } catch(Exception e) {
            System.out.println("HelloClient erro"+ e.getMessage());
    }
}
```

Quadro 4 – Implementação do aplicativo servidor

2.6 TRABALHOS CORRELATOS

A seguir são apresentados o trabalho de Souza Júnior (2003), as ferramentas Slony (GBORG DEVELOPMENT TEAM, 2006b), ERServer (GBORG DEVELOPMENT TEAM, 2006a) e Postgres-R (GBORG DEVELOPMENT TEAM, 2006c).

2.6.1 Software para replicação de objetos entre duas instâncias de um SGBD Oracle

Souza Júnior (2003, p. 2) desenvolveu um projeto no qual o "objetivo principal é o desenvolvimento de um software para realizar cópias entre duas instâncias de um SGBD Oracle 8i com o intuito de auxiliar o desenvolvedor na automatização do controle das restrições existentes por falta de relacionamentos existentes entre objetos."

Este trabalho também teve como característica o controle de permissões de objetos para usuários, além do gerenciamento da duplicidade de chaves e *foreing keys*/tabelas inexistentes, e possuir um histórico das cópias efetuadas (SOUZA JUNIOR, 2003, p. 2).

Souza Júnior (2003, p. 2) relata que o trabalho "tem por finalidade solucionar uma grande deficiência existente em diversas empresas no que tange à criação, manutenção, integridade e confiabilidade de uma base de dados adequada a ser utilizada em uma das principais etapas do desenvolvimento do sistema, no caso, a fase de testes."

2.6.2 Slony

O projeto Slony surgiu a partir da idéia de existir um replicador de dados que não fosse específico para nenhuma versão do SGBD PostgreSQL (GBORG DEVELOPMENT TEAM, 2006b).

Slony é um gerenciador de replicação *master to multiple slavers*, que tem por objetivo operar em *data centers* e como *backup on-line*. Slony não é indicado quando a ocorrência de quedas dos nodos seja freqüente. O projeto Slony efetua a replicação síncrona e assíncrona (GBORG DEVELOPMENT TEAM, 2006b).

Uma característica apresentada por este replicador de dados é a não detecção na falha de um nodo e também a não eleição de um nodo *master*, caso este venha, por algum motivo, não estar disponível (GBORG DEVELOPMENT TEAM, 2006b).

2.6.3 ERServer

Segundo GBorg Development Team (2006a.), o ERServer é um replicador assíncrono *master to multiple slavers*, baseado em *trigger*. Este replicador de dados foi descontinuado

pelos desenvolvedores para o desenvolvimento do Slony.

A idéia contida no desenvolvimento do ERServer foi a de desenvolver uma ferramenta que teria a capacidade de efetuar a replicação de dados em tempo real. Contudo, esta replicação seria apenas *read only* nas base de dados *slaves*, ou seja, inserções, alterações e exclusões somente poderiam ser efetuadas na base de dados *master*.

2.6.4 Postgres-R

O Postgres-R, um replicador síncrono, é tido como um modelo de replicador de base de dados, no qual foi baseado no SGBD PostgreSQL 6.4.2 e posteriormente migrado para a versão 7.2, além de ser considerado a mais avançada solução *open source* existente (GBORG DEVELOPMENT TEAM, 2006c). O uso deste replicador é aconselhado para redes locais, para uma replicação síncrona. Esta ferramenta foi implementada para ambientes Linux.

Os autores desta ferramenta propuseram a replicação imediata (*eager*) ao invés de uma replicação deferida (*lazy*[3]). Esta ferramenta também tem como característica a utilização de cópias *shadow*. Com estas cópias, foi possível contornar o problema de excessivas mensagens para efetuar a replicação, visto que com ela, consistências de leitura e escrita poderiam ser feitas na cópia, assim enviando um agrupamento de mensagens para as réplicas (LORÊDO, FERREIRA, ASSIS, 2004).

[3] Os efeitos executados são propagados tardiamente para as outras réplicas.

3 DESENVOLVIMENTO DO APLICATIVO

Neste capítulo são apresentados os requisitos, a especificação, a implementação, a operacionalidade do aplicativo, e os resultados e discussões.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O aplicativo desenvolvido neste trabalho é responsável pela realização da replicação assíncrona entre SGBDs PostgreSQL. O desenvolvimento foi efetuada na linguagem Java utilizando recursos do próprio SGBD para obter as informações a serem replicadas. Para a troca de informações entre os SGBDs foi utilizado o Java RMI.

Para ser desenvolvido, foi necessário apontar e analisar os requisitos do problema a ser tratado. Também foi necessária a especificação do aplicativo desenvolvido.

A seguir são apresentados os requisitos a qual o aplicativo seguiu para o seu desenvolvimento e como o mesmo deve se comportar:

a) replicar os dados após restabelecer a conexão quando da ocorrência de alguma falha (RF);
b) replicar o conjunto de dados de aplicações (RF);
c) permitir a configuração de parâmetros de uma replicação (como tempo e tabelas) (RF);
d) resolver conflitos na replicação de dados utilizando o conceitos de versão para linhas e prioridade do site (RF);
e) utilizar o SGBD PostgreSQL versão 8.0 (RNF);
f) implementar o projeto utilizando a linguagem de programação Java (RNF);
g) desenvolver utilizando para tanto ferramentas de programação *open source* ou *freeware* (RNF);
h) possibilitar a replicação assíncrona entre diversos sistemas operacionais (RNF).

A ferramenta apresenta o mesmo comportamento em sistemas operacionais que suportem a linguagem Java, ou seja é possível que um determinado nodo de origem esteja sendo executando em um determinado sistema operacional enquanto o nodo de destino esteja sendo executando em outro.

Caso a conexão entre dois determinados nodos não se estabeleça no momento em que deveria ocorrer a replicação dos dados, o aplicativo executa a replicação em uma próxima conexão com o nodo de destino.

A ferramenta replica os registros incluídos, alterados ou excluídos de um determinado banco de dados. Alterações na estrutura das tabelas, a criação de novas tabelas e seqüências não serão replicadas.

O funcionamento deste aplicativo necessita de um arquivo de configuração, na qual é utilizado em duas etapas: na inicialização do SGBD para começar a coletar informações para serem replicadas e a execução em si do replicador.

A configuração do replicador é feita através de um arquivo texto no qual contém informações como: *schema* de armazenamento, seleção de tabelas ou *schemas* para replicar, tempo entre a execução da replicação, nome do nodo, prioridade do nodo, conexão com a base de dados e a lista de nodos destinos.

A resolução de conflitos entre os registros é feito com a combinação de duas técnicas, versão de linha e prioridade do site.

O SGBD a ser utilizado como base para o desenvolvimento da ferramenta de replicação é PostgreSQL na sua versão 8.0.

Para a implementação deste aplicativo, devem ser utilizados ferramentas com a licença do tipo *open source* ou *freeware*.

3.2 ESPECIFICAÇÃO

Esta seção do trabalho exibe a especificação do aplicativo desenvolvido, utilizando para tanto os diagramas da UML. Os diagramas utilizados para fazer a especificação foram os de caso de uso, classes e de seqüência.

3.2.1 Ferramenta utilizada na especificação

Para a especificação do trabalho foi utilizada a ferramenta *Enterprise Architect* da SPARX SYSTEMS.

Antes de se decidir por utilizar esta ferramenta foi cogitado o uso da ferramenta *Microsoft Visio* da *Microsoft Corporation*, contudo seu uso foi abandonado pela razão que não se dispunha tempo para o aprendizado desta ferramenta, visto que a e ferramenta utilizada e suas funcionalidades na especificação são conhecidos previamente.

O *Enterprise Architect* é uma ferramenta utilizada na especificação de sistemas, que oferece ferramentas para modelar as estruturas e funcionalidades de um aplicativo. Esta ferramenta baseia-se na linguagem UML (SPARX SYSTEMS, 2007).

3.2.1.1 Replicar dados

A Figura 3 ilustra o diagrama de atividades “replicar dados”.

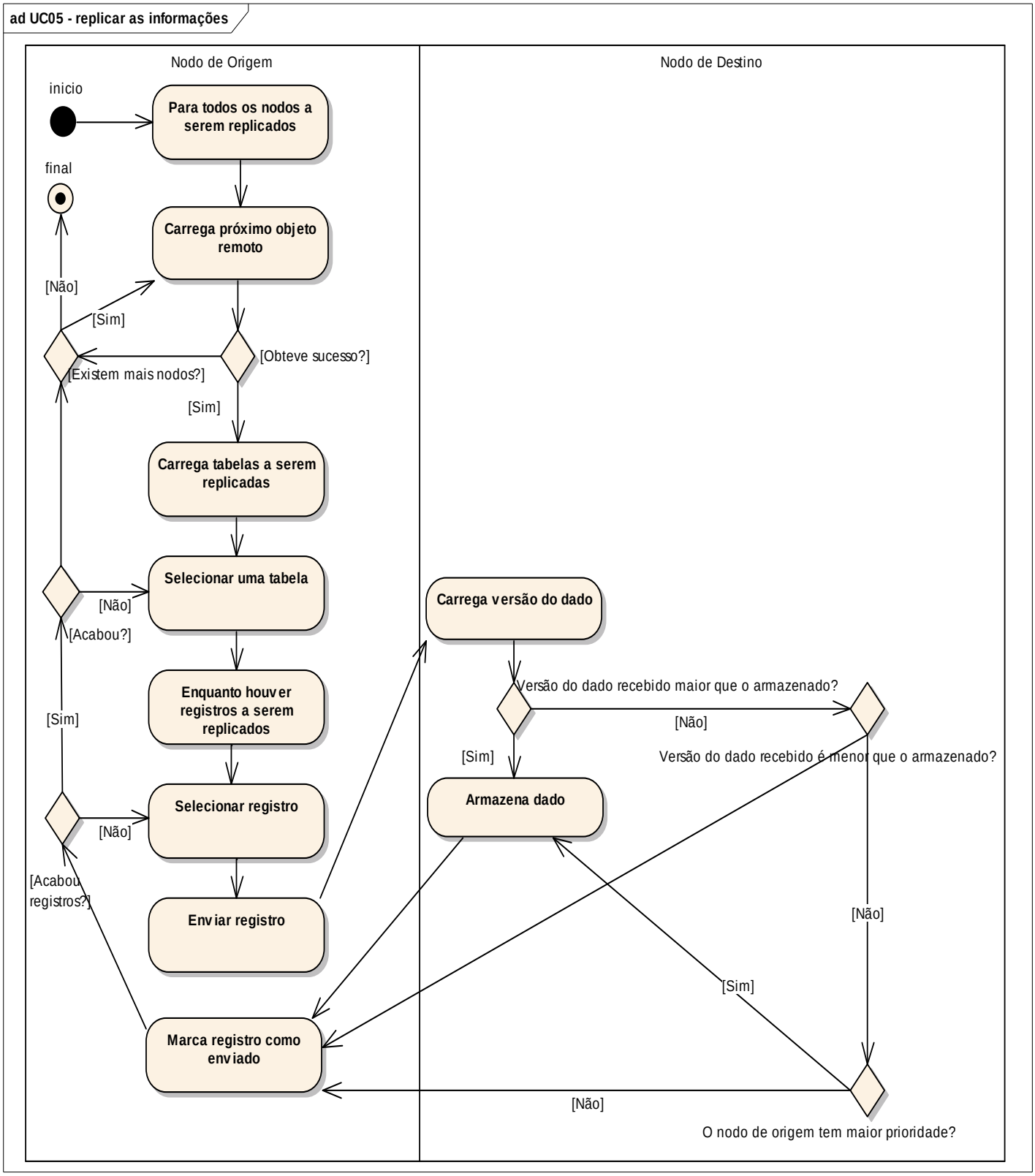

Figura 1 – Diagrama de atividades “replicar dados”

O usuário inicia o serviço de replicação onde para cada nodo de destino é obtida a referência do objeto remoto correspondente, conforme configurado no arquivo de configuração. Em caso de falha na tentativa de conexão com o nodo destino, o aplicativo faz a

conexão com o próximo nodo destino, deixando para tentar conectar no próximo ciclo de tempo.

Após a obtenção de todas as referências dos objetos remotos disponíveis, buscam-se a tabelas que foram configuradas para replicar os dados.

Para cada tabela a ser replicada, verifica-se a existência de dados para fazer a sincronização dos dados, utilizando as tabelas de armazenamento[4]. A seleção é definida pela chave primária, uma numeração seqüencial gerenciada pelo SGBD.

No momento que é encontrado algum dado a ser replicado, o mesmo é preparado para ser enviado para o nodo de destino através do objeto remoto na qual aguarda o recebimento das informações.

Ao receber o registro a ser atualizado do nodo de origem, verifica-se qual é a chave primária do registro de origem. Com esta informação, é obtida a versão de linha que se encontra no nodo de destino.

Neste momento, é determinado se existe ou não um conflito na atualização do dado. A primeira validação verifica se o nodo de origem possui uma versão de linha maior que a do nodo destino. Em caso afirmativo, o dado de origem é armazenado na estrutura de dados do nodo destino, sem que haja o incremento do número de versão, e envia-se a confirmação para o nodo de origem. Apurado que o dado de origem não possui versão maior, verifica-se se ela é menor e sendo verdadeiro, devolve-se a confirmação ao servidor de origem..

Na situação que é diagnosticado que existe o conflito de informações entre os dados de origem e destino, aplica-se a técnica para resolvê-lo. Para resolver este conflito, compara-se a prioridade entre os nodos e o que possuir maior prioridade é considerado como o dado a ser armazenado dentro da base de dados. Após a resolução do conflito é enviado uma resposta ao nodo de origem.

Encontrado algum problema no lado do nodo destino, onde por algum motivo não se pode sincronizar as bases de dados, devolve-se uma resposta ao nodo de origem informando que não foi executada com sucesso a atualização do dado.

3.2.2 Diagrama de classes

A utilização do diagrama de classes foi de fundamental importância no

[4] Tabelas que contem os registros a serem replicados

desenvolvimento do aplicativo. Com a utilização do diagrama de classes foi possível definir os atributos e métodos necessários para o inicio do desenvolvimento do aplicativo.

A ferramenta utilizada na modelagem do diagrama de classes foi o *Enterprise Architect* e com o auxílio da mesma foi possível gerar o código fonte inicial, sendo necessários apenas alguns ajustes para o início do processo de desenvolvimento.

O aplicativo desenvolvido neste trabalho necessitou a utilização de oito classes e uma interface necessária para a utilização do Java RMI.

A Figura 4 exibe o diagrama de classes utilizado para a execução deste trabalho.

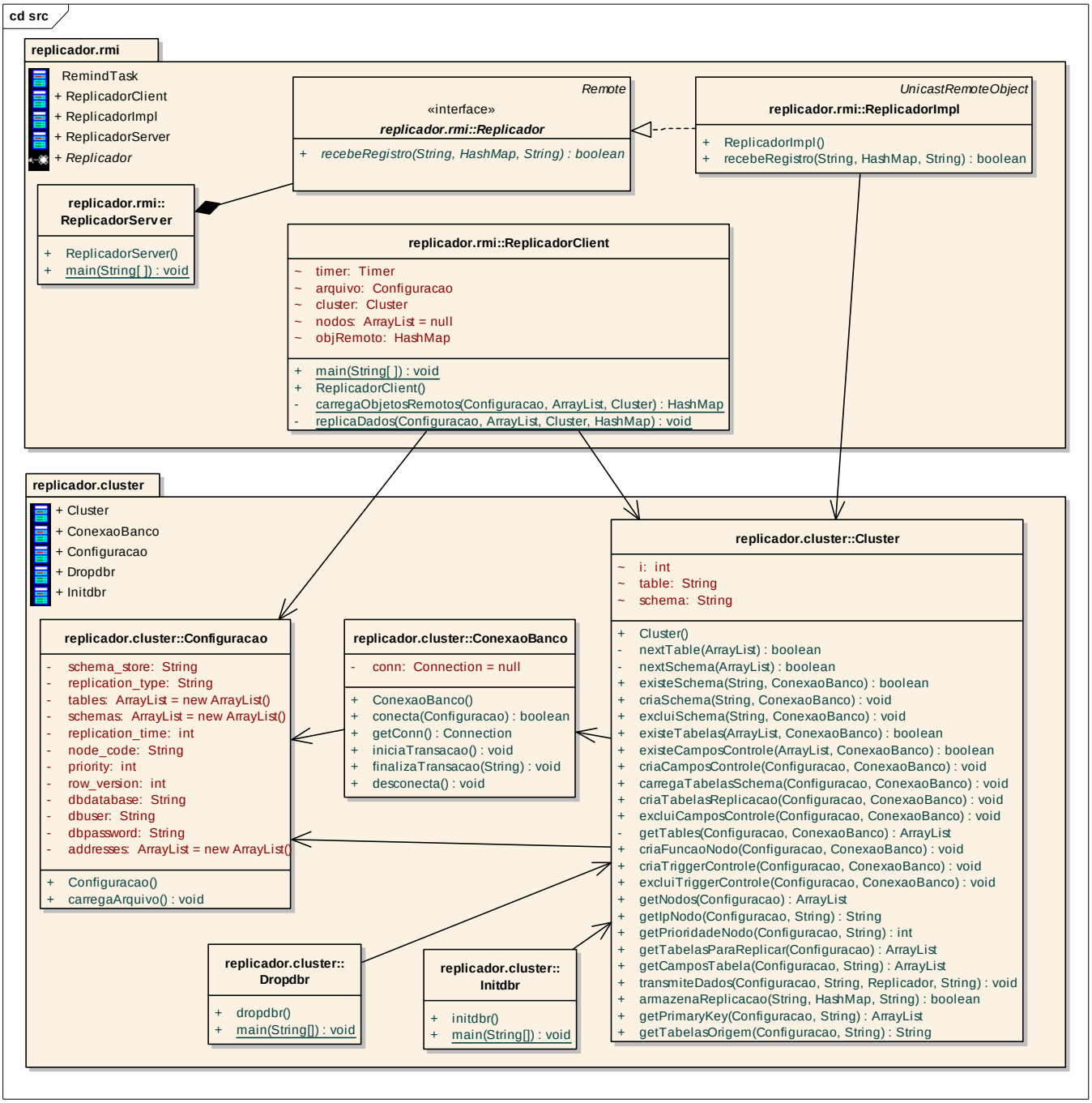

Figura 2 – Diagrama de classes

O diagrama foi construído em dois pacotes. O pacote `replicador.cluster` tem classes que estão direcionadas a execução de funções e não de serviços. O pacote `replicador.rmi` contém os principais serviços a serem executadas, a aplicação cliente e a aplicação servidor. A seguir é exibida uma breve descrição sobre as classes `Configuracao` e `Cluster,` ambas pertencentes ao pacote `replicador.cluster`.

A classe `Configuracao` possui métodos responsáveis pela leitura do arquivo de configuração do aplicativo. O Quadro 5 exibe os métodos implementados dessa classe.

Sumário de Métodos	
void	**carregaArquivo**() Método responsável pela leitura do arquivo de configuração
java.util.ArrayList	**getAddresses**() retorna a lista de nodos de destino
java.lang.String	**getDbdatabase**() retorna o nome da base de dados
java.lang.String	**getDbpassword**() retorna a senha do usuário utilizado para conectar a base de dados
java.lang.String	**getDbuser**() retorna o nome do usuário utilizado para conectar a base de dados
java.lang.String	**getNode_code**() retorna o nome do nodo
int	**getPriority**() retorna a prioridade do nodo
int	**getReplication_time**() retorna o ciclo de tempo utilizado para fazer a replicação
java.lang.String	**getReplication_type**() Retorna o tipo de replicação utilizada schema ou tables
java.lang.String	**getSchema_store**() Retorna o nome do schema de armazenamento
java.util.ArrayList	**getSchemas**() retorna a lista de schemas
java.util.ArrayList	**getTables**() Retorna a lista de tabelas
void	**setTables**(java.util.ArrayList tables) atualiza a lista de tabelas a serem replicadas

Quadro 5 – Métodos implementados na classe “Configuracao”

A classe `Cluster` implementa métodos utilizados para manipulação de dados dentro da base de dados. Esta classe é de vital importância dentro do aplicativo, pois nela estão implementados os principais métodos responsáveis pela replicação, tais como a estruturação

das tabelas de armazenamento dos dados a serem replicados, bem como seu controle e a resolução de conflitos. O Quadro 6 exibe os métodos implementados dessa classe.

Sumário de Métodos	
boolean	**armazenaReplicacao**(java.lang.String tabelaOrigem, java.util.HashMap campos, java.lang.String nodo) Método para efetivar a replicação de dados
void	**carregaTabelasSchema**(Configuracao arquivo, ConexaoBanco conexao) Método para carregar todas as tabelas em um ArrayList de um schema
void	**criaCamposControle**(Configuracao arquivo, ConexaoBanco conexao) Método para inicializar os campos "row_version" e "node_code"
void	**criaFuncaoNodo**(Configuracao arquivo, ConexaoBanco conexao) Função para criar a função que retorna o nome do nodo
void	**criaSchema**(java.lang.String schemaName, ConexaoBanco conexao) Método para criar schema
void	**criaTabelasReplicacao**(Configuracao arquivo, ConexaoBanco conexao) Metodo para criação das tabelas que serão replicadas
void	**criaTriggerControle**(Configuracao arquivo, ConexaoBanco conexao) Função que cria as trigger para controle
void	**excluiCamposControle**(Configuracao arquivo, ConexaoBanco conexao) Método para excluir os campos "row_version" e "node_code"
void	**excluiSchema**(java.lang.String schemaName, ConexaoBanco conexao) Método para excluir schema
void	**excluiTriggerControle**(Configuracao arquivo, ConexaoBanco conexao) Método para excluir triggers e functions
boolean	**existeCamposControle**(java.util.ArrayList array, ConexaoBanco conexao) Método para verificar a existecia dos campos "row_version" e "node_code"
boolean	**existeSchema**(java.lang.String schemaName, ConexaoBanco conexao) Método para verificar se existe o schema para armazenamento das tabelas a serem replicadas
boolean	**existeTabelas**(java.util.ArrayList array, ConexaoBanco conexao) Método para verificar a existencia de todas as tabelas no ArrayList
java.util.ArrayList	**getCamposTabela**(Configuracao arquivo, java.lang.String tabela)

java.lang.String	**getIpNodo**(Configuracao arquivo, java.lang.String nodo) retorna o ip do nodo
java.util.ArrayList	**getNodos**(Configuracao arquivo) Método para retornar a lista de nodos
java.util.ArrayList	**getPrimaryKey**(Configuracao arquivo, java.lang.String tabela) Método que retorna a chava primária de uma tabela
int	**getPrioridadeNodo**(Configuracao arquivo, java.lang.String nodo) Retorna a prioridade do nodo
java.lang.String	**getTabelasOrigem**(Configuracao arquivo, java.lang.String tabelaReplicada)
java.util.ArrayList	**getTabelasParaReplicar**(Configuracao arquivo) Método que retorna as tabelas a serem replicadas
void	**transmiteDados**(Configuracao arquivo, java.lang.String tabela, Replicador obj, java.lang.String nodo) Método responsável pela busca dos dados e enviá-los através do objeto remoto

Quadro 6 – Métodos implementados na classe “Cluster”

3.2.3 Tabelas de armazenamento e controle

As tabelas de armazenamento e controle têm como objetivo guardar as informações a serem replicadas e quais os nodos que já tiveram a sincronização efetuada para determinado registro.

A seguir é mostrado em exemplo do Modelo Entidade Relacional das tabelas de armazenamento.

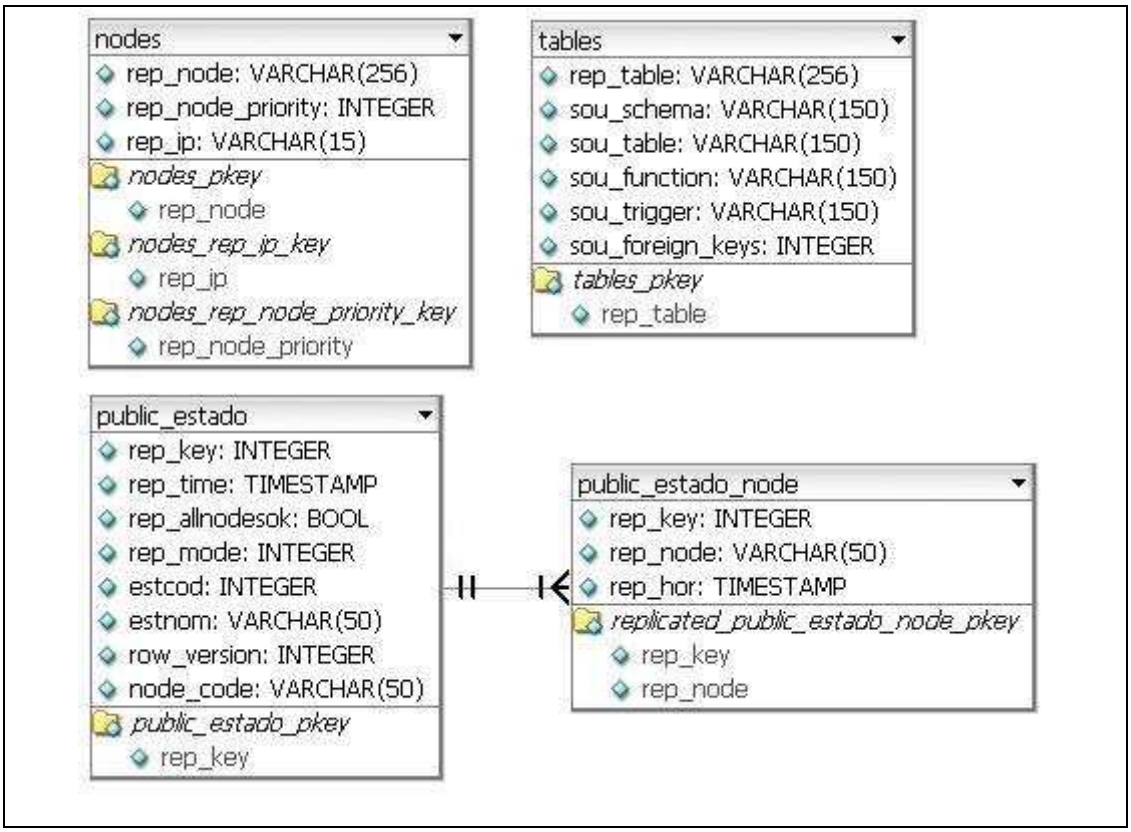

Figura 3 – MER das tabelas de armazenamento e controle

3.3 IMPLEMENTAÇÃO

A seguir são abordadas ferramentas e técnicas utilizadas para o desenvolvimento do aplicativo.

Para a implementação foi utilizado o NetBeans da *SUN Microsystems* na sua versão 5.5 sem a necessidade de algum *plugin* extra. Esta ferramenta oferece várias facilidades para o desenvolvimento e por isso da sua escolha.

A primeira etapa do processo de desenvolvimento efetuado foi a leitura do arquivo de configurações, utilizando-se para tanto a API do Java. Foi utilizado para a leitura do arquivo a classe `java.io.BufferedReader` juntamente com a `java.io.FileReader`. Neste momento preocupou-se em executar esta leitura de modo que ela não se prendesse a alguma plataforma, como por exemplo o Windows ou Linux.

Na próxima etapa foi desenvolvida a classe `ConexaoBanco`, na qual ficou responsável pelo estabelecimento de comunicação entre a aplicativo e a base de dados. Uma particularidade encontrada em relação a conexão foi o estabelecimento da propriedade

`charSet` para `latin1`, sem ela ocorreram erros ortográficos de acentuação.

A classe `Cluster` exigiu uma grande demanda de tempo para a sua implementação, devido a sua complexidade e quantidade de métodos envolvidos.

A utilização de tabelas de sistema do SGBD PostgreSQL foi de vital importância para implementar alguns métodos desta classe, contudo o uso destas tabelas foi um fator de extrema dificuldade pois não existe uma documentação em abundância sobre elas, além de ter uma complexa estrutura.

O Quadro 7 exibe um exemplo de um comando SQL utilizado para a obtenção das informações estruturais de uma tabela. A ferramenta *front-end* escolhida foi o pgAdmin na sua versão 1.6.3.

```
SELECT
     attrs.attname, "Type", attrs.attnotnull
FROM
     (SELECT c.oid, n.nspname, c.relname FROM pg_catalog.pg_class c
      LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
      WHERE pg_catalog.pg_table_is_visible(c.oid)) rel
JOIN
      (SELECT a.attname, a.attrelid,
       pg_catalog.format_type(a.atttypid,a.atttypmod) as "Type",
       a.attnotnull, a.attnum FROM pg_catalog.pg_attribute a WHERE
       a.attnum > 0 AND NOT a.attisdropped) attrs
ON
   (attrs.attrelid = rel.oid )
WHERE
   relname = 'estado' and rel.nspname = 'public'
ORDER BY
   attrs.attnum";
```

Quadro 7 – Comando SQL utilizado para obtenção das informações estruturais de uma tabela

Os campos de controle são de grande valor para o controle dos registros que necessitem ou não ter sua replicação efetuada. No momento em que se prepara a base de dados para a replicação, estes campos são criados na estrutura de cada tabela a ser replicada. A verificação de sua existência é necessária e o Quadro 8 mostra o comando SQL que resulta um *boolean* confirmando a existência dos campos de controle de uma tabela.

```
SELECT
   CASE
     WHEN count(*) = 2 THEN TRUE
     ELSE FALSE
    END
FROM
   pg_class c,
   pg_attribute a,
   pg_type t,
   pg_namespace n
WHERE
   c.relname = 'estado'
   and c.relnamespace = n.oid
   and n.nspname = 'public'
   and a.attnum > 0
   and a.attrelid = c.oid
   and a.atttypid = t.oid
   and a.attname in ('row_version','node_code');
```

Quadro 8 – Comando SQL utilizado que resulta um *boolean* confirmando a existência dos campos de controle de uma tabela

Na classe `Cluster` também foi implementado a criação das funções e gatilhos que tem suma importância para o funcionamento da aplicação deste projeto. O estudo de como seria estruturado as mesmas demandou uma grande reflexão sobre seu funcionamento. A sua lógica deveria verificar quando o registro modificado deveria ser replicado ou não. Para isto foi utilizado o campo de propriedade do registro.

Verifica-se a origem do dado replicado, ela sendo do nodo de origem, o mesmo deve ser inserido na tabela de armazenamento para futura sincronia com os nodos destino. Antes de armazenar, contudo, se faz necessário fazer o incremento da versão do registro, para que ao ser replicado, verificar se existe conflito com o registro do nodo destino e possibilitar a resolução de conflitos. Quando verificado que a origem do registro é de um nodo externo, apenas é assumido que o registro pertence ao nodo e não se incrementa a versão do registro.

Estas funções e gatilhos não observam a modificação estrutural das tabelas, bem como também não tem nenhuma interação com as seqüências do banco de dados, visto que as seqüências não são replicadas. O Quadro 9 exibe um exemplo de uma função e gatilho de controle.

```
CREATE OR REPLACE FUNCTION replicate_estado()
  RETURNS "trigger" AS
$BODY$
DECLARE node bpchar;
BEGIN
     SELECT INTO node replicated.thisnode();
     IF (TG_OP = 'DELETE') THEN
         OLD.row_version = OLD.row_version + 1;
         IF (OLD.node_code = node) THEN
             INSERT INTO replicated.public_estado(rep_mode, estcod,
estnom, row_version, node_code) VALUES (substr(TG_OP,1,1),
OLD.estcod, OLD.estnom, OLD.row_version, OLD.node_code);
             RETURN OLD;
         END IF;
      END IF;
      IF (TG_OP <> 'DELETE') THEN
         IF (NEW.node_code = node) THEN
            NEW.row_version = NEW.row_version + 1;
            INSERT INTO replicated.public_estado(rep_mode, estcod,
estnom, row_version, node_code) VALUES (substr(TG_OP,1,1),
NEW.estcod, NEW.estnom, NEW.row_version, NEW.node_code);
            RETURN NEW;
         ELSE
            NEW.node_code = node;
            RETURN NEW;
         END IF;
       END IF;
       RETURN NULL;
END;
$BODY$ LANGUAGE 'plpgsql' VOLATILE;

CREATE TRIGGER replicate_estadotrigger
  BEFORE INSERT OR UPDATE OR DELETE
  ON estado
  FOR EACH ROW
  EXECUTE PROCEDURE replicate_estado();
```

Quadro 9 – Exemplo de criação de função e gatilho de controle

Outra funcionalidade importante contida na classe `Cluster` é a criação das tabelas de armazenamento dos dados que são replicados. Nesta situação se apurou a necessidade da adição de alguns campos de controle a tabela original, são eles:

a) chave primária seqüencial;

b) hora de inserção do registro;

c) nodos destinos todos atualizados;

d) operação realizada (inserção, atualização ou exclusão).

Estes campos são adicionados na tabela no momento da sua criação, ou seja, quando se prepara o banco de dados para ser replicado. O resto da estrutura da tabela se mantém a mesma, nomes e tipos de dados dos atributos. O Quadro 10 mostra o código-fonte responsável

pela criação da tabela de armazenamento.

```
//Nome do atributo, tipo e não nulo
query = "SELECT attrs.attname, \"Type\", attrs.attnotnull "
     + "FROM (SELECT c.oid, n.nspname, c.relname FROM pg_catalog.pg_class c "
     + "LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace "
     + "WHERE pg_catalog.pg_table_is_visible(c.oid)) rel "
     + "JOIN (SELECT a.attname, a.attrelid, "
     + "pg_catalog.format_type(a.atttypid,   a.atttypmod) as \"Type\", "
     + "a.attnotnull, a.attnum FROM pg_catalog.pg_attribute a "
     + "WHERE a.attnum > 0 AND NOT a.attisdropped) attrs "
     + "ON (attrs.attrelid = rel.oid ) "
     + "WHERE relname = '"+ table +"' and rel.nspname = '" + schema
     + "' ORDER BY attrs.attnum";
rs = st.executeQuery(query);
//Obter os campos e tipos da tabela e montar o SQL para criação da tabela a
//ser replicada
sql = "CREATE TABLE " + arquivo.getSchema_store() + "." + schema + "_"
   + table + "(";
sql += "rep_key serial PRIMARY KEY, rep_time timestamp DEFAULT now(), "
    + "rep_allnodesOK boolean DEFAULT FALSE, rep_mode char(1) ";
while (rs.next()) {
    //verificar os nomes dos campos e seus tipos
    campo = rs.getString(1);
    tipo = rs.getString(2);
    notnull = rs.getString(3);
    sql += "," + campo +" "+ tipo;
    if (notnull.equals("t"))
       sql += " NOT NULL";
}
rs = null;
sql += ")";
System.out.println("Tabela a ser criada: " + arquivo.getSchema_store() + "."
                   + schema + "_" + table);
st.execute(sql);
```

Quadro 10 – Código-fonte responsável pela criação da tabela de armazenamento

Além destes campos da tabela original, cria-se uma segunda tabela que contém a lista dos nodos destinos que já foram sincronizados. Assim, após o registro ter sido replicado e tenha sido recebida uma resposta do nodo destino que a replicação ocorreu com sucesso, esta informação de sucesso é armazenada nesta tabela para que este mesmo registro não seja novamente enviado para o mesmo nodo destino.

A partir deste momento, foi implementado o aplicativo responsável pela preparação da base de dados para ser replicada. A classe definida foi a `Initdbr` que executa as operações necessárias para estruturar a base de dados de maneira que o banco de dados desempenhe sua função de verificar quais os registros que necessitam ser replicados.

Juntamente foi desenvolvida a classe `Dropdbr` que faz o procedimento inverso, ou seja, remove todas as estruturas criadas na preparação da base de dados.

A implementação da comunicação entre o nodo de origem e o nodo de destino foi

desenvolvida utilizado o Java RMI, portanto, foi necessário uma interface e uma classe, a `Replicador` e a `ReplicadorImpl` respectivamente.

A interface `Replicador` define um único método chamado `recebeRegistro` que é implementado pela classe `ReplicadorImpl`. Este método é utilizado para recepção do registro a ser sincronizado, bem como a sua operação, inserção, atualização ou exclusão.

Para implementação dos serviços de recepção e envio dos registros, as classes `ReplicadorServer` e `ReplicadorClient` utilizaram-se da estrutura de comunicação entre objetos remotos baseada no Java RMI.

O aplicativo responsável pelo recebimento das informações instancia para cada nodo de origem, um objeto que fica incumbido de fornecer ao nodo de origem a estrutura para envio dos dados. Este ao receber os dados, inicia o processo de verificação de conflito, resolução de conflito e o armazenamento dos mesmos. O Quadro 11 mostra o código-fonte executado ao receber um registro a ser sincronizado.

```
//Caso a versão do registro seja maior que a versão na base,
armazená-la
if (registroVersao > resultado){
    //Se for inserção ou atualização
    if (!modo.equalsIgnoreCase("D")){
       if (resultado < 0){
          //Executar o comando insert
          System.out.println(insert);
          st2.execute(insert);
          ok = true;
        }else{
          //Executar o comando update
          System.out.println(update);
          st2.execute(update);
          ok = true;
        }
     }else{
        //Executar o comando delete
        System.out.println(delete);
        st2.execute(delete);
        ok = true;
      }
}
//Caso a versão do registro seja igual a versão da base, verificar a
prioridade do nodo
if (registroVersao == resultado){
   //pegar o valor da prioridade do nodo
   int origem = getPrioridadeNodo(arquivo,nodo);
   int destino = arquivo.getPriority();
   if (origem <= destino){
      if (!modo.equalsIgnoreCase("D")){
         System.out.println(update);
         st2.execute(update);
         ok = true;
       }else{
          System.out.println(delete);
          st2.execute(delete);
          ok = true;
        }}}
```

Quadro 11 – Código-fonte executado ao receber um registro a ser sincronizado

3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Para demonstrar a operacionalidade do aplicativo foi disponibilizada uma rede local com os endereços de rede 10.1.1.4/8, 10.1.1.7/8 e 10.1.1.10/8. Nestes endereços estão operando os seguintes sistemas operacionais respectivamente, Microsoft Windows XP *Professional*, Microsoft Windows 2003 *Server Enterprise* e Linux. Todos os nodos têm operando o PostgreSQL 8.0.3 e o *front-end* pgAdmin III na versão 1.6.3. A versão da

máquina virtual Java instalada é a JRE 1.5. Cada nodo executa os aplicativos servidor e cliente que são responsáveis pela replicação.

O primeiro passo é configurar a arquivo de parâmetros. Foi estabelecido que se possui três nodos chamados nodo_1, nodo_2, nodo_3 e cada um deles possui a prioridade conforme sua numeração. Também se definiu que o período entre as atualizações é de 10 segundos, além de o tipo de inicialização das tabelas será por *schema*. O Quadro 12 mostra um dos arquivos de configuração.

```
# set up the name of stored replicated tables and sequences
schema_store = replicated

# choose the type of replication (tables or schemas)
replication_type = schemas

# set the delay between the synchronization between the replicas (in seconds)
replication_time = 10

# set my global node name. Max 50 characters
node_code = node_1

#set the priority of this node
priority = 1

#set default row version (integer)
row_version = 1

# database connections
# JDBC default
dbdatabase=jdbc:postgresql://localhost:5432/tcc
dbuser=postgres
dbpassword=master

# enter the priority number, node_code, IP address
[nodes]
2,node_2,10.1.1.7
3,node_3,10.1.1.10

#tables to be replicated (respecting replication_type option)
[tables]
#public.estado
#public.cidade

#schemas to be replicated (respecting replication_type option)
[schemas]
public
```

Quadro 12 – Arquivo de configuração

Devidamente configurado, inicia-se a estrutura de dados com o aplicativo `initdbr`. Sua execução pode ser vista na Figura 5.

Figura 5 – Execução do initdbr

A Figura 6 demonstra o diagrama de execução da inicialização da base de dados.

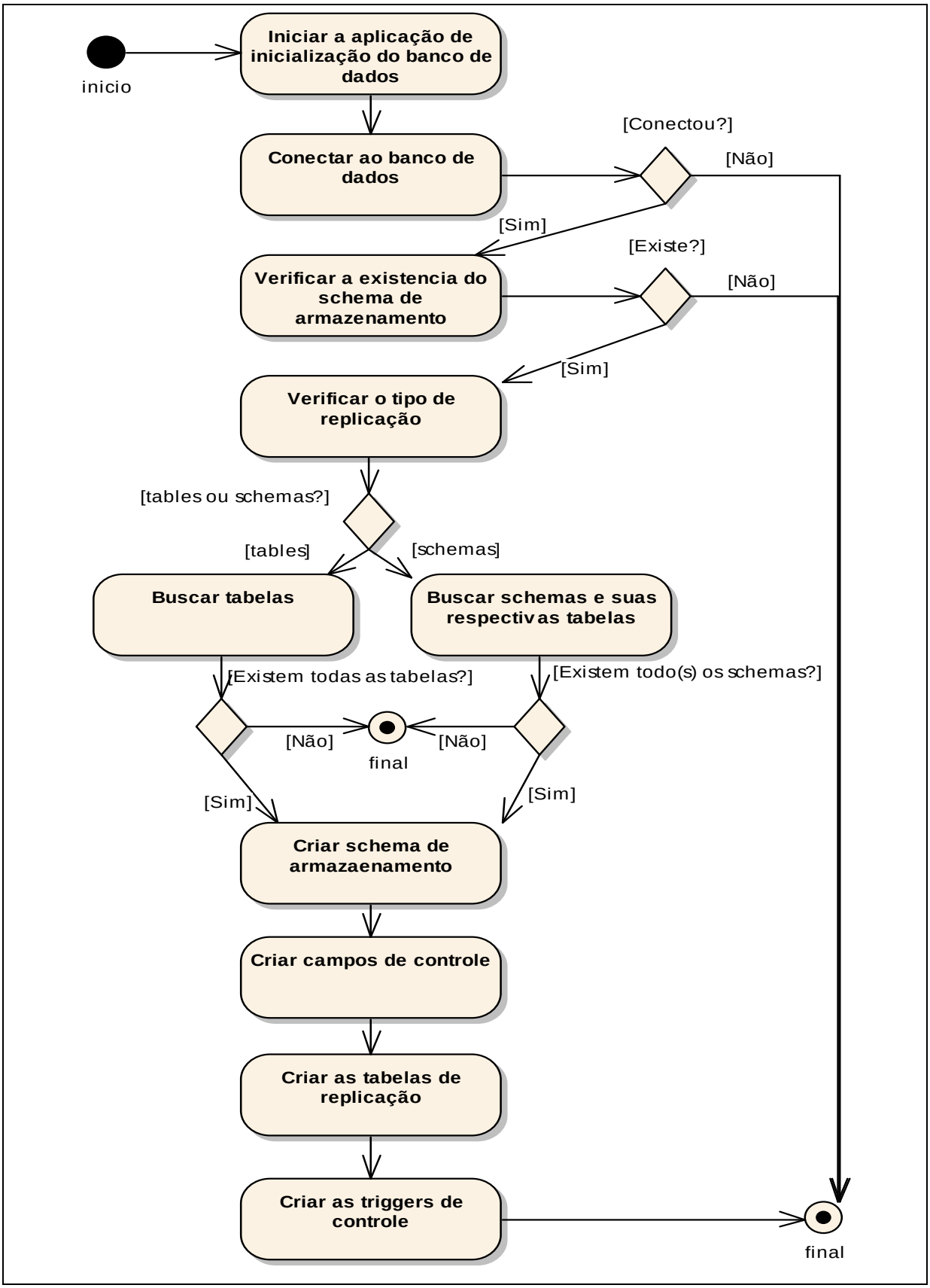

Figura 6 – Diagrama de execução da inicialização da base de dados

Posteriormente a sua execução, todos os registros alterados estão sendo monitorados pelo banco de dados e armazenados na estrutura de armazenamento. Para cada nodo se faz

necessário a inicialização do banco de dados.

A Figura 7 mostra como é feito o controle dos dados a serem replicados.

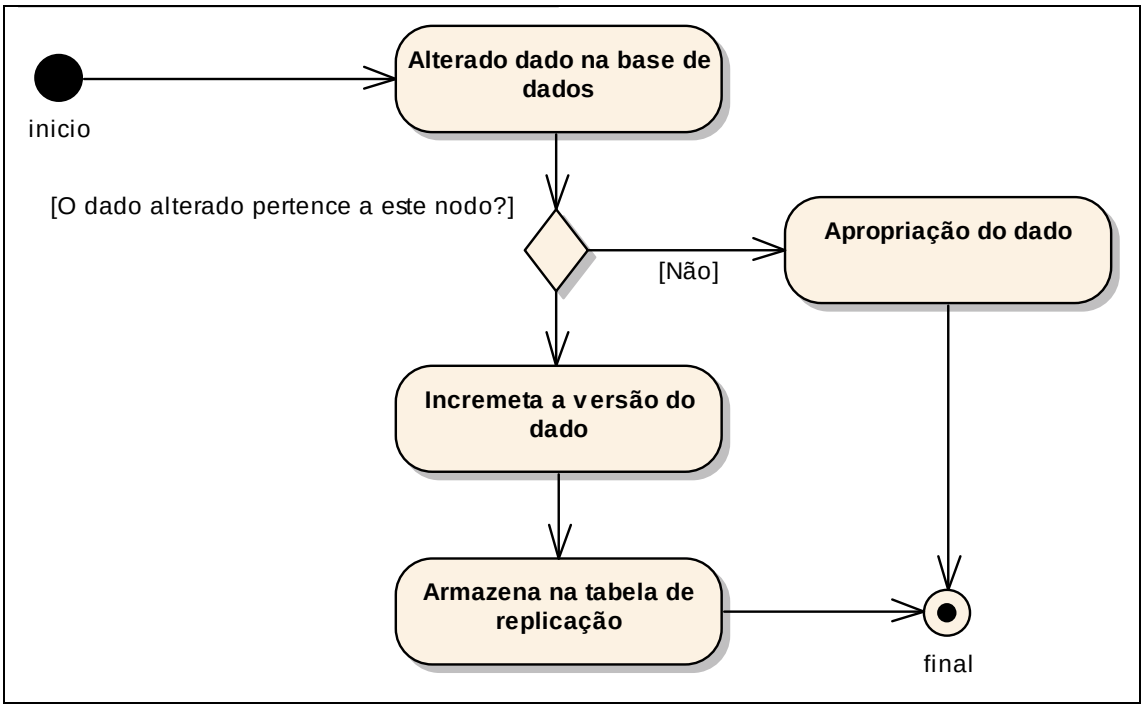

Figura 7 – Controle efetuado nos registros pelo banco de dados

Neste momento é necessário iniciar o serviço do rmiregistry e também o aplicativo servidor para que sejam disponibilizados os objetos remotos necessários a replicação de dados. O aplicativo servidor instancia para cada nodo de origem um objeto remoto. A Figura 8 demonstra sua execução.

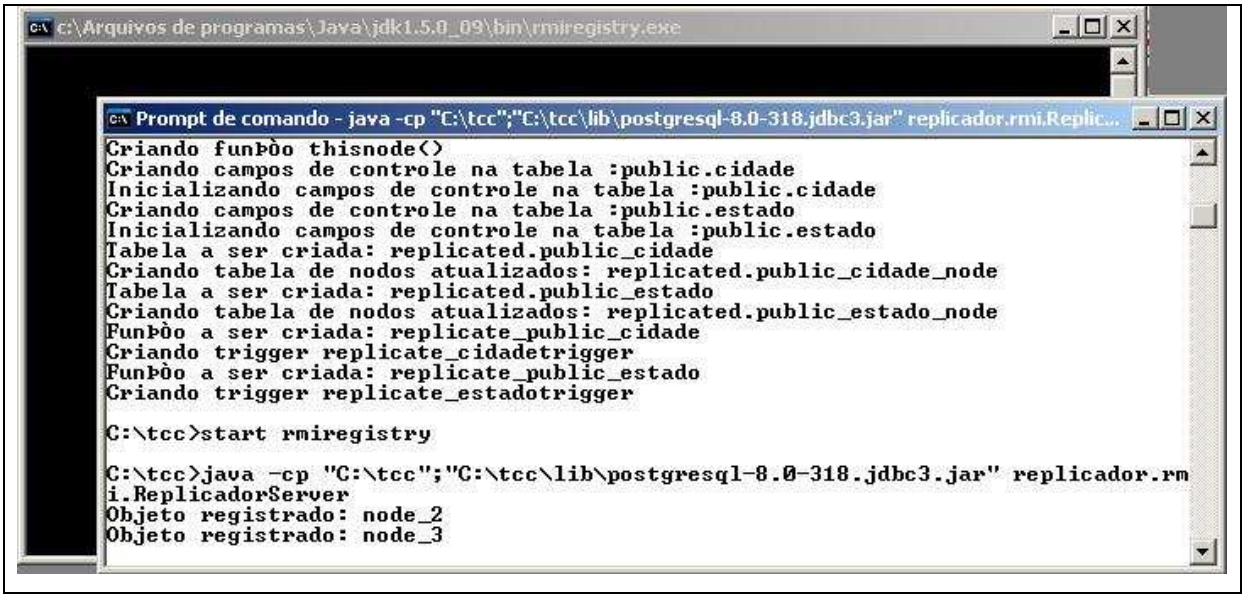

Figura 8 – Execução do aplicativo servidor

A partir deste instante, este nodo é capaz de receber conexões para efetuar a sincronização dos dados. Para cada nodo se faz necessário a execução destes dois aplicativos.

A última etapa é iniciar o aplicativo cliente no qual é o responsável pelo envio dos dados ao nodo de destino. A sua execução também é necessária em cada um dos nodos. O aplicativo cliente executa a sincronização a cada espaço de tempo definido no arquivo de configuração. A cada ciclo ele instancia o objeto remoto correspondente e envia os dados que necessitam ser sincronizados.

Para exibir sua funcionalidade, foram planejadas algumas situações específicas e exemplificados seus resultados.

3.4.1 Sincronia dos dados sem conflito e falha de comunicação entre os nodos

A primeira situação a ser exemplificada é a sincronia dos dados ocorrida sem nenhum conflito e nenhuma falha de comunicação entre os nodos de origem e de destino.

As bases de dados, todas elas, encontram-se sem nenhum dado dentro deles, apenas a estrutura inicializada. Com os aplicativos servidor e cliente operando, é inserido um registro dentro da tabela `public.estado`. Ao ocorrer a inserção nota-se que o nodo nomeado como `nodo_1` está com um registro armazenado e os nodos `nodo_2` e `nodo_3` estão com nenhum registro armazenado. O Quadro 13 exibe a tabela criada para replicação e também o primeiro dado inserido.

```
CREATE TABLE public.estado
(
  estcod char(2) PRIMARY KEY,
  estnom varchar(50)
);

INSERT INTO public.estado(estcod,estnom)
VALUES('SC','Santa Catarina');
```

Quadro 13 – Tabela criada para demonstrar a replicação

Ocorrendo o ciclo de tempo configurado de 10 segundos, inicia-se a sincronização de dados entre os nodos. O primeiro nodo faz a verificação da disponibilidade dos objetos remotos e faz sua instanciação. Obtida a referência do objeto remoto é iniciada a verificação nas tabelas de armazenamento se existe alguma informação a ser enviada a determinado nodo. Nesta situação é encontrado um registro e este ainda não foi sincronizado com nenhum dos nodos de destino.

O dado a ser enviado então é preparado dentro de uma estrutura e é enviado ao nodo destino. Ao chegar ao nodo destino, verifica-se qual a operação a ser executada, se é uma inserção, atualização ou exclusão. Essa informação de qual é a operação executada no nodo

de origem é enviada juntamente com o resto do registro. Obtida qual a operação a ser realizada no registro, busca-se a chave primária da tabela de destino e estrutura-se o comando SQL a ser executado na base de dados de destino. O comando SQL a ser utilizado é definido conforme a operação inserção, atualização ou exclusão.

Verifica-se agora qual é a versão do registro na base de destino. Como se trata de um novo registro, não se obtém esta informação, e assim sendo não se faz necessário resolver conflito, pois o mesmo não houve e então se executa a instrução SQL de inserção dentro da base de dados.

Estes procedimentos ocorrem da mesma maneira nos dois nodos de destino. Ao ser encerrado o armazenamento do registro, é retornado ao nodo de destino sucesso na sincronização e esta informação é armazenada na tabela de armazenamento na qual controla quais os nodos que foram atualizados para aquele registro. Assim no próximo ciclo este registro não é enviado aos nodos que se encontram nesta tabela.

3.4.2 Sincronia dos dados após a ocorrência de alguma falha de comunicação

Outra situação para se demonstrar é a sincronização dos dados após alguma falha de comunicação. Quando ocorre a falha de comunicação entre o primeiro nodo e o segundo. Neste caso as bases de dados somente ficarão sincronizadas depois que for possível estabelecer a conexão com o segundo dono e conseqüentemente a segunda base de dados permanecerá com os dados desatualizados perante os outros nodos.

Ao inserir um novo registro na tabela de estados no `nodo_1`, o mesmo é armazenado na tabela de armazenamento, de onde irá ser sincronizado. No momento que é efetuada a tentativa de obter a referência do objeto remoto de um nodo que não está disponível, o mesmo é abandonado, e efetua-se a replicação para o `nodo_3`, e armazena que este nodo está atualizado.

Nesta situação, os nodos `nodo_1` e `nodo_3` se encontram sincronizados, enquanto o `nodo_2` está fora de sincronia pela falha de comunicação encontrada entre o nodo de origem e o nodo destino.

No próximo ciclo de tempo, a conexão se restabelece e obteve-se a referencia do segundo nodo, único nodo não sincronizado. Neste momento, o `nodo_1` não tentará enviar o registro para o `nodo_3`, visto que a sincronização já havia ocorrido no ciclo anterior, e

sincroniza-se o nodo_2 inserindo-se o registro na sua base de dados.

Verifica-se que nesta segunda situação houve uma falha de comunicação em um determinado ciclo de sincronização e depois de restabelecida a comunicação entre os nodos de origem e destino, é efetuada a sincronia dos dados, sem haver conflito na replicação.

Uma terceira situação importante a ser verificada é quando ao serem sincronizados os dados, verifica-se a existência de conflito de dados entre os nodos.

3.4.3 Existência de conflito entre os dados a serem replicados

Um exemplo de cenário onde ocorre um conflito de informação é quando dentro de mesmo ciclo de tempo, inserem-se ou atualizam-se um registro com a mesma chave primária em duas bases de dados, por exemplo entre nodo_1 e nodo_2.

Para que se resolva este conflito, primeiramente é necessário que ocorra o ciclo de tempo e que estes registros com a mesma versão de linha sejam sincronizados. Na sincronização estes nodos enviarão para o nodo de destino o dado que contém a mesma chave primária e a mesma versão de linha.

Ao ser detectado o conflito[5], se faz necessário verificar qual dos nodos possuem maior prioridade na manutenção dos registros. A maior escala de prioridade é a de valor 1, portanto, quem possuir a maior prioridade obterá se registro armazenado nos nodos.

Exemplificando esta situação, entre os nodos nodo_1 e nodo_2 e considerando que os nodos tenham respectivamente as prioridades 1 e 2, caso necessite-se resolver um conflito, o nodo_1 obtém uma maior prioridade comparando com o nodo_2, e terão seu dado armazenado em ambas as bases de dados.

Verifica-se também que quanto maior o número de nodos, maiores são as probabilidades de acontecerem conflitos quando forem sincronizados os dados. Assim são geradas muitas situações a serem explanada neste capítulo, contudo todos os conflitos existentes obedecem a regra de prioridade do nodo.

[5] O conflito é detectado quando dois registros de origens diferentes têm a mesma versão de linha.

3.5 RESULTADOS E DISCUSSÃO

Durante a realização do trabalho ocorreram mudanças na especificação, para que se fosse possível sua conclusão. Foi constatado que na especificação original não seria possível atingir o objetivo determinado.

Apesar disso o trabalho atendeu os requisitos especificados, onde a função de um replicador assíncrono desenvolvido em Java para o SGBD PostgreSQL foi atendido. Baseando-se nos requisitos apresentados demonstra-se um comparativo entre os requisitos apresentados e os resultados obtidos.

a) possibilitar a replicação entre diversos sistemas operacionais: os aplicativos desenvolvidos são suportados pelos sistemas operacionais que suportem a máquina virtual Java, entre eles o sistemas operacionais da Microsoft e o sistemas operacionais Linux;
b) replicar conjunto de dados de aplicações: a replicação assíncrona efetua a sincronia dos dados de uma base dados. Alterações do metadados, seqüências e funções, por exemplo, não são replicadas, conferindo esta responsabilidade ao usuário do aplicativo;
c) replicar dados após restabelecer a comunicação: a sincronia dos dados é possível ser efetuada após o restabelecimento da comunicação entre os nodos, mantendo a consistência de informações entre as bases de dados.
d) permitir configuração através de arquivo: disponibilizou-se na configuração do replicador, através de um arquivo texto, opções de informar quais as tabelas que seriam disponibilizadas para replicação, bem como o valor em segundos de cada ciclo efetuado para a sincronização dos dados;
e) resolver conflitos na replicação: para solucionar problemas de conflitos entre as bases de dados, para permitir a consistência de dados, utilizou-se a combinação de duas técnicas, o versionamento de linhas e a prioridade de nodos.

A implementação foi executada utilizando somente a linguagem de programação Java e seus recursos. A principal ferramenta de apoio utilizada também atendeu os requisitos estipulados por ser *freeware*.

Existem restrições quanto a sua configuração e uso. A ferramenta não permite que dois nodos possuam a mesma prioridade. No caso de houver nodos com a mesma prioridade, ocorrerão inconsistências nos dados armazenados nas bases de dados. Além disso não é

permitido que hajam nodos com o mesmo nome, também com o risco de gerar inconsistências de dados entre as bases de dados.

Os testes realizados foram em entre três nodo, sendo dois deles com sistemas operacionais pertencentes a Microsoft e o terceiro sistema operacional foi o Linux. Nos testes realizados, não se preocupou com o desempenho nem com o uso de memória. Focou-se na funcionalidade do aplicativo e não na velocidade de processamento, visto que os testes ocorreram em ambientes emulados, com a utilização do utilitário VMWare. Um teste de maior porte foi executado utilizando cinco nodos e foram obtidos os resultados esperados na base de testes planejados.

O desenvolvimento baseou-se sempre na JDK 1.5 do Java. Sua utilização foi escolhida por ser a mais atualizada no momento do desenvolvimento deste trabalho e não ocorreram maiores problemas com o seu uso ou sintaxe, visto que o acadêmico do presente trabalho não possui uma ampla experiência nesta linguagem de programação.

A ferramenta desenvolvida neste projeto efetua a replicação assíncrona e tolerante a falha de comunicação, o que a diferencia de outros projetos tais como o Slony, Postgres-R onde o principal foco é a replicação síncrona.

O Slony tem como diferenciais a este trabalho a não dependência de versão do SGBD PostgreSQL no qual está operando. O aplicativo desenvolvido foi testado também nas versões 8.1 e 8.2, últimas versões disponíveis até o momento, não ofereceram nenhuma restrição ao uso nestas versões.

O Slony também possui como característica a existência de um nodo principal e nodos secundários, nos quais não são possíveis atualizações nos dados inseridos. Na ferramenta desenvolvida, não existe a figura de um nodo principal e nodos secundários, todos os nodos tem possibilidade de efetuar alterações na sua base de dados e esta alteração seja propagada aos outros nodos.

Uma característica do Slony é não detecção de falha de um nodo, e também não existe a eleição de um novo nodo principal onde esta ação deve ser humana. O replicador desenvolvido neste trabalho não necessita de um algoritmo de eleição, pois não há a existência de um nodo principal dentro do *cluster* de banco de dados.

Um ponto fundamental que diferencia os dois projetos é necessidade de uma confiabilidade da comunicação entre os nodos por parte do Slony. O Postgres-R apresenta esta mesma característica.

O Postgres-R tem como característica a replicação síncrona, além de ter sido projetado para a versão 6.4.2 do PostgreSQL. A sua execução é focada para redes locais, onde existe

maior confiabilidade de disponibilidade dos nodos. Sua operacionalidade é exclusivamente sobre ambiente Linux. Um ponto favorável a esta ferramenta é a otimização do processo de replicação, pois ela conta com o apoio de cópias *shadow* que se responsabilizam pela replicação ao outros nodos, ou seja, a replicação efetuada pelo nodo principal é executada uma vez para esta cópia, e ela se encarrega na distribuição as outras cópias.

Outra ferramenta de replicação utilizada para PostgreSQL é o ERServer que tem por particularidade ser um replicador assíncrono, porém, nela existe a necessidade de haver um nodo principal, como no Slony. Outra característica deste replicador é a não possibilidade de efetuar modificações nos nodos escravos, ou seja, estes nodos estão disponíveis apenas para leitura de dados. Comparando esta ferramenta com a desenvolvida, diferencia-se pela não existência de um nodo principal e a possibilidade de atualização em todos os nodos.

O quadro 14 abaixo demonstra as principais diferenças entre os trabalhos correlatos e a ferramenta desenvolvida.

COMPARATIVO ENTRE OS REPLICADORES DE DADOS ABORDADOS				
Característica	Slony	ERServer	Postgres-R	Ferramenta Desenvolvida
Replicação Assíncrona	X			X
Replicação Síncrona	X	X	X	
Tolerância a falta de comunicação				X
Existência de um nodo principal	X	X	X	
Nodos secundários atualizáveis				X[6]
Todos os nodos atualizáveis				X
Independência de versão do PostgreSQL	X			
Detecção de falhas				X
Necessidade de uma comunicação confiável	X	X	X	

Quadro 14 – Comparativo entre os replicadores de dados abordados

[6] Por não possuir o conceito de nodo primário, conseqüentemente não existem nodos secundários, porém todos os nodos são atualizáveis.

4 CONCLUSÕES

O trabalho desenvolvido contemplou todos os objetivos determinados no seu início, atendo todos os requisitos propostos.

A replicação de dados assíncrona e a consistência das informações em todas as réplicas, juntamente com a possibilidade de atualização dos nodos foi alcançada, o que é um grande diferencial comparando-se aos outros replicadores existentes que não possuem esta possibilidade.

A utilização do Java como linguagem de programação possibilitou que a ferramenta desenvolvida seja executada em sistemas operacionais que dêem suporte a ela, e assim permitindo que ela seja utilizada entre plataformas heterogêneas.

A comunicação não confiável não é um fator que impossibilite a sua utilização. Conseqüentemente com esta propriedade surge os conflitos entre os dados, já que todos os nodos são atualizáveis. A ferramenta implementou métodos que diagnosticam estes conflitos e utilizam as técnicas de versão de linha e de prioridades dos nodos.

As ferramentas utilizadas para o desenvolvimento da ferramenta supriram todas as necessidades exigidas para o mesmo. Dificuldades foram encontradas na obtenção das informações estruturais do banco de dados PostgreSQL, como definições de tabelas e chaves primárias e estrangeiras.

O trabalho também apresentou características ausentes em outras ferramentas como o Slony, ERServer, Postgres-R. Uma característica mais relevante que se difere das outras ferramentas é a possibilidade de atualização em todos os nodos.

4.1 EXTENSÕES

Como sugestão de extensão da ferramenta de replicação, o replicador de dado que hoje é específico para o SGBD PostgreSQL, poderia ser implementado para outros SGBDs.

Outra sugestão é a resolução da restrição de que a tabela para ser replicada necessita possuir chave primária.

Também seria interessante o desenvolvimento de outras técnicas de solução de conflitos de replicação, bem com a implementação da replicação síncrona dos dados.

REFERÊNCIAS BIBLIOGRÁFICAS

BEEHIVE. **BeehiveReplicatorAgent**. [S.l.], 2006. Disponível em: <http://www.beehive.com.br/replicator.html>. Acesso em: 03 jun. 2006.

BRAGA ,Carlos Henrique Maia. **Metamodelo para controle de estratégias assíncronas de replicação de dados**. 2001. 117 f. Dissertação para mestrado em Ciência da Computação.- Departamento de Ciências da Computação, Univesidade de São Paulo. São Paulo.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems**. 3rd ed. Londres: Pearson, 2001.

DATE, C. J. **Introdução a sistemas de bancos de dados**. 7. ed. Rio de Janeiro: Campus, 2000.

DEITAL, H. M.; DEITEL, P.J. **Java como programar**. 3. ed. Porto Alegre: Bookman, 2001.

DUARTE, E. M. **Replicação de dados no MySQL**. [S.l.], 2006. Disponível em: <http://www.sqlmagazine.com.br/Colunistas/eber/11_replicao.asp>. Acesso em: 03 jun. 2006.

ELMASRI, Ramez. **Sistemas de banco de dados**. São Paulo: Pearson, 2005.

FANDERUFF, D. **Oracle 8i**: utilizando SQL Plus e PL/SQL. São Paulo: Markron Books, 2000.

GBORG DEVELOPMENT TEAM. **ERServer**. [S.l.], 2006a. Disponível em: <http://gborg.postgresql.org/project/erserver>. Acesso em: 23 mar. 2006.

_____. **Slony-I**: a replication system for PostgreSQL. [S.l.], 2006b. Disponível em: <http://gborg.postgresql.org/project/slony1>. Acesso em: 23 mar. 2006.

_____. **PostgreSQL replication.** [S.l.], 2006c. Disponível em: <http://gborg.postgresql.org/project/pgreplication/projdisplay.php>. Acesso em: 06 jun. 2006.

LORÊDO H. Q.; FERREIRA L. N.; ASSIS G. T. **Replicação assíncrona entre bancos de dados heterogêneos**: uma abordagem prática. In: Congresso brasileiro de computação, IV., 2004, Itajaí.

PEREIRA NETO, A. **PostgreSQL**: técnicas avançadas versões open source 7.x. São Paulo: Erica, 2003.

POSTGRESQL. **Triggers**. [S.l.], [2005]. Disponível em: <http://www.postgresql.org/docs/8.0/interactive/triggers.html >. Acesso em: 22 mar. 2006.

POSTGRESQL BR. **Introdução**. [S.l.], 2003. Disponível em: <https://wiki.postgresql.org.br/wiki/Introdução>. Acesso em: 22 mar. 2006.

SOUZA JUNIOR, A. C. **Software para replicação de objetos entre duas instâncias de um SGBD Oracle**. 2003. 53 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SPARX SYSTEMS. **Enterprise Architect.** [S.l.], 2007. Disponível em <http://www.sparxsystems.com/products/ea.html>. Acesso em: 06 abr. 2007.

SUN DEVELOPER NETWORK. **Remote Method Invocation** Home. [S.l.], 2007. Disponível em: <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp> . Acesso em 04 jun. 2007.

TANENBAUM, A. S. **Distributed operating systems**. New York: Prentice Hall International Inc., 1995.

_____. **Distributed systems:** principles and paradigms. Amsterdam: Prentice Hall, 2002.

APÊNDICE A – Metadados criado para replicação da tabela de estados

```
--Criação do Schema de armazenamento
CREATE SCHEMA replicated
  AUTHORIZATION postgres;

--Criação da função que retorna o nome do nodo
CREATE OR REPLACE FUNCTION replicated.thisnode()
  RETURNS text AS
$BODY$
BEGIN
   RETURN 'node_1';
END;
$BODY$ LANGUAGE 'plpgsql' VOLATILE;

--Criação dos campos de controle na tabela de origem
ALTER TABLE public.estados
ADD row_version integer DEFAULT 1,
ADD node_code character varying(50) DEFAULT
    replicated.thisnode();

--Função que insere os dados dentro da tabela de armazenamento
CREATE OR REPLACE FUNCTION replicate_estado() RETURNS "trigger" AS $BODY$
DECLARE
   node bpchar;
BEGIN
   SELECT INTO node replicated.thisnode();
   IF (TG_OP = 'DELETE') THEN
      OLD.row_version = OLD.row_version + 1;
      IF (OLD.node_code = node) THEN
      INSERT INTO replicated.public_estado(rep_mode, estcod, estnom,
                  row_version, node_code)
                  VALUES
                  (substr(TG_OP,1,1),OLD.estcod, OLD.estnom,
                   OLD.row_version,OLD.node_code);
       RETURN OLD;
       END IF;
   END IF;
   IF (TG_OP <> 'DELETE') THEN
      IF (NEW.node_code = node) THEN
         NEW.row_version = NEW.row_version + 1;
         INSERT INTO replicated.public_estado(rep_mode, estcod,
                     estnom, row_version, node_code)
                     VALUES
                     (substr(TG_OP,1,1), NEW.estcod, NEW.estnom,
                      NEW.row_version, NEW.node_code);
         RETURN NEW;
      ELSE
         NEW.node_code = node;
         RETURN NEW;
      END IF;
    END IF;
    RETURN NULL;
END; $BODY$ LANGUAGE 'plpgsql' VOLATILE;

--Criação da trigger de controle
CREATE TRIGGER replicate_estadotrigger
  BEFORE INSERT OR UPDATE OR DELETE
```

```
  ON estado
  FOR EACH ROW
  EXECUTE PROCEDURE replicate_estado();

--Estrutura da tabela de armazenamento
CREATE TABLE replicated.public_cidade
(
  rep_key serial PRIMARY KEY,
  rep_time timestamp without time zone DEFAULT now(),
  rep_allnodesok boolean DEFAULT false,
  rep_mode character(1),
  estcod character(2) NOT NULL,
  estnom character varying(30) NOT NULL,
  row_version integer,
  node_code character varying(50)
);

--Estrutura da tabela que armazwnas os nodos atualizados
CREATE TABLE replicated.public_estado_node
(
  rep_key integer NOT NULL,
  rep_node character varying(50) NOT NULL,
  rep_hor timestamp without time zone DEFAULT now(),
  CONSTRAINT replicated_public_estado_node_pkey PRIMARY KEY (rep_key,
rep_node),
  CONSTRAINT replicated_public_estado_node_fkey FOREIGN KEY (rep_key)
      REFERENCES replicated.public_estado (rep_key) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
);

--Tabelas que contém os nodos de destino e suas prioridades
CREATE TABLE replicated.nodes
(
  rep_node character varying(256) NOT NULL,
  rep_node_priority integer,
  rep_ip character varying(15),
  CONSTRAINT nodes_pkey PRIMARY KEY (rep_node),
  CONSTRAINT nodes_rep_ip_key UNIQUE (rep_ip),
  CONSTRAINT nodes_rep_node_priority_key UNIQUE (rep_node_priority)
);

--Tabela que contém a relação de tabela de origem e de armazenamento

CREATE TABLE replicated.tables
(
  rep_table character varying(256) NOT NULL,
  sou_schema character varying(150) NOT NULL,
  sou_table character varying(150) NOT NULL,
  sou_function character varying(150),
  sou_trigger character varying(150),
  sou_foreign_keys integer,
  CONSTRAINT tables_pkey PRIMARY KEY (rep_table)
)
```

Quadro 15 – Metadados criado para replicação da tabela de estados

APÊNDICE B – Métodos para efetivar a replicação de dados

```
package replicador.cluster;

import java.rmi.RemoteException;
import java.sql.*;
import java.util.ArrayList;
import java.util.HashMap;
import replicador.rmi.Replicador;
/**
 *
 * @author malcus
 */
public class Cluster{

    ...

    /** Método para efetivar a replicação de dados */
    public boolean armazenaReplicacao(String tabelaOrigem, HashMap
campos,String nodo) {
         Configuracao arquivo = new Configuracao();
         arquivo.carregaArquivo();
         String schema,table;
         Statement st = null;
         Statement st2 = null;
         ResultSet rs  = null;
         String tabela = getTabelasOrigem(arquivo,tabelaOrigem);
         boolean ok = false;
         //Conexão com o banco de dados
         ConexaoBanco conexao = new ConexaoBanco();
         if (!conexao.conecta(arquivo)){
             return ok;
         }
         Connection conn = conexao.getConn();
         conexao.iniciaTransacao();
         //pegar as chaves primárias da tabela
         ArrayList key = getPrimaryKey (arquivo,tabela);
         //pegar os campos e verificar os seus valores
         ArrayList campo = getCamposTabela(arquivo,tabela);

          //Verificar se existe o registro e pegar o row_version
         String query = "SELECT row_version FROM " +  tabela + " WHERE ";
         for (int i = 0; i < key.size();i++){
             query += (String)key.get(i) + " = \'" +
campos.get((String)key.get(i)) + "\'";
             //Se não for a última chave
             if (i + 1 < key.size())
                 query += " AND ";
         }
         System.out.println(query);
         //Executar e verificar o row_version
         try {
            Integer resultado = -1;
            String modo = (String)campos.get("rep_mode");
            st = conn.createStatement();
            st2 = conn.createStatement();
            rs = st.executeQuery(query);
            while (rs.next()){
```

```
                    resultado = rs.getInt(1);
                }
            //Comparar o resultado com o do registro transmitido
            Integer registroVersao =
Integer.parseInt((String)campos.get("row_version"));

            //Montar os comandos
            String insert,update,delete;
            //INSERT
            insert = "INSERT INTO " + tabela + "(";
            for (int i = 0;i < campo.size();i++){
                insert += (String)campo.get(i);
                if (i + 1 < campo.size())
                    insert += ", ";
            }
            insert += ") VALUES (";
            for (int i = 0;i < campo.size();i++){
                //pegar os valores
                String valor = (String)campos.get((String)campo.get(i));
                if (valor == null)
                    valor = "NULL";
                else
                    valor = "\'" + valor + "\'";
                insert += valor;
                if (i + 1 < campo.size())
                    insert += ", ";
            }
            insert += ")";
            //UPDATE
            update = "UPDATE " + tabela + " SET ";
            for (int i = 0;i < campo.size();i++){
                update += (String)campo.get(i) + " = ";
                String valor = (String)campos.get((String)campo.get(i));
                if (valor == null)
                    valor = "NULL";
                else
                    valor = "\'" + valor + "\'";
                update += valor;
                if (i + 1 < campo.size())
                    update += ", ";
            }
            update += " WHERE ";
            for (int i = 0;i < key.size();i++){
                update += (String)key.get(i) + " = ";
                String valor = (String)campos.get((String)campo.get(i));
                if (valor == null)
                    valor = "NULL";
                else
                    valor = "\'" + valor + "\'";
                update += valor;
                if (i + 1 < key.size())
                    update += " AND ";
            }
            //DELETE
            delete = "DELETE FROM " + tabela + " WHERE ";
            for (int i = 0;i < key.size();i++){
                delete += (String)key.get(i) + " = ";
                String valor = (String)campos.get((String)campo.get(i));
                if (valor == null)
                    valor = "NULL";
                else
```

```
                    valor = "\'" + valor + "\'";
                delete += valor;
                if (i + 1 < key.size())
                    delete += " AND ";
            }

            //System.out.println(insert);
            //System.out.println(update);
            //System.out.println(delete);

            //Caso a versão do registro seja maior que a versão na base,
armazená-la
            if (registroVersao > resultado){
               //Se for inserção ou atualização
               if (!modo.equalsIgnoreCase("D")){
                   if (resultado < 0){
                       //Executar o comando insert
                       System.out.println(insert);
                       st2.execute(insert);
                       ok = true;
                   }else{
                       //Executar o comando update
                       System.out.println(update);
                       st2.execute(update);
                       ok = true;
                   }
               }else{
                   //Executar o comando delete
                   System.out.println(delete);
                   st2.execute(delete);
                   ok = true;
               }
            }
            //Caso a versão do registro seja igual a versão da base,
verificar a prioridade do nodo
            if (registroVersao == resultado){
                //pegar o valor da prioridade do nodo
                int origem = getPrioridadeNodo(arquivo,nodo);
                //int destino =
getPrioridadeNodo(arquivo,arquivo.getNode_code());
                int destino = arquivo.getPriority();
                if (origem <= destino){
                    if (!modo.equalsIgnoreCase("D")){
                        System.out.println(update);
                        st2.execute(update);
                        ok = true;
                    }else{
                        System.out.println(delete);
                        st2.execute(delete);
                        ok = true;
                    }
                }else
                    ok =  true;
            }

            //Caso a versão do registro seja menor a versão da base
            if (registroVersao < resultado){
               conexao.finalizaTransacao("COMMIT");
               ok = true;
            }
```

```
        }catch (SQLException ex) {
            conexao.desconecta();
            ex.printStackTrace();
            //conexao.finalizaTransacao("ROLLBACK");
            ok = false;
        }finally{
            conexao.finalizaTransacao("COMMIT");
            //Desconectar do banco de dados
            conexao.desconecta();
        }
        //Desconectar do banco de dados
        conexao.desconecta();
        return ok;
    }
    ...
}
```

Quadro 16 – Métodos para efetivar a replicação de dados