UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE UM FRAMEWORK PARA REPLICAÇÃO DE DADOS ENTRE BANCOS HETEROGÊNEOS

JOÃO BATISTA GIANISINI JÚNIOR

BLUMENAU
2006

2006/2-18

JOÃO BATISTA GIANISINI JÚNIOR

DESENVOLVIMENTO DE UM FRAMEWORK PARA REPLICAÇÃO DE DADOS ENTRE BANCOS HETEROGÊNEOS

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 - Orientador

BLUMENAU
2006

2006/2-18

DESENVOLVIMENTO DE UM FRAMEWORK PARA REPLICAÇÃO DE DADOS ENTRE BANCOS HETEROGÊNEOS

Por

JOÃO BATISTA GIANISINI JÚNIOR

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 – Mestrado, FURB
Membro:	Prof. Everaldo Artur Grahl, Mestrado – FURB
Membro:	Prof. Adilson Vahldick, Especialista – FURB

Blumenau, dezembro de 2006

Dedico este trabalho a toda minha família, principalmente meus pais, colegas de classe e ao meu orientador que acreditou no potencial do trabalho.

AGRADECIMENTOS

À Deus, que através da fé nos mantém fortes para alcançar nossos objetivos.

Aos meus pais, João e Mônica, que sempre estavam presentes me incentivando e oferecendo o máximo possível para superar esta e muitas outras caminhadas da minha vida.

A minha noiva Juliana Heiden por ter paciência nos momentos mais difíceis no desenvolvimento deste trabalho.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Alexander Roberto Valdameri, por ter acreditado na conclusão deste trabalho.

RESUMO

Neste trabalho são apresentadas a especificação e implementação de um framework desenvolvido na linguagem Java que tem como objetivo fornecer uma infra-estrutura para replicação de dados entre bancos heterogêneos. É utilizado o *Java Message Service* como mecanismo para envio e recebimento de mensagens e Hibernate que serve como um *middleware* para acesso a banco de dados. A ferramenta garante qualidade de serviço mesmo que não haja sincronismo na comunicação entre os *sites* e também fornece um nível considerável de abstração das bases de dados envolvidas na replicação.

Palavras-chave: Replicação de dados. Hibernate. *Java Message Service*.

ABSTRACT

The work the specification and implementation of a framework developed in the language Java are presented that has as objective to supply an infra structure response of data between heterogeneous database. The Java Message Service is used as mechanism for sending and act of receiving of messages and Hibernate that serves as one middleware for access the database. The tool guarantees quality of same service that does not have synchronism in the communication between the sites and also it supplies a considerable level of abstraction of the involved databases in the response.

Key-words: Response of data. Hibernate. Java Message Service.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Muitas empresas apresentam sua estrutura física descentralizada, isto é, unidades de produção, vendas, entre outras, geograficamente distantes entre si. Com isto, torna-se complexa a gerência das informações da empresa, principalmente nos aspectos de disponibilidade e compartilhamentos de dados.

O compartilhamento dos dados é um fator muito importante para que uma empresa possa analisar e tomar decisões. Portanto, a falta de integração que não viabiliza uma visão concentrada das informações, pode prejudicar as necessidades administrativas de uma empresa. No entanto, este foi um dos grandes motivos para o crescimento e inovações tecnológicas na área de banco de dados distribuídos (CASANOVA; MOURA,1985, p. 1), tendo como característica deste trabalho, o recurso de replicação de dados.

Para que possa ocorrer este compartilhamento de informações é necessário um mecanismo que replica os dados mantidos em um banco de dados local para outro remoto, tendo assim uma cópia dos dados atualizada na base em que foi replicada. Este conceito pode definir um cenário em que uma empresa necessita manter os dados replicados em cada filial. Neste caso as aplicações desenvolvidas devem realizar as consultas localmente, mas os comandos *Data Manipulation Language* (DML) (*insert*, *update* e *delete*) sobre as tabelas replicadas, devem ser obrigatoriamente replicados para as bases associadas (JOHANN; KROTH, 2005, p. 1).

Existem algumas características que podem dificultar o processo de replicação, como por exemplo, o caso em que algum dos participantes esteja *off-line* na rede, podendo ocorrer falhas nos processos de envio e recebimento de dados. Para resolver este problema foi necessário um recurso que garanta que a informação possa ser enviada ou recebida assim que seja restabelecida a comunicação, sendo caracterizado como comunicação assíncrona. A heterogeneidade nos Sistemas Gerenciadores de Bancos de Dados (SGBD) envolvidos é outro fator que pode dificultar a replicação.

Para implementar a heterogeneidade, uma das alternativas que pode ser utilizada é o Hibernate que conceitualmente, é um *framework*[1] de acesso a banco de dados, construído na linguagem Java. O mesmo realiza o mapeamento das tabelas do modelo relacional para classes da linguagem servindo como uma ponte entre a aplicação e o SGBD.

---

[1] É uma estrutura de suporte definida em que um outro projeto do software pode ser organizado e desenvolvido.

Como o Hibernate não é capaz de replicar as informações de um SGBD para outro, uma das alternativas que possibilitou a replicação foi implementar uma integração com algum mecanismo de troca de mensagens como, por exemplo, o *Java Message Service* (JMS).

Segundo Waeny e Numazaki (2004), JMS é uma *Application Program Interface* (API) que permite aplicações criar, enviar e ler mensagens. Trabalha com a estrutura de filas de mensagens que é gerenciada por um servidor chamado *Provider* JMS. O *Provider* tem a funcionalidade de gerenciar as mensagens enviadas pelos clientes na fila, disponibilizando as mensagens para consumo ou enviar diretamente para outros clientes.

Integrando as tecnologias supracitadas, é possível criar uma ferramenta que possibilita a replicação de dados. Esta ferramenta pode fornecer recursos como qualidade de serviço mesmo utilizando a comunicação de forma assíncrona e também um certo nível de abstração dos SGBD`s participantes da replicação.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *framework* que servirá como uma camada adicional entre a aplicação e a base de dados, mais precisamente um *Middleware*, capaz de interceptar e replicar as transações de um banco de dados local para um banco de dados remoto.

Os objetivos específicos do trabalho são:

a) fornecer heterogeneidade dos SGBD`s participantes da replicação que são compatíveis com a tecnologia Hibernate;
b) fornecer uma estrutura que facilite a construção de um sistema que necessita replicar dados;
c) tornar configurável e transparente os processos de replicação 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 nas tecnologias e componentes utilizados no desenvolvimento do trabalho.

O terceiro capítulo tem como foco o desenvolvimento do *framework*, 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

Neste capítulo são abordados os temas replicação de dados, Hibernate, *Java Message Service* (JMS) e *Providers* JMS. Na última seção são descritos alguns trabalhos correlatos.

2.1 REPLICAÇÃO DE DADOS

Segundo Moscardini (2006, p. 35) o objetivo de replicação de dados é manter duas ou mais cópias dos dados em diferentes servidores. Pode-se replicar um conjunto de tabelas, ou mesmo aplicar filtros nas linhas e colunas que se deseja replicar, mas pode-se também replicar um banco de dados em sua totalidade. A replicação tem sido uma grande aliada das empresas, provendo melhor desempenho e maior disponibilidade através da distribuição organizada de um conjunto de dados.

2.1.1 Análise dos requisitos do negócio

Segundo Moscardini (2006, p. 35) antes de iniciar a configuração da replicação, o projetista deve definir para que e como será a replicação de dados, de modo que seu usuário tenha satisfação e confiabilidade no processo como um todo. Para manutenção das cópias é necessário levantar os requisitos para o funcionamento da replicação como latência, autonomia da réplica, conflitos de atualização, disponibilidade física (hardware e software), velocidade dos *links* da conexão e tipo de arquitetura.

Latência é o tempo permitido para que haja um sincronismo entre as réplicas. À medida que a base de dados original sofre alterações, existe certo tempo (latência) até que estas sejam propagadas para as cópias (replicas) (MOSCARDINI, 2006, p. 35).

Existem empresas em que as filiais precisam continuar operando normalmente mesmo que os *sites* estejam com problemas de comunicação com a matriz. Para que isso ocorra, a réplica deve possuir dados suficientes para operar de forma autônoma quando o acesso do servidor central estiver indisponível (MOSCARDINI, 2006, p. 35).

Quando existem diversas réplicas onde os dados podem ser alterados, o mesmo registro pode ser modificado em réplicas diferentes gerando uma condição de conflito onde uma alteração sobrepõe a outra. Neste contexto, o projetista deve levantar quais serão os servidores e o meio de comunicação disponível (MOSCARDINI, 2006, p. 35).

A replicação implica em informações sendo enviadas na rede. Por essa razão, é importante estimar o volume de dados a serem replicados e verificar se o *link* disponível é suficiente para que se evite uma sobrecarga do sistema, impedindo ou atrasando o processo de sincronização (MOSCARDINI, 2006, p.35).

2.1.2 Tipos de arquiteturas para replicação

Valduriez (1999 apud LORÊDO; FERREIRA; ASSIS, 2004, p. 3) sugere algumas alternativas para alocação de dados entre Banco de Dados Distribuído (BDD), não replicados, totalmente replicados e parcialmente replicados. Quando se opta por bases de dados não replicadas, os dados são fragmentados entre os *sites* envolvidos, apresentando na rede somente uma cópia de cada fragmento. Em base de dados totalmente replicados, todos os *sites* possuem todos os dados. No modelo de replicação parcial, cada fragmento pode estar alocado em um ou mais *sites*. A replicação pode ser empregada com o objetivo de aumentar o desempenho e disponibilidade das aplicações.

Buretta (1997 apud LORÊDO; FERREIRA; ASSIS, 2004, p. 3) classifica como síncrona ou assíncrona o tempo de atualização de um dado e sua distribuição entre as demais réplicas. No modelo síncrono todas as replicações devem ocorrer simultaneamente à atualização dos dados. Já no modelo assíncrono, existe um intervalo entre a atualização e a replicação, podendo oferecer uma solução no cenário em que alguns *sites* possam estar sem conexão por algum tempo.

As replicações assíncronas devem ser implementadas, conforme Buretta (1997 apud LORÊDO; FERREIRA; ASSIS, 2004, p. 3), a partir de atualizações completas ou incrementais[2] ou por propagação delta de eventos[3], sendo o modelo mais adequado para situações onde é necessário preservar os eventos transacionais.

[2] Apenas o que foi alterado desde a última replicação realizada é enviado para as réplicas.
[3] Preserva os eventos nas réplicas, envolvendo apenas uma transação ou até mesmo um conjunto de transações.

Em aplicações em que se deseja utilizar replicação de dados, segundo Buretta (1997 apud LORÊDO; FERREIRA; ASSIS, 2004, p. 3), deve ser definido quem será o detentor do direito de propriedade, classificando como mestre-escravo ou atualizável em qualquer lugar.

No modelo mestre-escravo existe uma base de dados alocada de forma centralizada chamado mestre. Somente esta base receberá atualizações, as replicadas ou escravos servirão para consultas e serão atualizadas de forma assíncrona.

No modelo atualizável em qualquer lugar, a base central inclusive as replicadas, podem receber atualização nas formas síncrona ou assíncrona.

2.2 HIBERNATE

Segundo Bauer e King (2005, p. 4) pode ser dito que Hibernate é um projeto ambicioso que propõe uma solução em Java para gerenciamento de dados persistentes. É considerado também como uma camada entre o aplicativo e um banco de dados relacional. Sendo assim, deixa o desenvolvedor livre para concentrar-se nos problemas específicos de negócio.

2.2.1 Mapeamento objeto relacional (ORM)

A principal função do Hibernate é realizar *Object Relacional Mapping* (ORM), sendo o nome dado a soluções em que é automatizado o problema de incompatibilidade entre aplicativos e SGBDs (BAUER; KING, 2005, p. 31).

Uma solução ORM consiste em quatro partes fundamentais. O relacionamento entre cada parte do ORM com o Hibernate é descrito no Quadro 1:

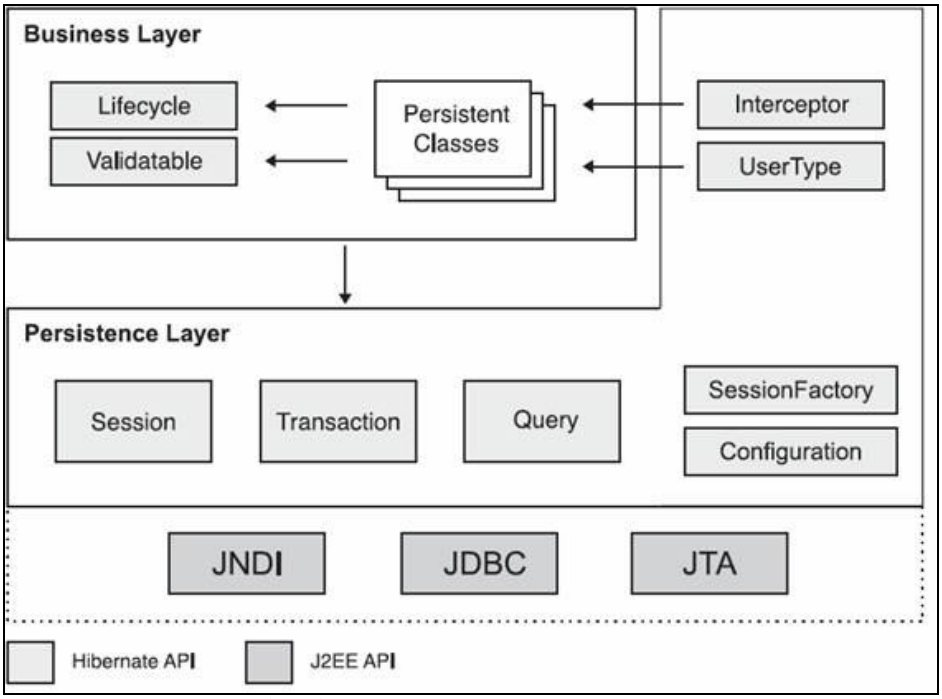
ORM	**HIBERNATE**
Uma API que possui operações *Create, Read Update e Delete* (CRUD) básicas em objetos de classes persistentes.	Hibernate disponibiliza interfaces que realizam as operações CRUD para qualquer objeto mapeado.
Uma linguagem ou API para especificar consultas que recorrem a classes e atributos de classes.	Hibernate possui uma linguagem própria chamada *Hibernate Query Language* (HQL), utilizada para recuperar os dados persistidos em base, onde sempre é referenciada a classe e não tabela do banco.
Uma facilidade para especificar mapeamento de metadados.	Hibernate utiliza arquivos *Extensible Markup Language* (XML) que descrevem a associação entre a classe e a tabela do banco.
Uma técnica para implementação de ORM para interagir como objetos transacionais a fim de executar verificação suja, irem buscar associações lentas, e outras funções de otimização.	Hibernate fornece interfaces que possuem métodos capazes de declarar os limites de uma transação de banco de dados e também é possível configurar o nível de isolamento das transações.

Fonte: Bauer e King (2005, p. 32).

Quadro 1 – Recursos fundamentais de ORM relacionando com o Hibernate

2.2.2 Arquitetura

Segundo Bauer e King (2005, p. 50) a arquitetura do Hibernate é formada basicamente por um conjunto de interfaces. A Figura 1 apresenta as interfaces com os papéis mais importantes nas camadas de negócio e persistência. A camada de negócio aparece acima da camada de persistência por atuar como uma cliente da camada de persistência. Note que algumas aplicações podem não ter a separação clara entre as camadas de negócio e de persistência.



Fonte: Bauer e King (2005, p. 51).

Figura 1 - Avaliação da arquitetura do Hibernate estendida em camadas

De acordo com Figura 1, as interfaces do Hibernate podem ser classificadas em:

a) Interfaces responsáveis por executar operações CRUD no banco de dados: `Session`, `Transaction` e `Query`;
b) Interface utilizada pela aplicação para configuração do Hibernate: `Configuration` e `SessionFactory`;
c) Interfaces responsáveis por realizar a interação entre os eventos do Hibernate e a aplicação: Interceptor, `Lifecycle` e `Validatable`;
d) Interfaces que permitem a extensão das funcionalidades de mapeamento do Hibernate: `UserType`, `CompositeUserType` e `IdentifierGenerator`.

Hibernate faz uso de várias outras APIs Java existentes, como *Java Database Connectivity* (JDBC), *Java Transaction API* (JTA) e serviço de nomes e diretórios Java chamado *Java Naming and Directory Interface* (JNDI).

Segundo Bauer e King (2005, p.51) JNDI e JTA permite que o Hibernate seja suportado pela maioria dos aplicativos servidores *web* da API *Java 2 Platform, Enterprise Edition* (J2EE).

O Banco de dados à ser utilizado pela aplicação é configurado diretamente em sua infra-estrutura, não apresentando maiores desafios para migrar a aplicação para outro SGBD

distinto, desde que suportado pelo Hibernate. O Quadro 2 apresenta as bases de dados suportadas pelo Hibernate.

DB2 7.1, 7.2, 8.1
HSQL DB
HypersonicSQL 1.61, 1.7.0, 1.7.2, 1.8
Microsoft SQL Server 2000
MySQL 3.23, 4.0, 4.1, 5.0
Oracle 8i, 9i, 10g
PostgreSQL 7.1.2, 7.2, 7.3, 7.4, 8.0, 8.1
SAP DB 7.3
Sybase 12.5 (JConnect 5.5)
Timesten 5.1
Apache Derby
HP NonStop SQL/MX 2.0 (requires Dialect from HP)
Firebird (1.5 with JayBird 1.01)
FrontBase
Informix
Ingres
Interbase (6.0.1)
Mckoi SQL
Pointbase Embedded (4.3)
Progress 9

Fonte: Hibernate (2003, p. 1).

Quadro 2– Bases suportadas pelo Hibernate

2.2.3 Mapeamento de classes persistentes

Segundo Bauer e King (2005, p.90) as classes persistentes de uma aplicação são aquelas que implementam as entidades domínio de negócio. O Hibernate trabalha associando cada tabela do banco de dados a um *Plain Old Java Object* (POJO). POJOs são objetos Java que seguem a estrutura de *JavaBeans* (construtor padrão sem argumentos, e com métodos para manipulação de seus atributos). No Quadro 3 é apresentado um exemplo de mapeamento no Hibernate, juntamente com seu POJO.

```
Arquivo XML que descreve o mapeamento com o POJO User
(User.hbm.xml)

<hibernate-mapping>
      <class name="User" table="Usuario">
            <id name="codUser" column="cod_usuario"
type="java.lang.Integer" unsaved-value="null">

                  <generator class="assigned"/>
            </id>
            <property
                  name="firstname"
                  type="java.lang.String"
                  column="ds_primeiro_nome"
                  length="100"
                  not-null="true"
            />
      </class>
</hibernate-mapping>

Arquivo fonte do POJO (User.java)

public class User{
    private Integer codUser;
    private String firstname;
    public User(){}
    public void setCodUser(Integer codUser){
        this.codUser = codUser;
    }
    public void setFirstName(String name){
        this.firstname = name;
    }
    public Integer getCodUser(){
        return this.codUser;
    }
    public String getFirstName(){
        Return this.firstname;
    }
}
```

Quadro 3 – Exemplo de mapeamento utilizando Hibernate

O Exemplo mapeia a tabela Usuário do banco de dados para a classe `User` do Java. É definido o atributo `codUser` como identificador da classe, que refere-se a *constraint primary-key* da tabela, que contempla a coluna `cod_usuario`. Também é definida uma propriedade simples, `firstname` que mapeia a coluna `ds_primeiro_nome`.

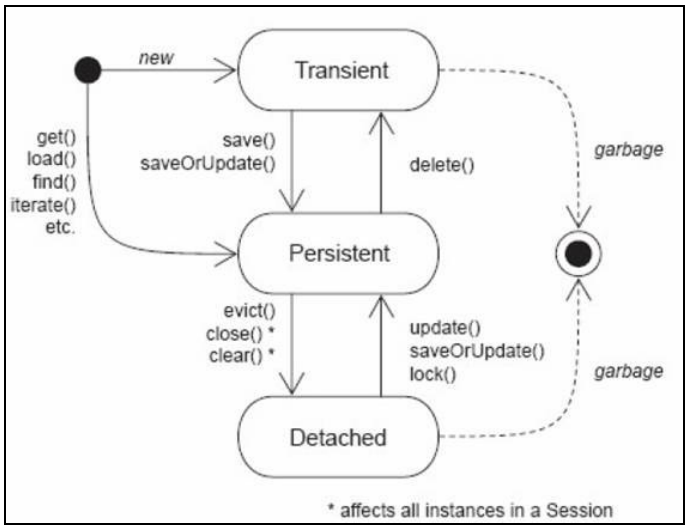
2.2.4 Estado e transições de objetos em aplicativos que utilizam Hibernate

Segundo Bauer e King (2005, p.152) sempre que uma aplicação precisa propagar o estado de um objeto que está alocado em memória para o banco de dados ou vice-versa, há a necessidade de que a aplicação possa interagir com uma camada intermediária entre a aplicação e base de dados, que forneça todos os recursos de persistência. Com o Hibernate, é feito invocando as interfaces do gerenciador de persistência. Interagindo com o mecanismo de persistência, é necessário que a aplicação conheça os estados do ciclo de vida de um objeto.

Em aplicações orientadas a objetos, a persistência permite que um objeto continue a existir mesmo após a destruição do processo que o criou. Na verdade, o que continua a existir é seu estado, já que pode ser armazenado em disco e então, no futuro, ser recriado em um novo objeto.

Segundo Bauer e King (2005, p.153) em uma aplicação nem todos os objetos precisam ser persistidos, pode haver objetos transientes que possuem um ciclo de vida limitado. Objetos transientes são aqueles que perdem a referência ao término do processo que o instanciou. Em relação às classes persistentes, nem todas as suas instâncias possuem necessariamente um estado persistente. Elas também podem ter um estado transiente ou destacados.

São definidos na Figura 2 os estados e transições de um objeto como também as chamadas para o gerenciador de persistência:



Fonte: Bauer e King (2005, p. 153).

Figura 2 – Avaliação da arquitetura do Hibernate estendida em camadas

Em um clico de vida, um objeto pode alterar seu estado de transiente para persistente seguindo para o estado destacado. Segundo Bauer e King (2005, p. 153) são descrito cada um destes estados:

a) objetos transientes: objetos instanciados que usam o operador `new` não são imediatamente persistentes. O estado dele é transiente, o que significa que eles não são associados a qualquer linha da tabela de banco de dados, e assim o estado dele é perdido a partir de momento em que perder a referência com qualquer outro objeto pelo aplicativo. Neste momento, efetivamente ocorre o término de seu ciclo de vida deste objeto;
b) objetos persistentes: Instâncias persistentes participam de transações, o estado delas é sincronizado com o banco de dados ao término da transação. Quando uma transação é efetiva, o estado mantido em memória é propagado para o banco de dados pela execução das instruções de SQL `insert, update e delete`;
c) objetos destacado: Quando uma transação termina, as instâncias persistentes associadas ao gerenciador de persistência ainda existem, porém perdem sua associação quando é fechada a sessão. Referimos estes objetos como destacados, indicando que seu estado não está mais sobre o gerenciador do Hibernate.

2.2.5 Hibernate Query Language (HQL)

Segundo Bauer e King (2005, p. 184) a linguagem de Consulta (HQL) é um dialeto orientado para objetos da linguagem de consulta relacional familiar *Structure Query Language* (SQL). HQL possui muitas semelhanças com *Object Data Management Group* (ODMG), *Object Query Language* (OQL) e *Entreprise Java Beans Query Language* (EJB-QL), mas ao contrário de OQL, ele está adaptado para uso com bancos de dados SQL e também muito mais poderoso e elegante que EJB-QL. HQL é fácil de aprender com conhecimentos básicos de SQL, mas não é uma linguagem de manipulação de dados como SQL. Só é usado como obtenção de objetos, não para atualizar, inserir ou excluir dados. A sincronização de estado do objeto é o trabalho do gerenciador de persistência e não do desenvolvedor.

A maior parte do tempo, somente é necessário apenas obter os objetos de uma classe particular e restringir através das propriedades dessa classe. Por exemplo, a consulta indicada no Quadro 4 obtém um usuário através do primeiro nome:

```
Query q = session.createQuery("from User u where
u.firstname = :fname");
q.setString("fname","Max");
List result = q.list();
```

Fonte: Bauer e King (2005, p. 185).

Quadro 4 – Exemplo de consulta a banco de dados utilizando HQL

Depois de preparar a consulta `q`, é ligado o valor do identificador a um parâmetro nomeado, `fname`. O resultado retorna como objetos `List` de `User`.

Segundo Bauer e King (2005, p.185) HQL é poderoso, e embora possa não usar os recursos avançados todo o tempo, em algum momento será necessário utilizá-los para alguns problemas difíceis. A seguir são listados os recursos disponíveis da linguagem HQL:

a) habilidade para aplicar restrições a propriedades de objetos associados, possibilitando a navegação do gráfico de objetos;
b) habilidade de retornar somente as propriedades desejadas de uma ou várias entidades, também conhecido como projeção;
c) habilidade de ordenar registros na consulta e paginar;
d) suporte a funções disponíveis no SQL, como `group by, having` e funções de agregações `sum, min e max`;
e) possibilidade de chamar funções SQL definidas pelo próprio usuário;
f) realização de sub-consultas.

2.2.6 Transações de banco de dados

Segundo Bauer e King (2005, p.204) transações de banco de dados se resume nas existências dos critérios como atomicidade, consistência, isolamento e durabilidade, onde todas juntas formam a propriedade ACID.

Atomicidade é manter vários processos agrupados juntos em uma unidade indivisível. Permitir que vários usuários trabalhem simultaneamente com os mesmos dados sem comprometer a integridade e precisão dos dados.

Consistência significa que qualquer transação trabalha com um *set* consistente de dados e deixa os dados em um estado consistente quando a transação se completa.

Durabilidade significa que ao finalizar uma transação, todos os dados modificados se tornem persistente, e nem sequer perdidas caso o sistema falhar durante este processo.

Segundo Bauer e King (2005, p. 212) banco de dados e outros sistema transacionais tentam assegurar isolamento de transação, significando que do ponto de vista de cada transação concorrente, pareça que nenhuma outra esteja em desenvolvimento. Tradicionalmente isto tem sido implementado usando *locking*. Uma transação pode colocar um `lock` em um item particular de dados, prevenindo acesso temporariamente àquele item através de outras transações.

Hibernate se comunica com a base de dados utilizando conexões JDBC ou JTA. Em um sistema que armazena dados em bancos de dados múltiplos, uma unidade particular de trabalho pode envolver acesso a mais de um depósito de dados. Neste caso, você não pode alcançar atomicidade usando somente JDBC. Você requer um gerenciador de transação com suporte para transações distribuídas. Você se comunica com o gerenciador de transações usando JTA (BAUER; KING, 2005, p. 207).

A interface de transação fornece métodos para declarar os limites de uma transação de banco de dados. A interface `Session` possui um método `beginTransaction()` que marca o começo de uma transação de banco de dados. Neste caso, utilizando um ambiente não gerenciado, isso inicia uma transação JDBC na conexão de JDBC. No caso de um ambiente gerenciado, inicia uma transação JTA se não existir nenhuma transação atual, ou somente associa a transação JTA existente. Tudo isso é controlado pelo Hibernate, não é necessário se preocupar com a implementação.

Ao finalizar a transação chamando o método `commit()` é sincronizado o estado da sessão com o banco de dados. Somente serão propagadas as modificações dos objetos persistentes, caso tiver sido anteriormente chamado `beginTransaction()`, senão `commit()` apenas sincroniza o estado da sessão com o banco de dados.

Caso ocorra alguma exceção na conclusão da transação, pode ser chamado o método `rollback()` para que possa retornar ao estado dos objetos modificados antes da chamada `beginTransaction()`.

2.3 JAVA MESSAGE SERVICE (JMS)

> O JMS é, grosso modo, uma API da linguagem Java que permite aplicações como criar, enviar, receber e ler mensagens. Além disso, auxilia o suporte em ambientes heterogêneos e em situações onde a conexão é fornecida de modo intermitente. A comunicação, dentro desse contexto, pode ocorrer de modo assíncrono, mas o JMS

> garante que as mensagens sejam entregues uma única vez. (WAENY JR.; NUMAZAKI, 2004, p. 1).

Definitivamente JMS não é *email*. Ele pode ser definido como um *Midlleware* que implementa uma interface, cuja API permite a comunicação entre componentes ou aplicações, possibilitando que seja distribuída em situações nas quais existe um fraca conectividade. A aplicação remetente e a aplicação destinatário não necessitam estar on-line, pois o recebimento pode ser assíncrono. A grande diferença entra as duas tecnologias não são apenas essa. No JMS existe a garantia do envio e recebimento de uma determinada mensagem. Também existe o sincronismo, de modo que as mensagens podem percorrer um caminho predeterminado de modo garantido e seguro (WAENY JR.; NUMAZAKI, 2004, p. 2).

2.3.1 *Providers* e Clientes JMS

Os *Providers* são aplicações servidoras, desenvolvidas por empresas especializadas, que implementam as interfaces do padrão JMS. Também implementam condições para a administração e controle do serviço a ser prestado ou oferecido. Deve-se notar que essa tecnologia fomenta aplicações comerciais e *open source*. Dentre as aplicações comerciais, podemos destacar o SonicMQ, distribuído em conjunto com o Jbuilder da Borland, e o MQSeries. Entre os servidores *open source,* destaca-se o OpenJMS que está sendo considerado como um produto de boa qualidade, no qual muitos desenvolvedores estão aderindo a sua utilização (WAYNE; LIMA, 2004, p. 2).

Os clientes são programas ou componentes altamente especializados, geralmente desenvolvidos em Java. Utilizando um JMS *Provider* específico, podem produzir e consumir mensagens. Dependendo da especificação do sistema o cliente pode consumir mensagens de forma síncrona ou assíncrona.

2.3.2 Mensagens JMS

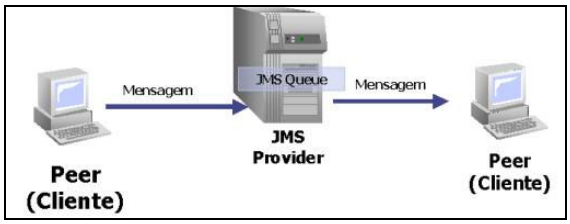
As mensagens padrão JMS permitem transportar além de simples textos, objetos genéricos serializáveis, podendo ser um recurso bastante útil para implementação de um ambiente distribuído (WAYNE; LIMA, 2004). A distribuição é realizada na troca de

mensagens entre o cliente e o *Provider*.

Segundo Waeny Jr e Numazaki (2004, p. 4) mensagens podem estar associadas a domínios diferentes, ou a tipos diferentes, de modo a satisfazer a necessidade de um determinado sistema. Por exemplo, um *chat* deveria utilizar mensagens texto, entretanto, não se aplica ao Sistema de Pagamentos Brasileiro (SBP) pois suas mensagens deveriam ser construídas por *arrays* de *bytes*.

As mensagens podem ser distribuídas através de dois domínios distintos ponto a ponto (P2P) ou dentro do formato *publish/subscribe*.

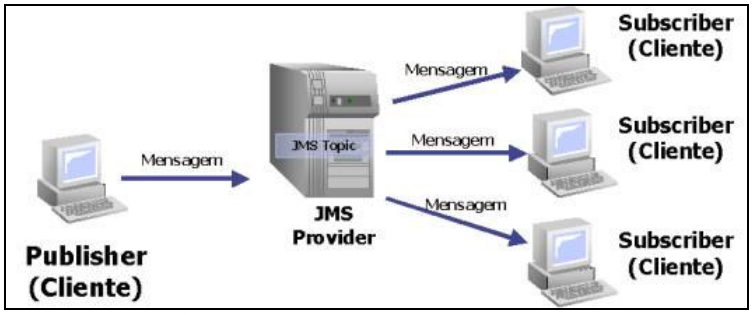
No domínio P2P o cliente envia a mensagem para uma única fila (*queue*), neste caso é somente permitido que exista um produtor e um consumidor. A Figura 3 apresenta uma ilustração da iteração de uma estrutura ponto a ponto.



Fonte: Wayne e Lima (2004, p. 1).

Figura 3 - Interação do domínio P2P

No domínio *publish/subscribe* as mensagens são direcionadas para um tópico (*Topic*) e então entregues para vários clientes inscritos naquele tópico. A Figura 4 apresenta uma ilustração da iteração de uma estrutura *publish/subscribe*.



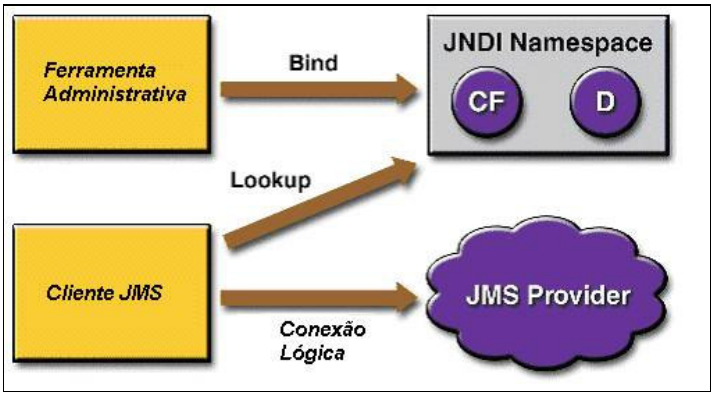
Fonte: Wayne e Lima (2004, p. 1).

Figura 4 - Interação do domínio publish/subscribe

Utilizando os domínios P2P e *publish/subscribe*, as mensagens são entregues somente aos clientes que estão ativos naquele momento. Para resolver este problema as mensagens devem ser criadas utilizando o tipo durável (*Durable*) que permite serem persistidas em disco, e recuperadas logo após os clientes estarem ativos novamente, garantindo desta forma o modelo assíncrono (WAENY JR.; NUMAZAKI, 2004, p. 7).

2.3.3 Arquitetura JMS

A arquitetura JMS é formada basicamente pelos elementos representados na Figura 5, ferramenta administrativa, serviço de nomes, cliente e *Provider* JMS:



Fonte: Wayne e Lima (2004, p. 1).

Figura 5 - Arquitetura JMS

De primeiro momento, no lado do servidor existe alguma ferramenta específica do *Provider* ou algum outro programa especializado que instância os objetos administrativos e publica no contexto do serviço de nomes JNDI.

JNDI é basicamente uma API J2EE que fornece uma interface padrão para localizar usuários, máquinas, objetos, redes e serviços que estão disponibilizados remotamente. Por exemplo, você pode utilizar o JNDI para localizar um computador na rede. Pode ser usado também para buscar objetos Java.

A parte cliente da aplicação, depois disso, obtém as referências remotas dos objetos administrativos através do JNDI, e estabelece uma conexão com o *Provider*. Em seguida, dependendo da aplicação, deve ser adotado uma estrutura adequada para a distribuição das respectivas mensagens.

2.4 TRABALHOS CORRELATOS

Existe hoje uma ferramenta o*pen source* chamada Postgres-R, que implementa o mecanismo de replicação entre bancos Postgres. Pode ser considerada como uma das mais avançadas no que diz respeito à replicação de dados entre SGBDs livres (KEEME, 2000).

O Postgres-R não replica consultas, somente atualizações. Para reduzir o tráfego da

rede é preservado o evento transacional, ou seja, somente os dados que são atualizados na base local são replicados para a base remota, preservando a ordem dos comandos que são executados (KEEME, 2000).

O Departamento de Informática da Pontifícia Universidade Católica (PUC), Rio de Janeiro, desenvolveu um *framework* para Sistema Gerenciador de Bancos de Dados Heterogêneos (SGBDH), chamado Heros.

> O *framework* HEROS tem por objetivo possibilitar a instanciação de softwares (SGBDHs) que permitam que um conjunto de sistemas de bancos de dados heterogêneos, cooperativos mas autônomos, sejam integrados de tal forma em uma federação, que consultas e atualizações possam ser realizadas, de forma transparente à localização dos dados, aos caminhos de acesso e a qualquer heterogeneidade ou redundância que exista. (UCHÔA; MELO, 1999, p. 3).

O Heros disponibiliza para comunicação em experiências já comprovadas, componentes que interagem com os protocolos *Remote Procedure Call* (RPC), *Object Request Broker* (ORB) e chamada direta. No entanto, podem ser definidos outros tipos de comunicação como o *Distributed Component Model* (DCOM) da Microsoft (UCHÔA; MELO, 1999, p. 5).

Tanto o Postgres-R como o Heros, possuem características em relação ao desenvolvimento do trabalho. Apesar do Postgres-R não oferecer heterogeneidade entre os bancos envolvidos na replicação, possui a preservação dos eventos transacionais como também somente a replicação de atualizações. O Heros apresenta uma arquitetura menos acoplada, oferecendo heterogeneidade entre as bases. A replicação de consultas e integração com outros mecanismos de envio de mensagens do Heros, não se aplica no desenvolvimento deste trabalho.

3 DESENVOLVIMENTO DO TRABALHO

O software desenvolvido neste trabalho realiza a tarefa de replicar dados atualizados em um *site* de origem para um *site* de destino. Para permitir este processo, foi criado um *framework* desenvolvido na linguagem Java que facilita a utilização desta estrutura em uma organização, deixando transparente o modo em que ocorre a replicação dos dados.

Utilizando a orientação a objetos e também o *pattern Data Access Object* (DAO), foi possível definir interfaces responsáveis pela persistência dos dados, que facilmente possibilita ampliar o *framework* para outras tecnologias de persistência sem re-trabalho. Isto realmente é interessante no sentido em que desacopla o formato como o dado é persistido, daquele que realmente será replicado.

Basicamente a aplicação que utiliza os recursos do *framework*, necessita apenas informar os parâmetros necessários nos arquivos de configuração, e utilizar os métodos estáticos disponibilizados em uma classe que implementa o *pattern Singleton*.

O *pattern Singleton* foi utilizado para garantir a separação entre camadas. A camada de apresentação não precisa saber a forma como os dados são persistidos ou qual tecnologia está sendo utilizada. Desta forma evita que o código de persistência ou de negócio seja poluído com argumentos desnecessários.

Para desenvolver o trabalho foi necessário primeiramente realizar um levantamento e analisar os requisitos que definem as características que o sistema de possuir. A seguir é descrita a análise e suas especificações.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos descrevem as funcionalidades e características que o sistema deve apresentar. Neste trabalho os requisitos foram divididos em dois grupos, sendo eles funcionais e não funcionais. Os requisitos funcionais listados a seguir, descrevem as funcionalidades do software no que diz respeito em como o mesmo deve se comportar.

a) replicar um dado para um destino informado;
b) preservar os eventos transacionais no destino conforme origem;
c) tratar erros de integração de dados no destino;

d) permitir configurar quais transações serão replicadas;

e) permitir gerenciar as filas locais como também as filas remotas de mensagens.

O *framework* deverá permitir informar inicialmente em arquivos, utilizando o formato padrão XML, os parâmetros necessários para que ele conheça o destino (servidor central) da replicação, e também as classes responsáveis pela persistência dos dados.

Através de uma interface *web* alocado no *site* central deve ser possível configurar quais transações são replicadas e também incluir novas filas de mensagens. Deve permitir monitorar a quantidade de mensagens pendentes na fila, e verificar se existe conexão com o *site* local, que está recepcionando a fila cadastrada no *site* central.

A partir do momento em que os parâmetros são informados e o servidor de mensagens local é iniciado, uma transação pode ser disparada na base de dados pela aplicação. Caso esta transação não apresentar erros durante a persistência local, esta é interceptada pelo *framework* que monta uma mensagem padrão, com todo o lote de transação e comandos aplicados na base local. Esta mensagem é disponibilizada em um servidor de mensagens local para ser enviado para o *site* central de forma assíncrona.

No *site* informado como central, existe um serviço responsável por recepcionar as mensagens disponibilizadas na fila local do *site* de origem. Este serviço tem como responsabilidade consumir a mensagem e processar a transação no *site* central, respeitando a seqüência da transação mantida na base local. Ao processar a transação com sucesso, é disponibilizada a mensagem para as outras filas interessadas na transação, conforme previamente configurado.

Os requisitos não funcionais do sistema apresentam característica da infra-estrutura do sistema. A seguir são apresentados estes requisitos:

a) permitir que as filas remotas possam ser gerenciadas via interface web;

b) garantir entrega das mensagens entre servidores JMS;

c) permitir que possa ser configurado qualquer SGBD compatível com o Hibernate.

O sistema deve permitir que os serviços de recepção de mensagens sejam configurados entre servidores JMS, garantindo assim a entrega de mensagens de modo assíncrono. O Hibernate foi citado como requisito para que o *framework* possa usufruir da compatibilidade de diversos bancos de dados diferentes nos *sites* participantes da replicação.

3.2 ESPECIFICAÇÃO

Nesta sessão do trabalho está disposta a especificação do software desenvolvido, através do uso de diagramas com a *Unified Modeling Language* (UML). Maiores detalhes sobre a UML podem ser obtidos em PAGE-JONES (2001).

Na especificação deste trabalho foram utilizados os diagramas de casos de uso, de atividades dos casos de uso e de classes.

3.2.1 Ferramentas utilizadas na especificação do trabalho

Para a especificação deste trabalho foi escolhida a ferramenta *Enterprise Architect* (EA). Segundo Sparx Systems (2005), o EA é uma ferramenta para auxiliar na especificação de sistemas, que dispõe de componentes para modelar as estruturas e funcionalidades de um software. Esta ferramenta baseia-se na linguagem de modelagem unificada (UML).

O uso do EA foi mais intenso durante a fase inicial do projeto, no qual foi utilizado para documentar e definir todos os detalhes das funcionalidades do projeto, representando por diagramas de casos de uso, atividades e classes.

Dentre os recursos mais marcantes da utilização do EA, pode-se citar alguns que foram de suma importância para definição do projeto:

a) facilidade na criação de vários diagramas, criando *links* entre eles para poder representar suas associações;
b) geração de relatórios, utilizados para documentação do projeto;
c) geração de código fonte diretamente pelo diagrama de classe modelado.

A partir do momento que o projeto estava todo modelado com as definições utilizando a UML, foi utilizado um dos recursos que demonstrou um grande facilitador da ferramenta. Com base no diagrama de classes, foi gerado automaticamente o código fonte das classes do sistema contendo as assinaturas dos métodos e atributos de cada classe. Desta forma, o EA permitiu que não ocorresse o re-trabalho em reescrever novamente todos os métodos e atributos que já tinham sido definidos durante a modelagem. A Figura 6 ilustra a visão de um diagrama de classes na ferramenta EA.

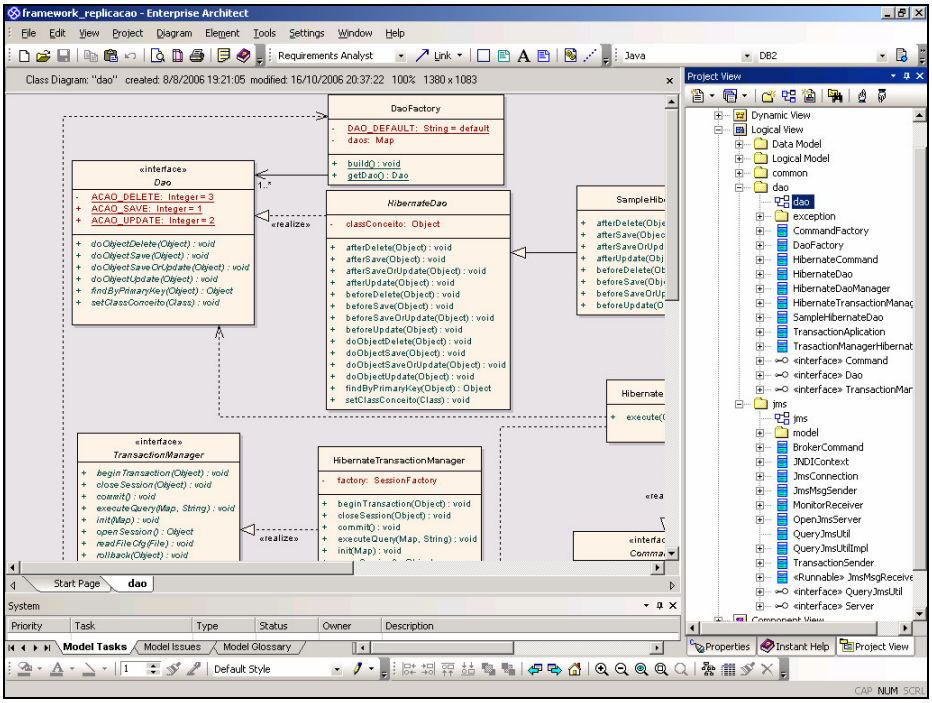


Figura 6 – Exemplo de tela do Entreprise Architect

3.2.2 Casos de uso

O diagrama de casos de uso é utilizado pela UML para definir as interações que o usuário ou algum ator específico da aplicação tem com o software. Cada ação realizada é demonstrada por um caso de uso. A definição do framework resultou em oito casos, conforme listado a seguir:

a) configurar parâmetros de inicialização;

b) iniciar servidor de replicação;

c) cadastrar fila de mensagem;

d) cadastrar transação;

e) associar transação nas filas de mensagens;

f) manter transação;

g) consumir e processar mensagem;

h) distribuir mensagem para as filas interessadas.

A Figura 7 ilustra o diagrama de casos de uso do *framework*.

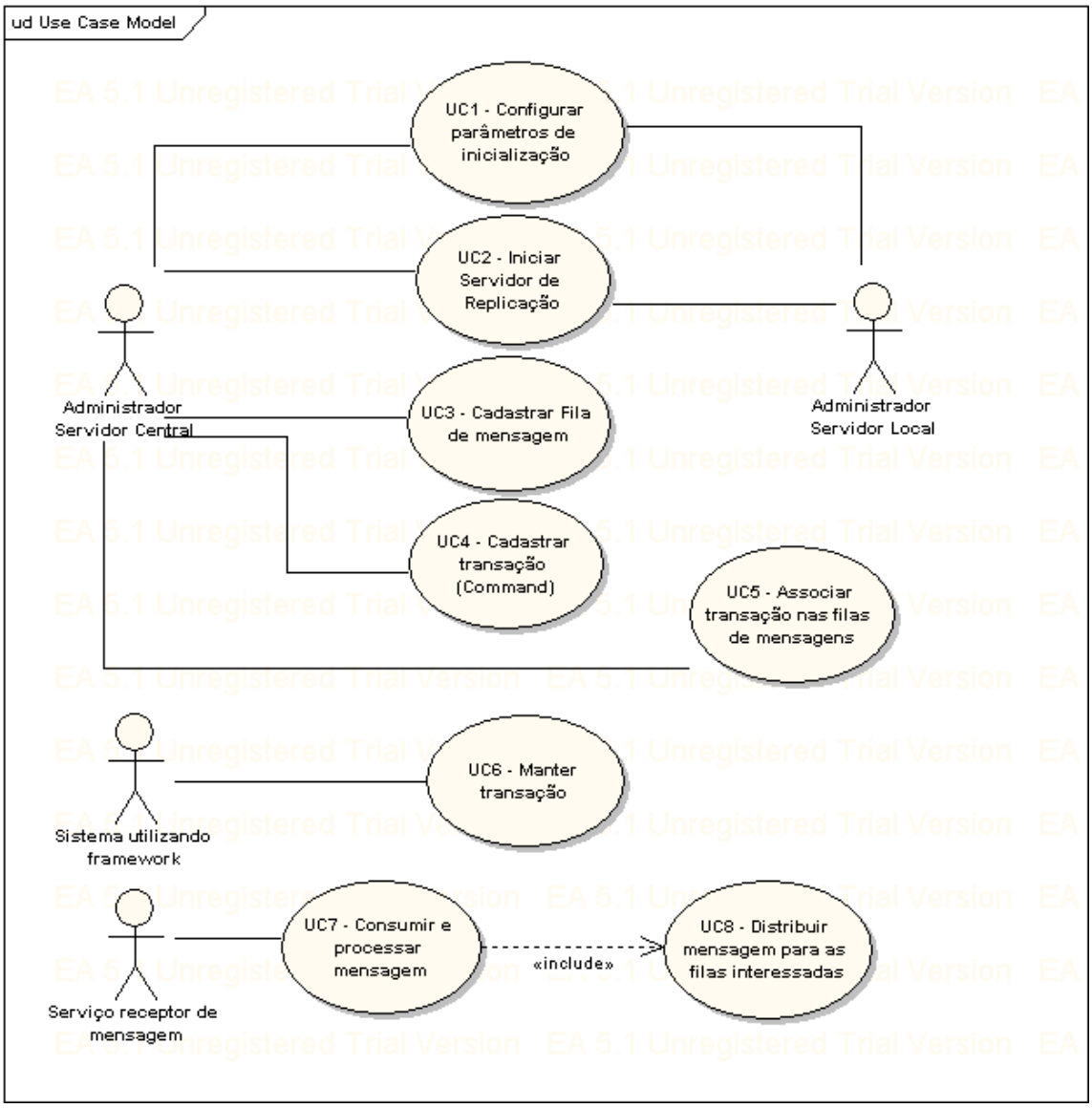


Figura 7 – Diagrama de caso de uso do *framework*.

A seguir serão apresentados nos próximos tópicos uma contextualização de cada caso de uso, detalhando o mesmo em diagramas de atividades.

3.2.2.1 Configurar parâmetros de inicialização

A Figura 8 ilustra o diagrama de atividades do caso de uso "`configurar parâmetros de inicialização`".

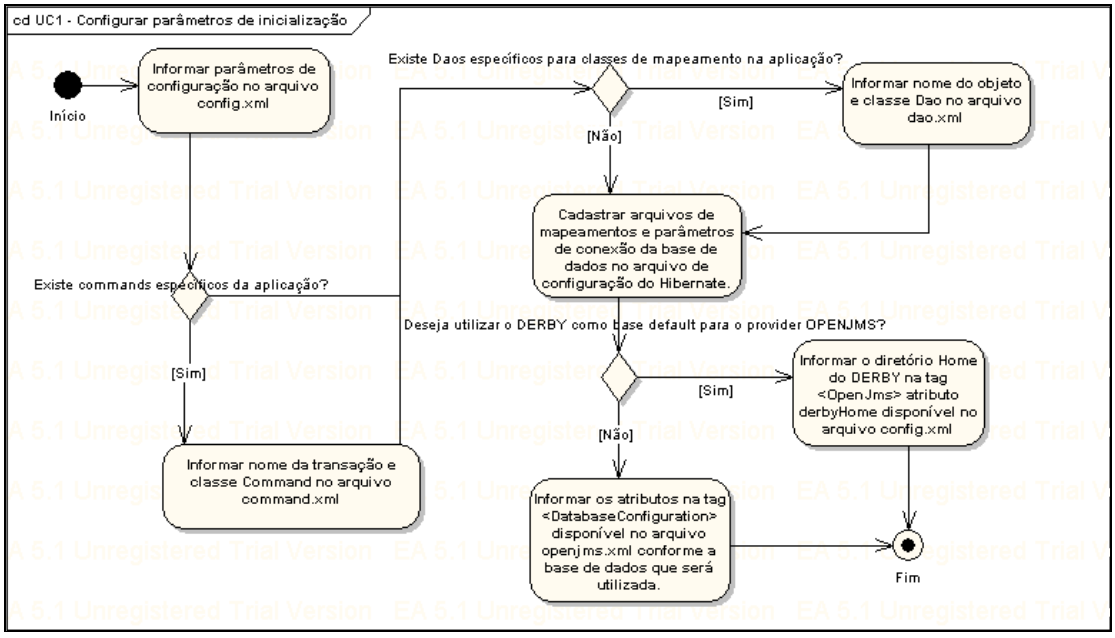


Figura 8 – Diagrama de atividades do caso de uso "`configurar parâmetros de Inicialização`"

O administrador informa no arquivo config.xml os parâmetros necessários para que possa ser iniciada o fluxo de replicação dos dados. Caso for adotado um outro tipo de tecnologia para persistência dos dados, deverá ser informado no mesmo arquivo, as classes que realizam as interfaces de persistência disponibilizadas pelo *framework*.

Como padrão, a execução das transações no destino sempre executa os comandos provenientes da transação, na seqüência em que foi executado no *site* de origem. Existindo a necessidade da transação ser executada de modo diferenciado no *site* de destino, deverá ser criado novas classe que realizam a interface `Command`.

Como o site de destino precisa saber automaticamente que comando ser executado, deve ser informada estas classes de execução da transação no arquivo command.xml, relacionando a classe com o nome da transação.

Também é disponibilizado um arquivo chamado dao.xml, onde pode ser informado classes que possibilitam um pré ou pós processamento dos objetos que são persistidos na base de dados. Deve ser informado neste arquivo o nome do objeto e a classe que realiza a interface `Dao`.

Os parâmetros de configuração da conexão com o banco de dados, devem ser informados no arquivo de configuração do Hibernate. No mesmo arquivo também é necessário registrar os arquivos de mapeamento dos modelos de negócios utilizados na aplicação.

Por padrão o *framework* utiliza o *provider* OpenJms, como servidor de mensagens. Este *provider* utiliza o banco de dados Derby para persistência das mensagens JMS. Havendo necessidade de utilizar outro banco de dados para persistência das mensagens, devem ser alterados os parâmetros que estão previamente configurados para o banco Derby no arquivo openjms.xml.

3.2.2.2 Iniciar servidor de replicação

A Figura 9 ilustra o diagrama de atividades do caso de uso "`iniciar servidor de replicação`".

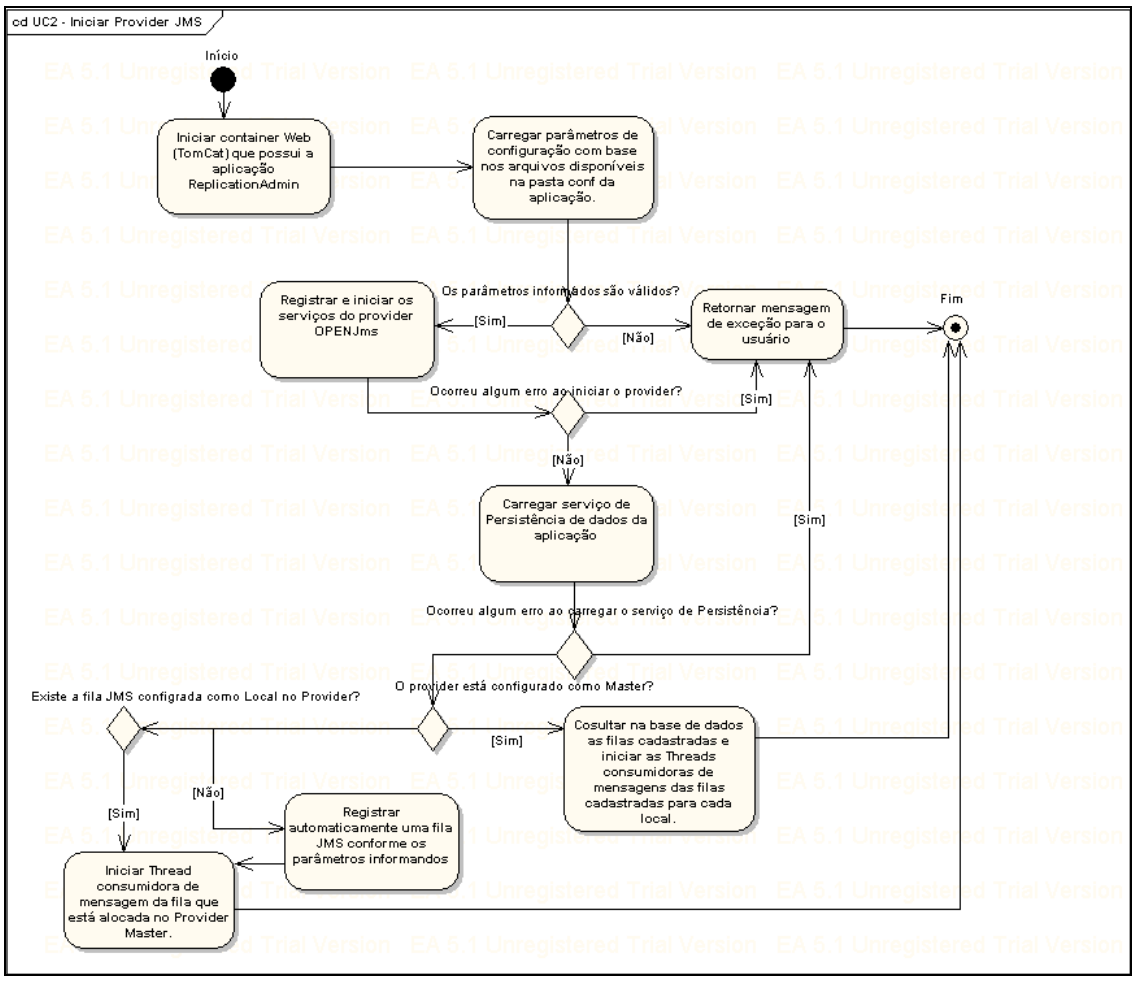


Figura 9 – Diagrama de atividades do caso de uso "`iniciar servidor de replicação`"

O administrador inicia o servidor de replicação que está registrado no container *web* Tomcat. O sistema realiza a leitura dos arquivos de configuração e valida se os parâmetros

foram informados corretamente. Havendo alguma irregularidade nos parâmetros informados, é mostrada uma mensagem de exceção para o usuário abortando o processo.

Após ter carregado todas as configurações, é iniciado os serviços do *provider* OpenJMS e também os serviços de persistência de dados do *framework*. Ocorrendo algum erro durante o início destes serviços, o processo é abortado e retornado mensagem de exceção para o usuário .

No momento em que todos os serviços do OpenJMS como também os serviços de persistência dos dados estejam iniciados, são criadas as *threads* que caracterizam os serviços receptores de mensagens.

Em primeiro lugar a aplicação verifica se o *framework* foi configurado como *master* ou central. Caso sim, realiza uma consulta na base de dados buscando todas as filas de mensagens já previamente cadastradas em alguma outra interação. Os registros retornados, possuem o nome da fila que está registrada no servidor *master* como também um endereço de rede utilizado para consumo das mensagens. Para cada fila cadastrada é criado um serviço receptor de mensagem que se conecta com o *provider* remoto através do endereço de rede cadastrado.

Caso o *framework* estiver configurado como cliente ou local, é verificado se o nome da fila parametrizada existe no *provider*. Não existindo a fila criada, será então automaticamente incluída pelo sistema, deixando transparente esta função para o administrador local. Por fim, é registrado um serviço que irá consumir as mensagens da fila cadastrada no servidor *master*, que possui o mesmo nome da fila do *provider* local.

Sendo assim, podemos perceber que neste momento teremos no servidor *master*, várias filas cadastradas e vários serviços receptores de mensagens para *sites* diferentes. Já no *provider* local, temos apenas uma fila e um serviço receptor de mensagem.

3.2.2.3 Cadastrar fila de mensagens

A Figura 10 ilustra o diagrama de atividades do caso de uso "`cadastrar fila de mensagens`".

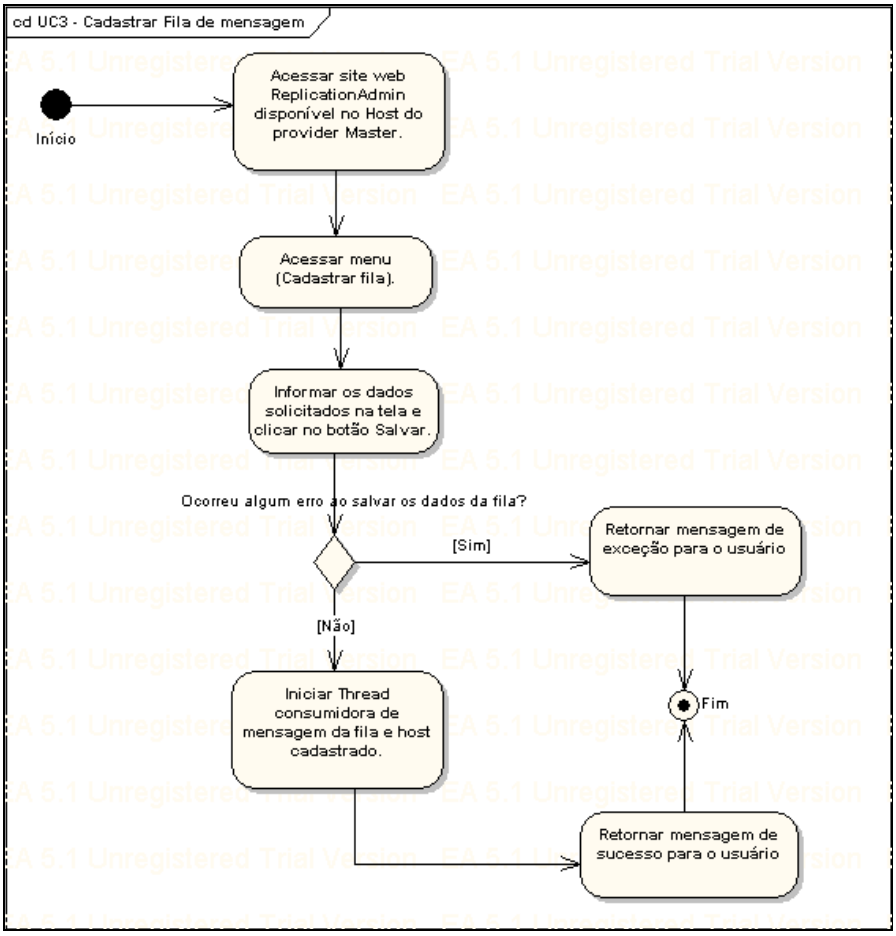


Figura 10 – Diagrama de atividades do caso de uso "`cadastrar fila de mensagens`"

O administrador acessa a interface *web* disponibilizada no servidor *master* no menu "`cadastrar fila`". Será informado na página, nome da fila, endereço de rede e localização geográfica do servidor. O nome da fila deverá conter o mesmo nome parametrizado no *provider* localizado no endereço de rede informado na tela. Ao salvar o cadastro da fila, será automaticamente registrado um serviço receptor de mensagens conectado com o *provider* localizado no endereço de rede informado. Este novo registro incluído será mostrado no menu "listar filas" para monitoramento de conexão e quantidade de mensagens na fila.

3.2.2.4 Cadastrar transação

A Figura 11 ilustra o diagrama de atividades do caso de uso "`cadastrar transação`".

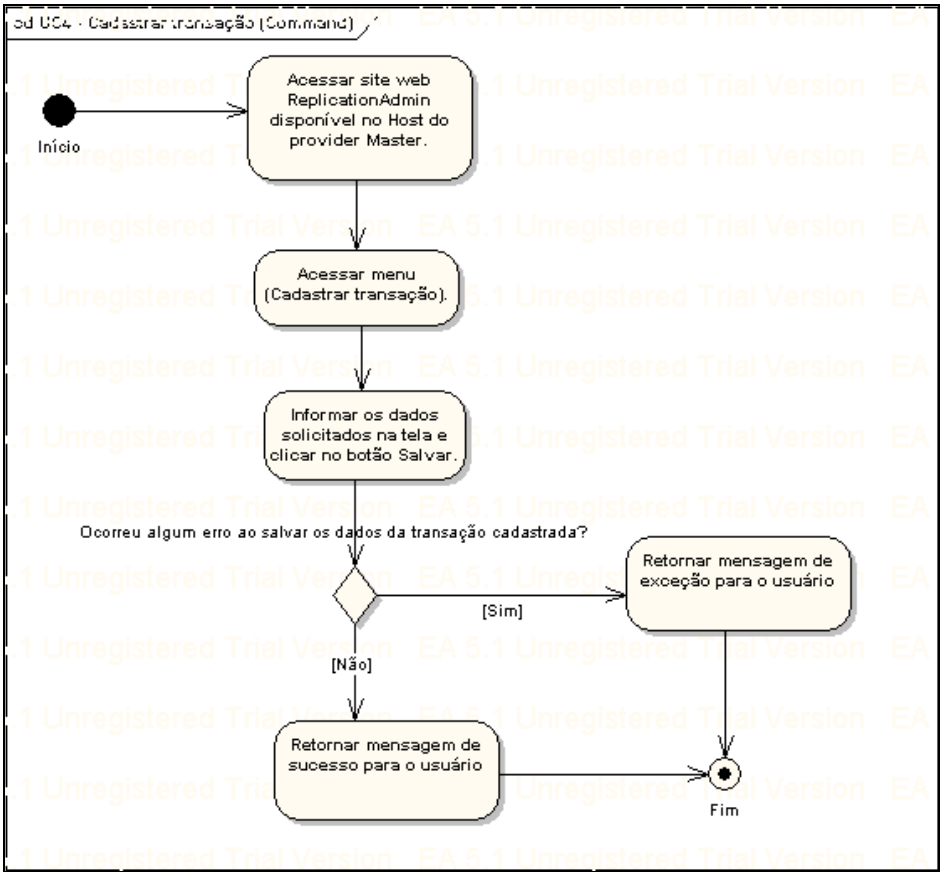


Figura 11 – Diagrama de atividades do caso de uso "`cadastrar transação`"

O administrador acessa a interface *web* disponibilizada no servidor *master* no menu "cadastrar transação". Será informado na página pelo usuário, nome da transação e uma breve observação dos objetivos da transação. Ao concluir o cadastro, a nova transação é mostrada na listagem localizada na parte inferior da página.

3.2.2.5 Associar transação nas filas de mensagens

A Figura 12 ilustra o diagrama de atividades do caso de uso "`associar transação`".

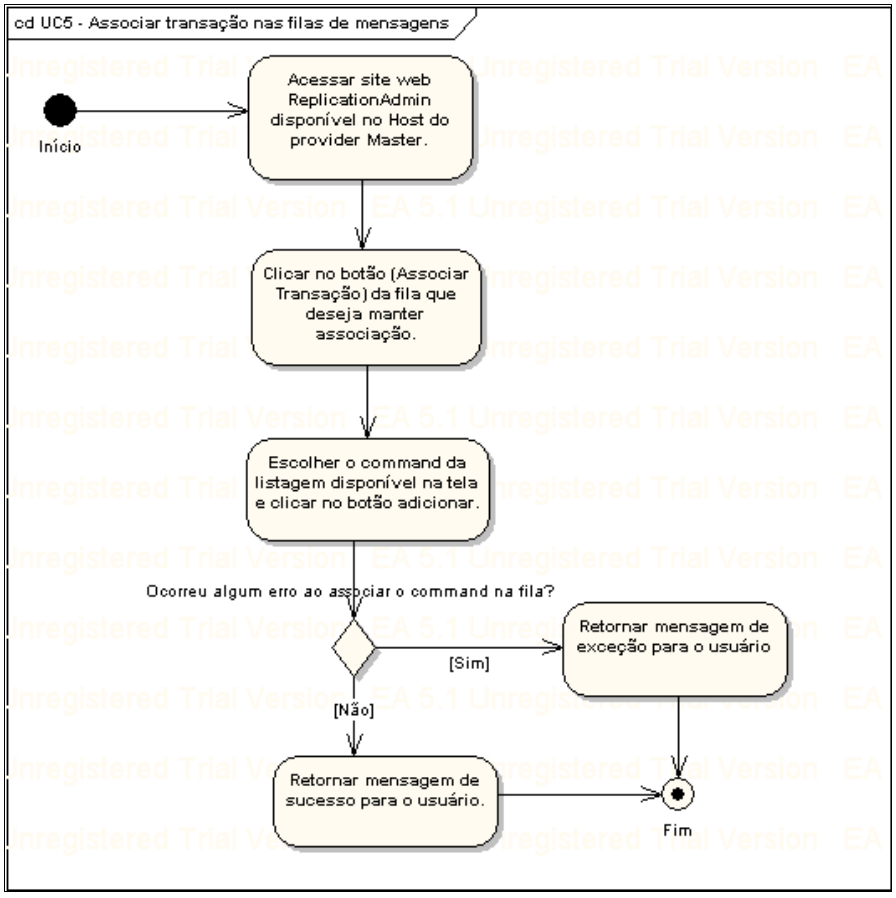


Figura 12 – Diagrama de atividades do caso de uso "`associar transação`"

O administrador acessa a interface *web* disponibilizada no servidor *master*. No menu "`listar filas`" e pressiona sobre o link "`associar transação`" localizada na fila em que se deseja associar uma transação. Será selecionado na página pelo usuário, nome da transação que se deseja associar. Ao concluir a associação, a nova transação é adicionada para fila corrente é mostrada na listagem localizada na parte inferior da página. A transação que foi adicionada é removida da listagem de seleção, não permitindo que seja incluída duas vezes a mesma transação.

3.2.2.6 Manter transação

A Figura 13 ilustra o diagrama de atividades do caso de uso "`manter transação`"

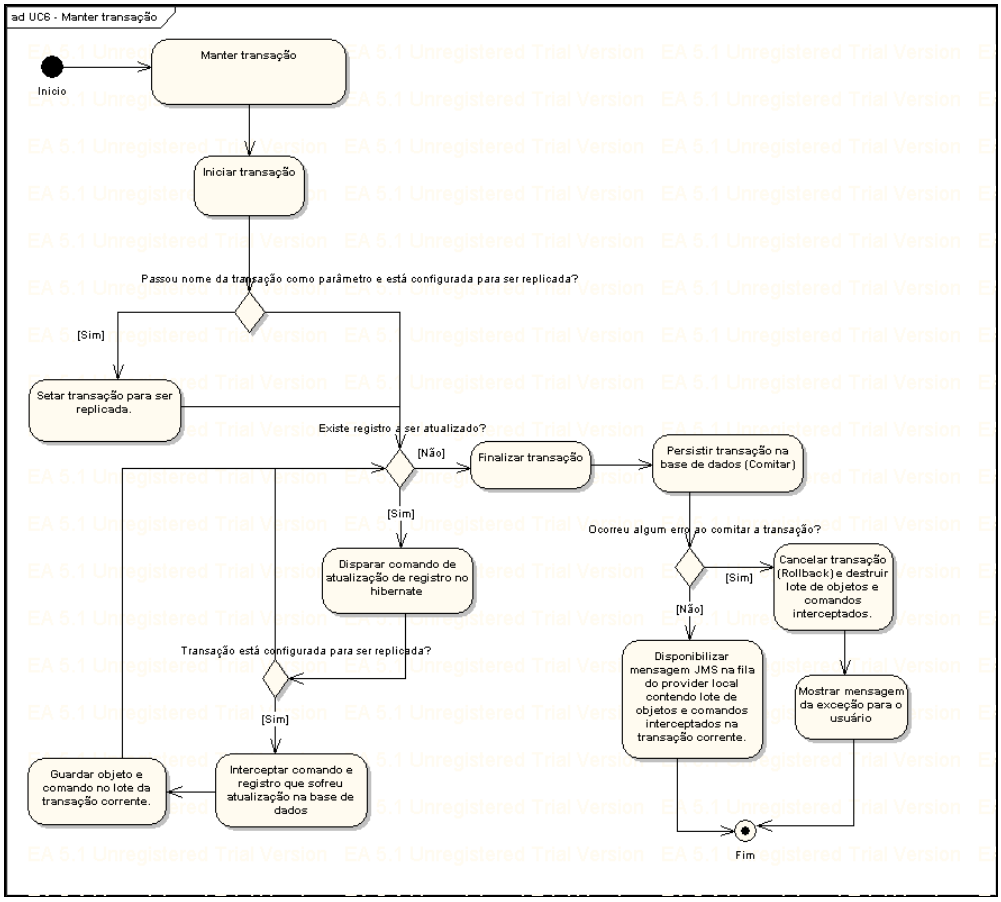


Figura 13 – Diagrama de atividades do caso de uso "`manter transação`"

A aplicação utilizando *framework* solicita que uma transação seja iniciada e passa o nome da transação como parâmetro e automaticamente é demarcado o escopo da transação para ser replicado, caso contrário a transação somente terá efeito na base local.

Enquanto existiram registros a serem atualizados e escopo para replicação esteja demarcado, todos os objetos e comandos de persistência disparados pela aplicação são interceptados e incluídos no escopo de replicação. No momento em que não existir mais registros a serem atualizados a transação será finalizada pela aplicação.

Ao finalizar a transação será disparado o comando para gravar os dados fisicamente na base de dados, mais conhecido como comando *commit*. Ocorrendo algum erro ao gravar os dados na base, é retornada uma mensagem de exceção para a aplicação e o estado dos objetos anterior ao início desta transação é recuperado, também conhecido como comando *rollback*. Finalizando com sucesso a transação, é criada uma mensagem JMS contendo o lote de objetos e comandos contidos no escopo de replicação. Esta mensagem é disponibilizada na fila do *provider* local.

3.2.2.7 Consumir e processar mensagem

A Figura 14 ilustra o diagrama de atividades do caso de uso "`consumir e processar mensagem`".

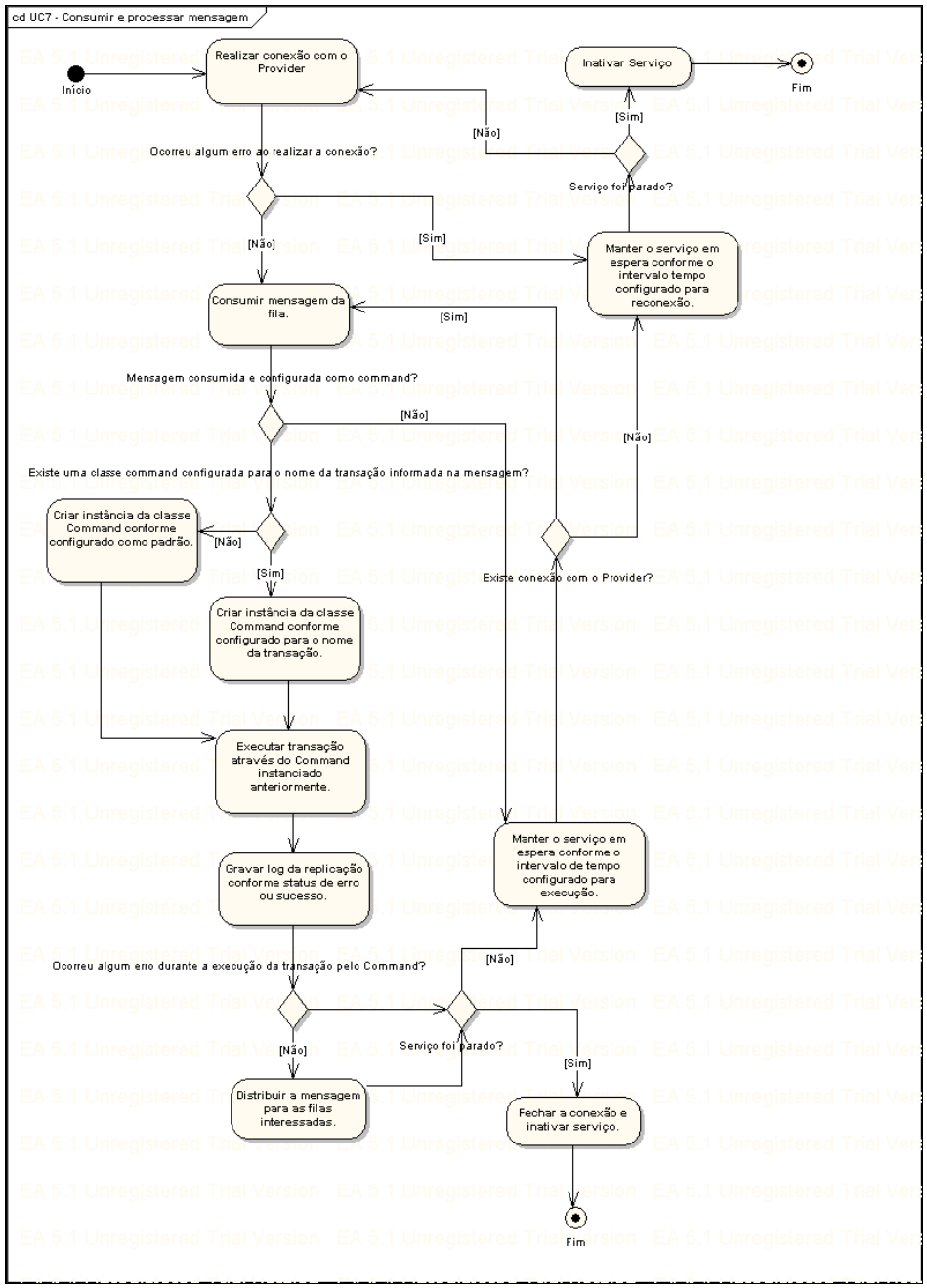


Figura 14 – Diagrama de atividades do caso de uso "`consumir e processar mensagem`"

O serviço receptor de mensagens realiza uma conexão com o *provider*, localizado no endereço de rede passado como parâmetro ao registrar o serviço. Ocorrendo algum erro de

comunicação com o *provider*, o serviço é mantido em espera durante um tempo pré-definido de re-conexão parametrizado nas configurações do *framework*. Ao expirar o tempo de espera, terá outra tentativa, até que consiga estabelecer uma conexão. Caso algum outro processo solicitar que o serviço seja parado, automaticamente a *thread* será colocada no estado inativo.

Tendo uma conexão estabelecida, o processo de consumo de mensagem da fila será iniciado. Ao consumir uma mensagem e estando a mesma configurada como transação, é delegada a mensagem para o *broker*. O *broker* verifica se o nome da transação está informada no arquivo `command.xml`, para identificar se foi configurada alguma classe `Command` específica para executar a transação que foi replicada com um determinado nome. A classe informada é criada, senão, é criada uma instância da classe configurada com o nome "`default`". A partir da instancia criada da classe `Command`, é invocado o método `execute()` passando como parâmetro a transação que foi delegada para o *broker*.

Executando a transação, não importando o estado de erro ou sucesso, é gravado um *log* em uma tabela específica do *framework*, destinada para consultas e verificação de erros. Executada a transação com sucesso, o *broker* disponibiliza a mensagem para as filas interessadas, caso contrário não disponibiliza.

Finalizando o processamento da mensagem, o serviço verifica novamente se existem mensagens a consumir, não existindo é mantido o serviço em espera por um tempo pré-configurado, caracterizando a janela de tempo entre a execução dos serviços. Expirando este intervalo de espera, novamente será verificado se ainda existe comunicação com o *provider*. Não existindo conexão, o serviço entrará no escopo de re-conexão que já foi citado anteriormente.

3.2.2.8 Distribuir mensagem para as filas interessadas

A Figura 15 ilustra o diagrama de atividades do caso de uso "`distribuir mensagem para as filas interessadas`".

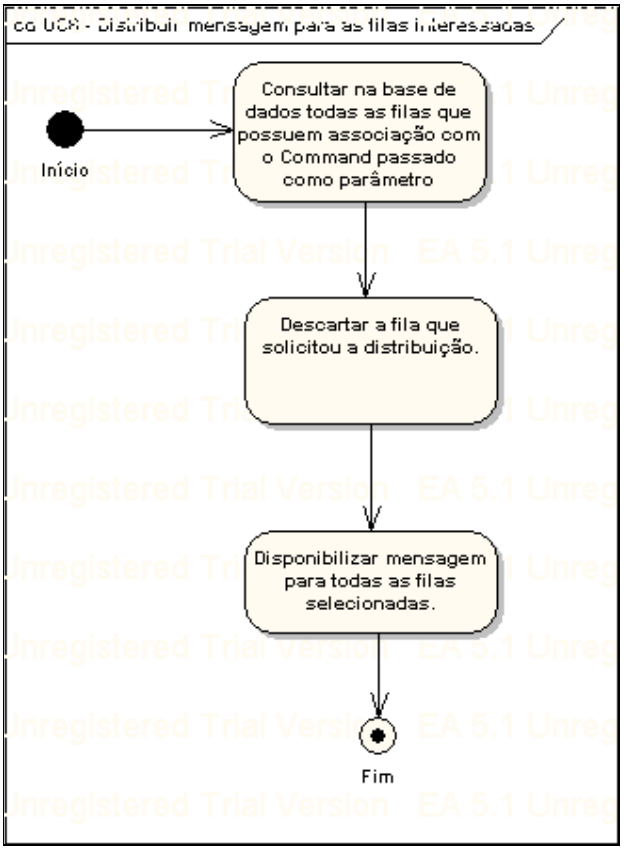


Figura 15 – Diagrama de atividades do caso de uso "`distribuir mensagem para as filas interessadas`"

O serviço receptor de mensagem solicita ao *broker* que a mensagem seja disponibilizada para as outras filas interessadas na transação.

O *broker* consulta na base de dados todas as filas que possuem associação com o nome da transação passada como parâmetro. Tendo o retorno das filas interessadas, é disponibilizada a mensagem contendo a transação para cada fila retornada, descartando somente a fila que solicitou a distribuição da mensagem. A fila que solicita a distribuição, não precisa receber novamente pois já processou a transação contida na mensagem .

3.2.3 Diagrama de classes

O diagrama de classes foi utilizado na especificação para modelar as funcionalidades do projeto. Este diagrama teve grande importância na especificação do *framework* pois através dele foi possível definir todos os atributos e assinatura de métodos de cada classe necessária para o desenvolvimento. Devido ao fato de ser definido grande parte das classes

durante a fase de especificação e também a geração de código automático a partir da ferramenta EA, o desenvolvimento foi altamente produtivo. Todo o desenvolvimento foi realizado a partir das classes previamente definidas na fase de especificação.

O software implementado neste trabalho foi composto por 34 classes sendo que 12 delas foram definidas para representar exceções específicas do *framework*. Também foram definidas outras 5 interfaces para tornar menos acoplado algumas tecnologias utilizadas no projeto.

A parte mais importante do sistema foi definida em dois diagramas que separou as camadas de persistência e replicação dos dados. A seguir será explicada cada camada apresentando seus diagramas. Por fim será demonstrado o diagrama de classes necessário para apresentar a definição da aplicação *web* para o gerenciamento da replicação.

3.2.3.1 Camada de persistência dos dados

A camada de persistência dos dados é o ponto do *framework* que interage com a tecnologia utilizada para realizar a persistência dos dados na aplicação. Como padrão foi utilizado o Hibernate como *middleware* que facilita a execução das operações que precisam ser realizadas com o banco de dados. Foram criadas interfaces para poder deixar a tecnologia de persistência menos acoplado com o projeto, e também outras classes utilitárias que servem como fábricas de objetos.

A Figura 16 ilustra o diagrama de classes que representa a camada de persistência de dados.

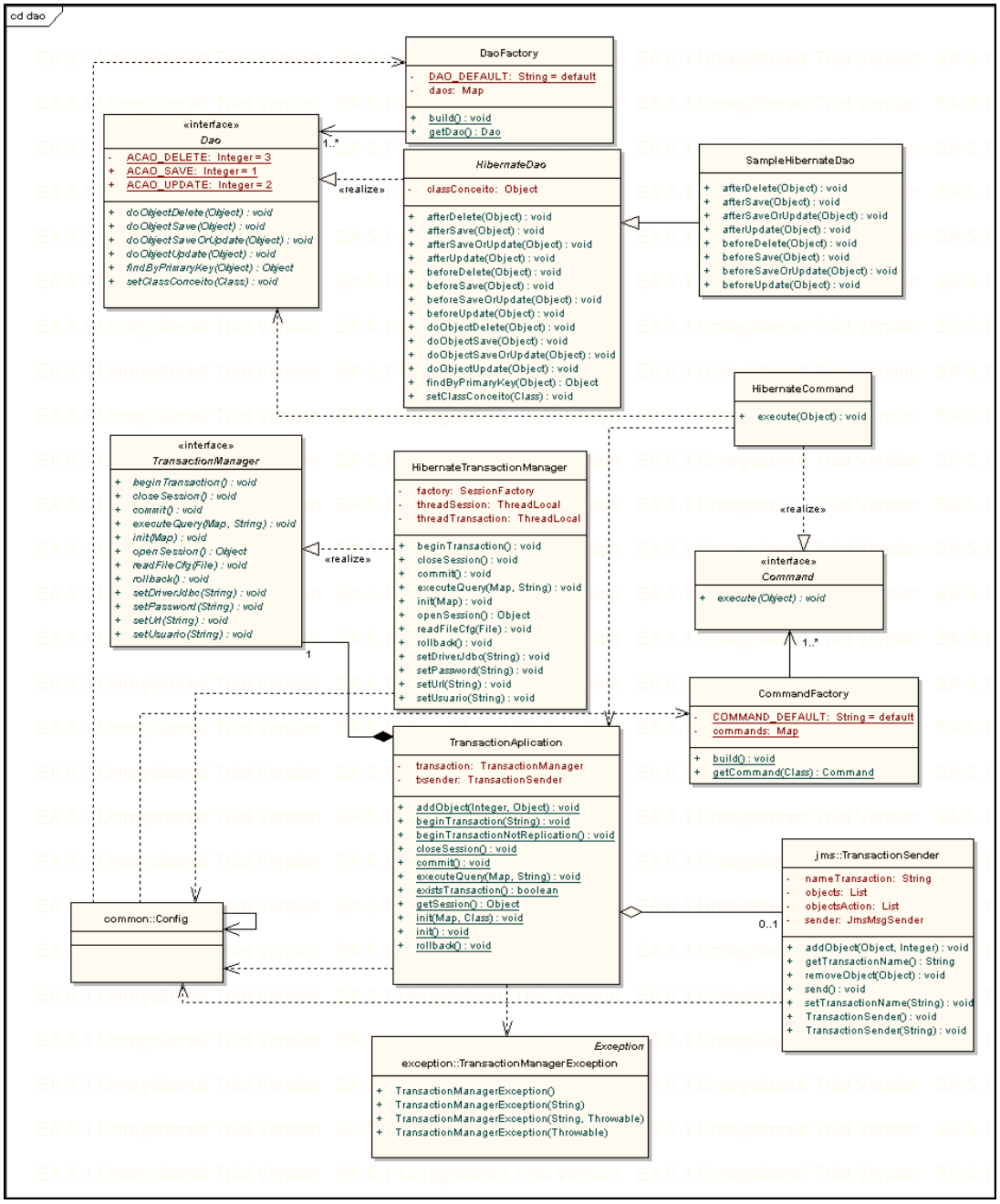


Figura 16 – Diagrama de classes da camada de persistência do *framework*

A seguir é apresentada uma breve descrição das classes e interfaces consideradas mais importantes da camada de persistência dos dados. Algumas classes são omitidas pois apresentam apenas características de controle da aplicação.

A classe `HibernateDao` implementa a interface `Dao`, que possui as assinaturas dos métodos básicos utilizados para interagir com os objetos POJOS. Esta classe pode ser caracterizada como um objeto genérico utilizado para qualquer tipo de POJO a ser manipulado, tendo como pré-requisito estar mapeado e registrado na configuração do

Hibernate. Esta classe não pode ser instanciada pois foi definida como abstrata, desta forma deve obrigatoriamente conter uma sub-classe da mesma. O Quadro 5 indica a estrutura da classe `HibernateDao`.

HibernateDao	
Atributos	**Descrição**
String classConceito	Nome da objeto POJO que será utilizado pela classe.
Métodos	**Descrição**
abstract afterDelete(Object)	Se implementado na sub-classe será executado após o método doObjectDelete.
abstract afterSave(Object)	Se implementado na sub-classe será executado após o método doObjectSave.
abstract afterSaveOrUpdate(Object)	Se implementado na sub-classe será executado após o método doObjectSaveOrUpdate.
abstract afterUpdate(Object)	Se implementado na sub-classe será executado após o método doObjectUpdate.
abstract beforeDelete(Object)	Se implementado na sub-classe será executado antes do método doObjectDelete.
abstract beforeSave(Object)	Se implementado na sub-classe será executado antes do método doObjectSave.
abstract beforeSaveOrUpdate(Object)	Se implementado na sub-classe será executado antes do método doObjectSaveOrUpdate.
abstract beforeUpdate(Object)	Se implementado na sub-classe será executado antes do método doObjectUpdate.
doObjectDelete(Object)	Exclui o registro passado como parâmetro da base de dados.
doObjectSave(Object)	Inclui o registro passado como parâmetro na base de dados.
doObjectSaveOrUpdate(Object)	Inclui ou atualiza o registro passado como parâmetro.
doObjectUpdate(Object)	Atualiza o registro passado como parâmetro na base de dados.
findByPrimaryKey(Object)	Retorna um registro da base de dados a partir da classe que mapeia o identificador do objeto.
setClassConceito(Object)	Configura a classe para um determinado POJO.

Quadro 5 – Atributos e métodos da classe `HibernateDao`

A classe `HibernateTransactionManager` implementa a interface `TransactionManager`, que possui os métodos básicos para conexão com o banco de dados e demarcação de transação.

Nesta classe foi utilizado a classe `java.lang.ThreadLocal` da biblioteca padrão do Java. Segundo (BAUER; KING, 2005, p. 391) uma `ThreadLocal` cria uma área de armazenamento de objetos particular para uma `Thread`. Ela permite que seja implementado

um contexto de persistência, parecido com JTA de um contexto de transação. Quaisquer componentes chamados na mesma requisição compartilharão a mesma sessão e contexto de persistência. O Quadro 6 indica a estrutura da classe `HibernateTransactionManager`.

HibernateTransactionManager	
Atributos	**Descrição**
ThreadLocal threadTransaction	Armazena o contexto da transação.
ThreadLocal threadSession	Armazena o contexto da sessão.
SessionFactory factory	Objeto que cria sessões do Hibernate
Métodos	**Descrição**
beginTransaction	Cria um objeto de transação do Hibernate e armazena no contexto threadTransaction
closeSession	Fecha a sessão e transação corrente e elimina do contexto.
commit	Efetiva a transação.
executeQuery(Map,String)	Executa um comando HQL passando os parâmetros utilizados na cláusula como também o próprio comando na chamada do método.
init	Método utilitário para iniciar a SessionFactory
openSession	Retorna uma sessão do Hibernate e armazena no contexto threadSession.
readFileCfg(File)	Método utilitário para ler as configurações obtidas através de um arquivo.
rollback(Object)	Retorna o estado atual dos objetos antes de demarcar a transação.
setDriverJdbc(String)	Método utilitário para setar o nome da classe do Driver JDBC.
setPassword(String)	Método utilitário para setar senha para conexão com banco de dados.
setUrl(String)	Método utilitário para setar Url para conexão com banco de dados.
setUsuario(String)	Método utilitário para setar o nome do usuário para conexão com banco de dados.

Quadro 6 – Atributos e métodos da classe `HibernateTransactionManager`

A classe `TransactionAplication` é também bastante importante nesta camada. Esta classe é utilizada para se comunicar com a implementação de `TransactionManager`, que é disponibilizado `HibernateTransactionManager` como padrão. Basicamente as assinaturas dos métodos são idênticos com as assinaturas dos métodos da interface `TransactionManager`. Todos os métodos desta classe são estáticos contemplando o *patern Singleton* de projetos. A aplicação que utiliza o *framework* se comunica na maioria das vezes com esta classe em conjunto com as implementações de `Dao`, que permite realizar todas as operações com o banco de dados, desta forma não poluindo a camada de apresentação ou de

negócios da aplicação. O Quadro 7 indica a estrutura da classe `TransactionAplication`.

TransactionAplication	
Atributos	**Descrição**
TransactionManager transaction	Define a implementação de TransactionManager
TransactionSender txSender	Gerencia e armazena o escopo de replicação de dados
Métodos	**Descrição**
addObject(Integer,Object)	Inclui no escopo de replicação um objeto relacionando com o comando disparado.
beginTransaction(String)	Inicia uma transação delegando a execução para TransactionManager e cria um escopo de replicação.
beginTransactionNotReplication	Inicia uma transação delegando a execução para TransactionManager, sem criar um escopo de replicação.
closeSession	Fecha a sessão mantida em threadSession.
commit	Finaliza a transação delegando a execução para TransactionManager.
executeQuery(Map String)	Executa um comando HQL delegando a execução para TransactionManager.
existsTransaction	Verifica se existe uma transação iniciada em threadTransaction
getSession	Retorna a sessão do contexto threadSession.
init(Map,Class)	Cria e inicia as configurações de TransactionManager conforme passagem de parâmetros.
init	Cria e inicia TransactionManager conforme previamente configurado no framework.
rollback	Retorna o estado atual do objeto antes de demarcar a transação. Esta execução é delegada para TransactionManager.

Quadro 7 – Atributos e métodos da classe `TransactionAplication`

A classe `TransactionSender` define a responsabilidade do escopo de replicação da transação. Para cada transação terá uma nova instância desta classe, que irá manter uma lista de objetos e ações que foram aplicados sobre os objetos durante a transação corrente. O Quadro 8 indica a estrutura da classe `TransactionSender`.

TransactionSender	
Atributos	**Descrição**
String nameTransaction	Nome da transação utilizado no processo de replicação.
List objects	Lista de objetos interceptados durante a transação.
List objectsAction	Lista de ações aplicadas sobre os objetos interceptados.
JmsMsgSender sender	Classe do framework utilizada para enviar mensagem.
Métodos	**Descrição**
addObject(Object,Integer)	Adiciona objeto e ação interceptada nas listas apropriadas.
getTransactionName	Retorna o nome da transação.
removeObject(Object)	Remove objeto e ação das listas apropriadas.
send	Monta um lote contendo objetos e ações e envia através do atributo sender.
setTransactionName	Atribui o nome da transação.
TransactionSender(String)	Construtor que passa como parâmetro o nome da transação.

Quadro 8 – Atributos e métodos da classe `TransactionSender`

A classe `HibernateCommand` implementa a interface `Command` que possui apenas um método que executa a transação passada como parâmetro. Está classe utiliza `TansactionAplication` em conjunto com implementações de `Dao` para executar uma a transação no destino proveniente de uma mensagem replicada.

3.2.3.2 Camada de replicação dos dados

A camada de replicação de dados é a camada responsável por enviar e consumir as mensagens JMS de uma determinada fila alocada em um *provider*. Possui também outra função que se destaca entre as demais que é gerenciar os serviços receptores de mensagens.

Todas as classes foram definidas para exercerem funções específicas, não misturando responsabilidades entre elas. Desta forma foram definidas classes para conexão, envio, consumo de mensagens, tornado-se claro a definição desta camada. A Figura 17 ilustra o diagrama de classes que representa a camada de replicação de dados.

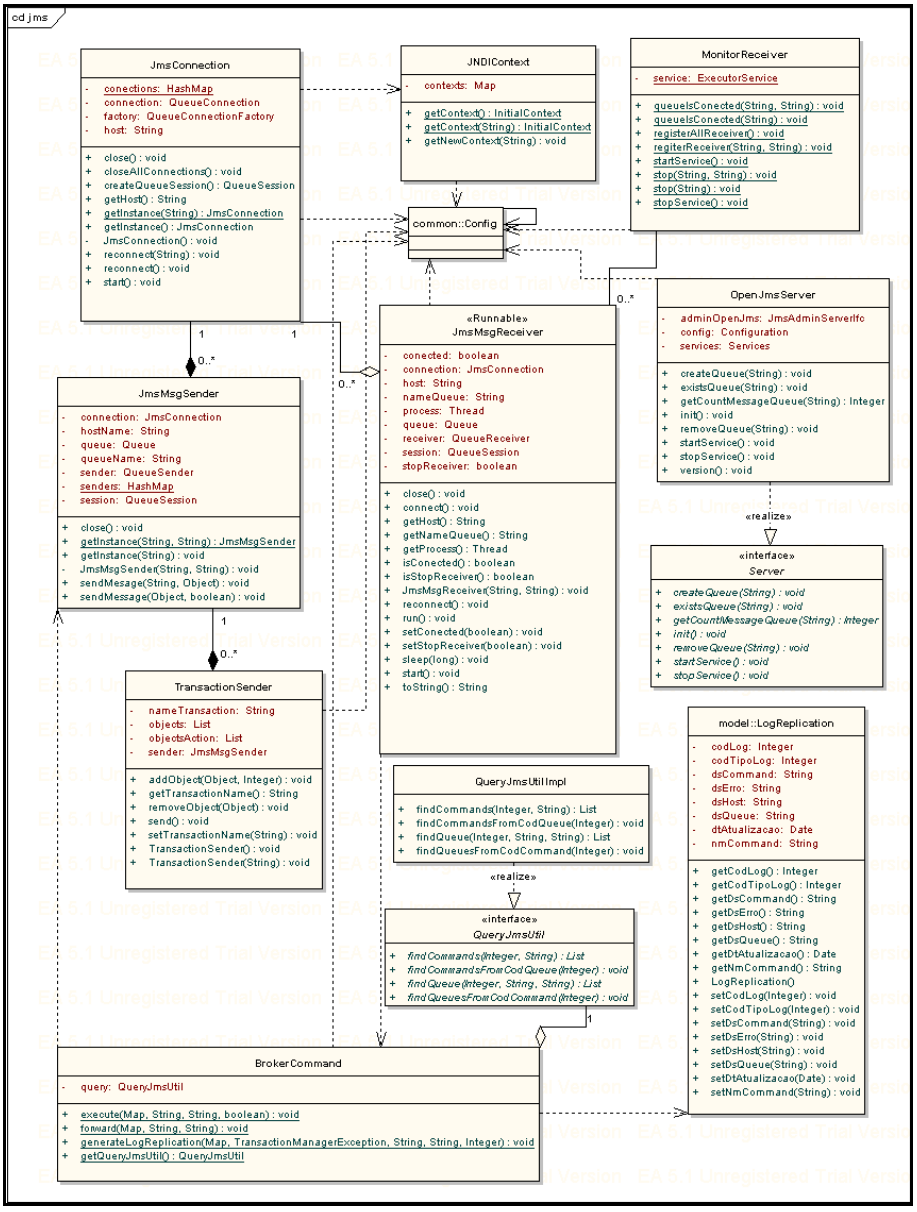


Figura 17 – Diagrama de classes da camada de replicação de dados do *framework*

A seguir é apresentada uma breve descrição das classes e interfaces consideradas mais importantes da camada de replicação dos dados. Algumas classes serão omitidas pois apresentam apenas características de controle da aplicação.

A classe `JmsConnection` é responsável por criar e armazenar as conexões existentes entre *provides* JMS. É utilizada com maior freqüência no gerenciamento do servidor central onde necessita conter várias conexões para *providers* dispersos geograficamente. Para não ocorrer duplicidade de conexões, é armazenado na própria classe as instâncias que são criadas, pois posteriormente pode ser retornada caso outro serviço solicitar a mesma conexão. O Quadro 9 indica a estrutura da classe `JmsConnection`.

JmsConnection	
Atributos	**Descrição**
static Map conections	Utilizado para armazenar as instâncias de JmsConnection.
QueueConnection connection	Classe da API JMS, que representa uma conexão com o provider JMS.
QueueConnectionFactory factory	Classe da API JMS, que representa uma fábrica de conexões com o provider.
String host	Endereço de rede utilizado para conexão com o provider.
Métodos	**Descrição**
close	Fecha a conexão com o provider.
closeAllConnections	Fechar todas as conexões existentes com qualquer provider.
createQueueSession	Cria uma sessão JMS a partir do atributo connection.
getHost	Retorna o endereço de rede da conexão.
static getInstance(String)	Utilizado para retornar uma instância de JmsConnection conforme endereço de rede passado como parâmetro e armazenar no atributo conections.
static getInstance	Utilizado para retornar uma instância de JmsConnection do provider local configurado e armazenar no atributo conections.
JmsConnection	Construtor privado que cria uma conexão com o provider.
static reconnect(String)	Caindo a conexão pode ser chamado este método para restabelecer a conexão com o endereço de rede passado como parâmetro.
static reconnect	Caindo a conexão pode ser chamado este método para restabelecer a conexão com o provider local.
start	Inicia a conexão e permite o consumo de mensagem.

Quadro 9 – Atributos e métodos da classe `JmsConnection`

A classe `JmsMsgSender` é responsável por criar e armazenar os objetos utilizados para o envio das mensagens JMS, que possui o mesmo comportamento da classe `JmsConnection` para evitar a duplicidade de instâncias. É composta da classe `JmsConnection` utilizada para criar a sessão com o *provider*. A partir da sessão é instanciado o objeto da API JMS que efetivamente irá enviar as mensagens. O Quadro 10 indica a estrutura da classe `JmsMsgSender`.

JmsMsgSender	
Atributos	**Descrição**
JmsConnection connection	Representa a conexão com o provider.
String hostName	Endereço de rede do provider
Queue queue	Classe que representa uma fila da API JMS.
String queueName	Nome da fila.
QueueSender sender	Classe para envio de mensagem da API JMS.
static Map senders	Utilizado para armazenar as instâncias de JmsMsgSender.
QueueSession session	Classe que representa uma sessão da API JMS.
Métodos	**Descrição**
close	Fecha a sessão JMS.
static getInstance(String, String)	Utilizado para retornar uma instância de JmsMsgSender conforme endereço de rede e fila passado como parâmetro e armazenar no atributo senders.
static getInstance	Utilizado para retornar uma instância de JmsMsgSender conforme endereço de rede e fila configurado como local e armazenar no atributo senders.
JmsMsgSender	Construtor privado da classe.
sendMessage(Object,boolean)	Recebe um objeto como parâmetro, monta uma mensagem JMS e envia para a fila configurada.

Quadro 10 – Atributos e métodos da classe `JmsMsgSender`

A classe `JmsMsgReceiver` representa o serviço receptor de mensagens. Esta classe a característica de rodar como um processo em segundo plano, contendo uma janela de tempo para execução do consumo de mensagens.

Sempre que ocorre alguma falha na conexão com o *provider* , o método `reconnect` é invocado automaticamente, restabelecendo a conexão.

Esta classe não possui a responsabilidade de definitivamente processar e executar a transação composta na mensagem. Somente consome e delega a execução para a classe `BrokerCommand`. O Quadro 11 indica a estrutura da classe `JmsMsgReceiver`, sendo que alguns métodos que não representam grande importância foram omitidos.

JmsMsgReceiver	
Atributos	**Descrição**
JmsConnection connection	Representa a conexão com o provider.
String hostName	Endereço de rede do provider
Queue queue	Classe que representa uma fila da API JMS.
String queueName	Nome da fila.
boolean conected	Flag que determina se o serviço possui uma conexão estabelecida com o provider
QueueSession session	Classe que representa uma sessão da API JMS.
Thread process	Thread que executa o consumo de mensagem
QueueReceiver receiver	Classe para consumo de mensagem da API JMS.
boolean stopReceiver	Flag que determina se o serviço foi inativado por algum outro processo externo.
Métodos	**Descrição**
close	Fecha a sessão JMS.
connect	Cria uma conexão com o provider.
JmsMsgReceiver(String,String)	Construtor que cria um serviço conforme endereço de rede e nome da fila passado como parâmetro.
reconnect	Caindo a conexão com o provider este método é chamado para tentar restabelecer a conexão.
run	Implementação do método da interface Runnable utilizado para execução da Thread.
sleep(long)	Utilizado para manter o serviço em espera conforme o tempo passado como parâmetro.
start	Inicia a execução da Thread.

Quadro 11 – Atributos e métodos da classe `JmsMsgReceiver`

A classe `MonitorReceiver` é responsável por criar e gerenciar os serviços receptores de mensagem. Através desta classe é possível registrar novos serviços, verificar se existe conexão estabelecida com o *provider* e também iniciar aqueles que já foram registrados anteriormente. Ao chamar o método `stopService()` todos os serviços que estão sendo executados são parados de forma simultânea. Sempre que a aplicação que está utilizando o *framework* necessita registrar um novo serviço, utiliza esta classe utilitária para manter todos os serviços no mesmo escopo. O Quadro 12 indica a estrutura da classe `MonitorReceiver`.

MonitorReceiver	
Atributos	**Descrição**
static Map services	Utilizado para armazenar as instâncias de JmsMsgReceiver.
Métodos	**Descrição**
queueIsConected(String,String)	Verifica se existe conexão do serviço identificado pelo host e fila passada como parâmetro.
registerAllReceiver	Registra todos os serviços previamente cadastrados na base de dados.
regiterReceiver(String,String)	Registra um novo serviço conforme host e fila passada como parâmetro.
startService	Incia a execução de todos serviços contidos em services.
stop(String,String)	Finaliza a execução do serviço identificado pelo host e fila passado como parâmetro.
stopService	Finaliza a execução de todos os serviços contidos em services.

Quadro 12 – Atributos e métodos da classe `MonitorReceiver`

A classe `BrokerCommand` é responsável por executar a transação delegada pelos serviços receptores de mensagens. Executando a transação com sucesso, é disponibilizada a mensagem para as filas interessadas na transação, conforme parametrizado na base de dados. O Quadro 13 indica a estrutura da classe `BrokerCommand`.

BrokerCommand	
Atributos	**Descrição**
static QueryJmsUtil query	Representa a classe que executa as consultas das tabelas do modelo de negocio do framework.
Métodos	**Descrição**
execute(Map, String, String, boolean)	Executa a transação passada como parâmetro.
forward(Map, String, String)	Disponibiliza a mensagem para as filas interessadas.
generateLogReplication(Map, Exception, String, String)	Gera log da execução da transação, contendo o estado de sucesso ou erro.
getQueryJmsUtil	Retorna a instância de QueryJmsUtil.

Quadro 13 – Atributos e métodos da classe `BrokerCommand`

A classe `OpenJmsServer` implementa a interface `Server`, tem a responsabilidade de iniciar o *provider* OpenJms e possui métodos para interagir com as classes administrativa do *provider*. A partir desta classe é possível criar uma nova fila de mensagem, verificar a existência de alguma fila específica, retornar a quantidade de mensagens pendentes a serem consumidas e iniciar os serviços necessários para o funcionamento do OpenJms. O Quadro 14

indica a estrutura da classe `OpenJmsServer`.

OpenJmsServer	
Atributos	**Descrição**
JmsAdminServerIfc adminOpenJms	Classe disponibilizada pelo OpenJms para administrar o
Configuration config	Classe disponibilizada pelo OpenJms para carregar a configurações do provider.
Services services	Classe disponibilizada pelo OpenJms para registrar e iniciar os serviços do provider.
Métodos	**Descrição**
createQueue(String)	Cria uma nova fila no provider.
existsQueue(String)	Verifica se existe o nome da fila passada como parâmetro
getCountMessageQueue(String)	Retorna a quantidade de mensagens pendentes na fila passada como parâmetro.
init	Inicia as configurações do OpenJms.
removeQueue(String)	Remove uma fila do provider.
startService	Inicia todos os serviços do OpenJms que foram registrados.
stopService	Finaliza todos os serviços do OpenJms.

Quadro 14 – Atributos e métodos da classe `OpenJmsServer`

Para manter os cadastros de filas e serviços foi definido um modelo de negócio para a camada de replicação. Este modelo possui suas classes mapeadas com o Hibernate caracterizando cada classe como POJO. A Figura 18 ilustra o diagrama de classes que representa este modelo de negócio.

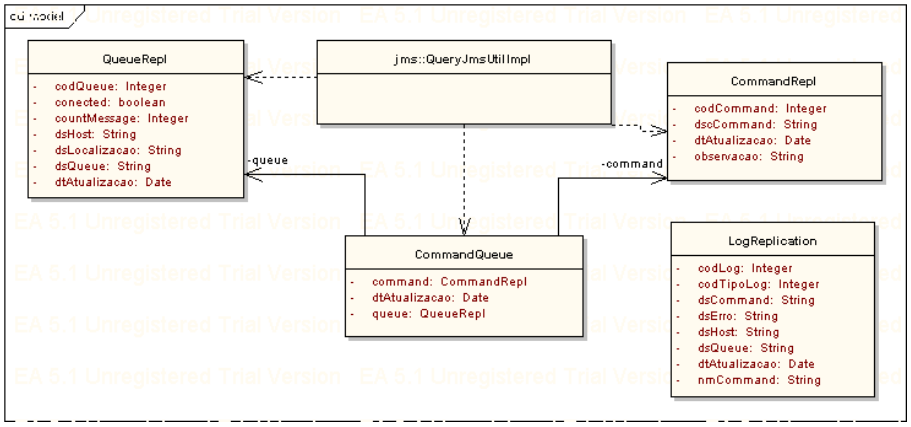


Figura 18 – Diagrama de classes do modelo de negócio da camada de replicação

A classe `QueueRepl` representa uma fila e um serviços receptor de mensagem cadastrado no banco de dados.

A classe `CommandRepl` representa um cadastro de transação no banco de dados. Os

nomes de transação que esta classe mapeia devem ser idênticos, aos nomes utilizados pela aplicação que deseja usufruir da replicação de dados. Conforme já explicado anteriormente, sempre que iniciada uma nova transação, dever ser passado o nome da transação caso queira que seja interceptada e replicada.

A classe `CommandQueue` representa a associação entre a fila e a transação cadastrada. É utilizada pela classe `BrokerCommand` para filtrar quais filas possuem interesse na transação.

A classe `LogReplication` representa os *logs* da execução das transações nos *sites* de destino, mantendo como histórico os erros ocorridos na execução.

A classe `QueryJmsUtilImpl` é uma implementação da interface `QueryJmsUtil` que contêm os métodos que executam consultas nas tabelas do modelo negócio da camada de replicação do *framework*.

3.2.3.3 Diagrama de classes do gerenciador de replicação

A Figura 19 ilustra o diagrama de classes que representa a aplicação *web* responsável pelo gerenciamento da replicação de dados.

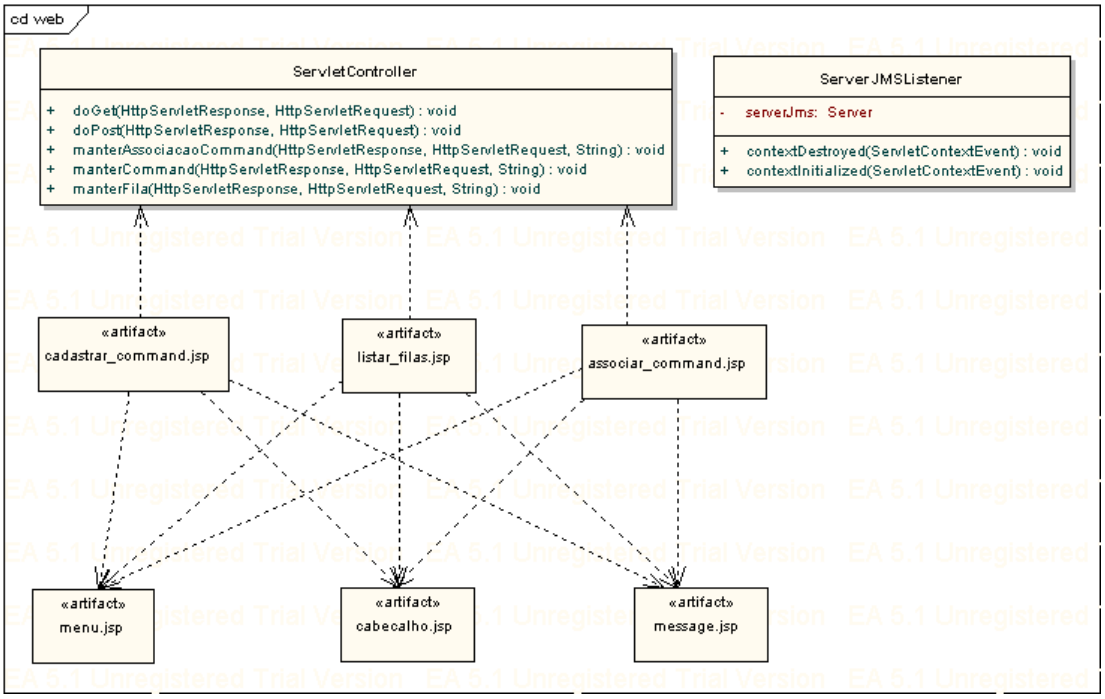


Figura 19 – Diagrama de classes da aplicação web

A classe `ServletController` é responsável por receber as requisições das páginas *Java Server Pages* (JSP) e encaminhar para o método específico da ação requisitada pela página. Esta classe estende a classe `HttpServlet` do pacote J2EE que controla todas as requisições das páginas, sendo o único ponto de acesso ao contexto da página contida no container *web* gerenciado pelo Tomcat.

A classe `ServerJmsListener` possui apenas dois métodos implementados. O método `contextInitialized()` é chamado sempre que o container *web* é iniciado e `contextDestroyed()` é chamado sempre que o container *web* é finalizado. Desta forma foi definido que o melhor ponto para iniciar e parar os serviços de replicação seria nesta classe, não dependendo de uma ação do usuário após já ter iniciado ou parado o próprio container.

Foi definida alguns fragmentos de páginas padrões `menu.jsp`, `cabecalho.jsp` e `mensagem.jsp` para serem utilizadas no desenvolvimento das páginas, responsáveis por manter o modelo de negócio da camada de replicação, representando as configurações das filas de mensagens e também os filtros de replicação. Estes fragmentos possuem informações do menu, cabeçalho e rodapé podendo ser reutilizadas no desenvolvimento das outras páginas da aplicação.

O arquivo `cadastrar_command.jsp` é responsável por manter os registros contidos na tabela `COMMAND` mapeada no POJO `CommandRepl`, podendo ser informado os nomes das transações que poderão ser replicadas entre os participantes .

O arquivo `listar_filas.jsp` é reponsável por manter os registros contidos na tabela `QUEUE` mapeada no POJO `QueueRepl`. Através desta página é possível monitorar as filas de mensagens, identificando o status da conectividade com o serviço receptor, e quantidade de mensagens que estão pendentes para serem consumidas.

O arquivo `associar_command.jsp` é responsável por manter os registros da tabela `COMMAND_QUEUE` mapeada no POJO `CommandQueue`. Através desta página é informado quais transações uma determinada fila terá interesse em receber, caracterizando os filtros de replicação.

3.3 IMPLEMENTAÇÃO

Os assuntos a seguir descrevem as ferramentas e técnicas utilizadas durante o

desenvolvimento *framework*.

Para o desenvolvimento do *framework* foi utilizada a plataforma Eclipse, que não é uma ferramenta específica de uma determinada linguagem. Possui recursos adicionais que permitem que a ferramenta seja estendida dependendo da linguagem que está sendo utilizada. Estes recursos são chamados de *plugins* que especializam a plataforma para determinadas tarefas ou linguagens. Não foi necessário utilizar outros recursos além daqueles que a própria plataforma disponibiliza da linguagem Java, destacando-se compilação, depuração de código e execução.

Como servidor de aplicação Java para *web* foi utilizado o Tomcat, para disponibilizar as interfaces desenvolvidas para gerenciamento da replicação.

O Tomcat é distribuído como software livre e desenvolvido como código aberto dentro do projeto Apache Jakarta. É utilizado como a implementação de referência para as tecnologias JSP. Pode ser considerado robusto e eficiente para ser utilizado mesmo em um ambiente de produção. Tecnicamente cobre parte da especificação J2EE com tecnologias como Servlet, JSP e outras consideradas de apoio.

O OpenJMS é um provider JMS *open source* que implementa a especificação da versão 1.0.2 da API JMS definida pela Sun Microsystem´s. Foi utilizado no framework para gerenciar as filas de mensagens sendo também a ferramenta para testes de envio e recebimento de mensagens. Foi escolhido por ser um software *open source* de tamanho relativamente pequeno mas que atendeu os requisitos do framework como:

a) domínio de mensagens point-to-point;
b) garantia de entrega de mensagens;
c) suporte a um grande número de filas.

Este *provider* é disponibilizado de forma que não é necessário criar qualquer outra estrutura para iniciar seu processo. Ao realizar o *download* do software disponível em (OPENJMS, 2005) existem arquivos *bat* que iniciam todos os serviços do *provider*, mantendo o mesmo em uma única instância de *Java Virtual Machine* (JVM) particular.

Para simplificar o início de todos os serviços agregados a replicação, foi estudada a forma em que era iniciado OpenJMS, para poder ser iniciado juntamente com o Tomcat, mantendo as duas aplicações rodando na mesma JVM.

Após ter agregado o conhecimento do código necessário para iniciar o *provider*, foi implementado a classe `OpenJmsServer` utilizando a própria API do OpenJMS, que simplificou o início do serviços necessário para replicação de dados do *framework*. Foi utilizada na classe `ServerJMSListener` que possui um método que sempre é executado

quando o Tomcat é iniciado. No Apêndice A é demonstrada a classe `OpenJmsServer` contendo o código necessário para iniciar o *provider* OpenJMS.

Para realizar a leitura dos arquivos de configuração do *framework* foi utilizada a API JDOM que é a implementação da API DOM disponibilizada pela linguagem Java. Esta API fornece interfaces para manipulação de arquivos XML, sendo utilizada para ler o valor de atributos das *tags* contidas nos arquivos de configuração.

Esta API possui recursos de validação do arquivo XML, caso não esteja no formato padrão da linguagem, será lançado uma exceção ao carregar o arquivo. O Quadro 15 indica parte da leitura do arquivo dao.xml de configuração do *framework*. Caso não esteja no formato correto, será propagada a exceção `BuildConfigurationException` da própria aplicação.

```
Document d = null;
 try{
      SAXBuilder sb = new SAXBuilder();
      d = sb.build(in);
      n.close();
       }catch(Exception e){
            throw                                          new
BuildConfigurationException("\\conf\\dao.xml          Erro:
Formato    do arquivo inválido.");
      }
      Element eldaos = d.getRootElement();
      List lista = eldaos.getChildren();
      Iterator it = lista.iterator();
      ...
```

Quadro 15 – Leitura de um arquivo XML de configuração

Durante o desenvolvimento percebeu-se que seria importante utilizar uma técnica que facilitasse a geração de *logs* do *framework* , para tornar a gravação das mensagens de *logs* configuráveis desde o local de gravação até quais classe poderiam gerar *log*. Dependendo do fluxo e quantidade de transações que esteja sendo manipulado pela replicação, pode poluir a gravação dos *logs* com outras mensagens que não possuem importância, quando é preciso depurar alguma funcionalidade em específico.

Partindo deste problema, a API log4j apresentou todas as recursos necessários para o *framework*. A partir de um arquivo XML é possível configurar quais classes deverão gerar *logs* e também em quais situações, podendo citar erro, depuração ou informação. Para cada classe é possível gerar uma saída diferente, desde a gravação no próprio console até em arquivos textos separados. Para sua utilização, foi necessário carregar suas configurações e também utilizar as classes disponíveis para geração de *log*. O Quadro 16 indica o modo utilizado na classe `Config` para gerar os *logs* com o log4j.

```
Logger.getLogger(Config.class).info("Iniciando
configuração do framework com base no arquivo
config.xml")
```

Quadro 16 – Geração de log a partir da API log4j

Na camada de persistência foi empregado o padrão de projetos DAO pela necessidade visível que existe nos objetos composto de atributos e comportamento, ter um ponto de acesso único no que diz respeito a persistência de dados. Segundo Alberto (2006, p.1) este padrão de projetos traz algumas vantagens já comprovadas em outras aplicações, conforme citado abaixo:

a) aumento de transparência: A aplicação não precisa saber que banco de dados usa, nem como os dados são montados, visto que você passa a trabalhar apenas com objetos;
b) facilidade de migração: Os dados podem vir de qualquer lugar, como um XML, *webservice* ou um banco de dados. A migração para outras fontes basta apenas alterar o DAO e a aplicação passará a utilizar o novo modelo de dados;
c) reduz a complexidade na aplicação: Se você não vai trabalhar com `recordsets` da API JDBC ou diretamente com instruções SQL, o código fica mais legível e mais fácil de programar;
d) centralização de acesso em uma camada separada: Com o *pattern* DAO, você centraliza todo o acesso no DAO, facilitando a manutenção da aplicação.

O diagrama de seqüência representado na Figura 20 demonstra a troca de mensagens entre as classes que compõe esta camada, desde o início de uma transação até o momento em que ela é finalizada e disponibilizada uma mensagem na camada de replicação.

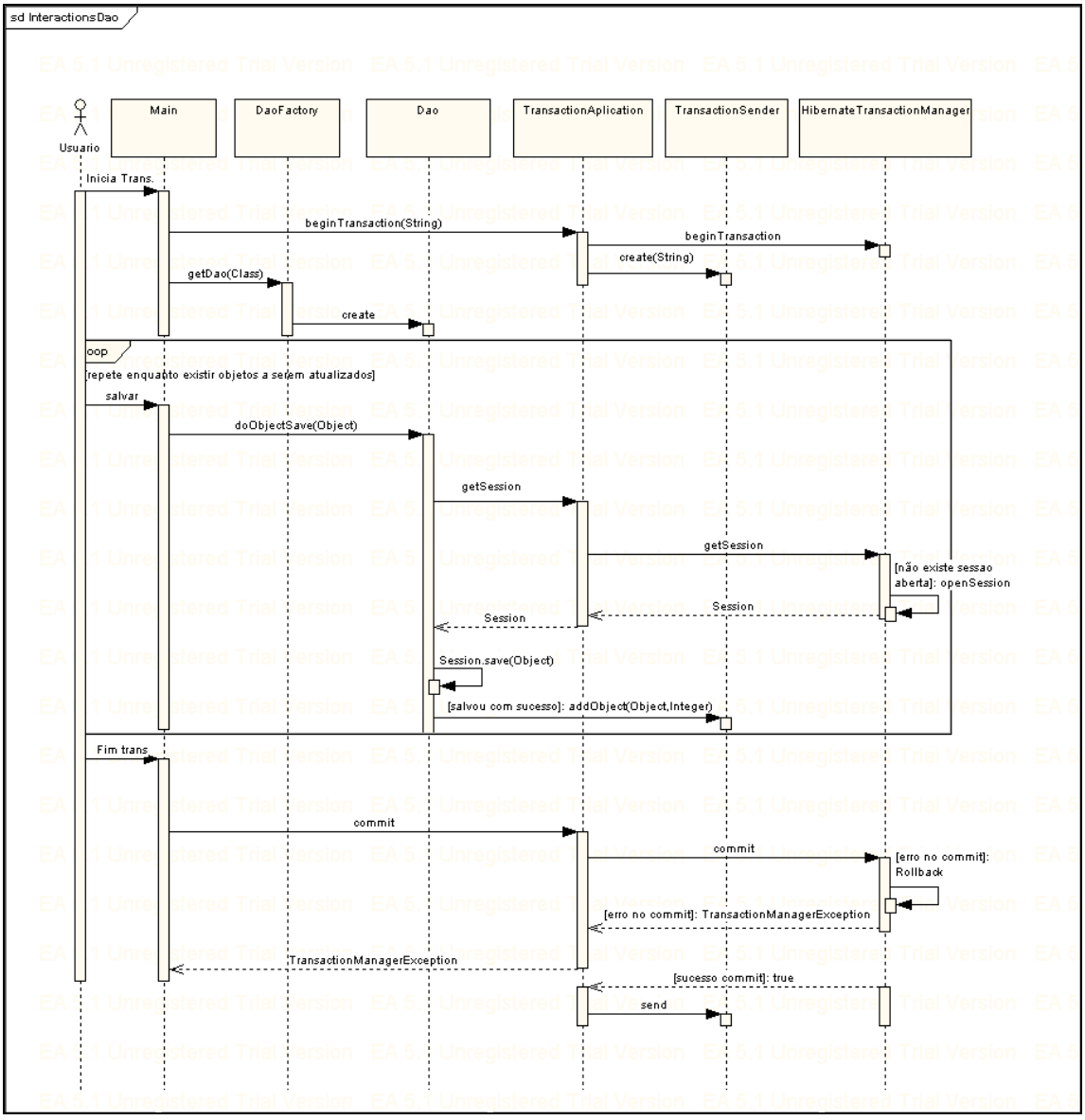


Figura 20 – Diagrama de seqüência da camada de persistência

Inicialmente foi criada uma interface padrão chamado `Dao` contendo os métodos básicos para interagir com os objetos de negócio. Também foi criada outra interface `TransactionManager` responsável por gerenciar a conexão, sessões e transações do banco de dados. Portanto apenas é necessário implementar estas interfaces conforme a tecnologia de persistência adotada. Basicamente as implementação de `Dao` trocará mensagens com a classe `TransactionAplication` que irá delegar suas requisições para a implementação de `TransactionManager` obtendo a sessão com o banco de dados para persistir os objetos.

Foi apresentado no diagrama apenas o método `doObjectSave(Object)` no laço de repetição que ocorre logo após o início da transação e criação de um objeto `Dao`, este método

pode variar também nas chamadas de `doObjectUpdate(Object)` e `doObjectDelete(Object)` possuindo a mesma troca de mensagens com as outras classes que foram apresentadas.

Como foi adotada a tecnologia Hibernate para manipular a base de dados, utilizou-se as classes disponibilizada pela API na implementação de `HibernateDao` que realizou a interface `Dao`. No Apêndice B é demonstrada a implementação desta classe.

Durante a transação pode ser utilizado diretamente as classes `Dao` configuradas no arquivo dao.xml. Este arquivo possui o mapeamento entre a classe POJO com a implementação de `Dao` responsável pela sua manipulação com a tecnologia de persistência. Desta forma o desenvolvedor não precisa saber qual classe `Dao` manipula o objeto que está sendo mantido, utiliza diretamente uma fábrica de `Dao` chamada `DaoFactory`. Chamado o método estático `getDao(Class)` é retornado exatamente a instância da classe que está mapeada da classe POJO passada como parâmetro. Caso não existir a classe no arquivo, será utilizada a classe configurada como `default`.

Para gerenciamento das sessões e transação de banco de dados no Hibernate, foi implementada a classe `HibernateTransactionManager` que realizou a interface `TransactionManager`. No Apêndice C é demonstrada a implementação desta classe.

Devido o alto custo que existe em abrir múltiplas conexões com o banco e também carregar configurações da tecnologia de persistência, percebeu-se que seria importante compartilhar somente uma instância da implementação de `TransactionManager`. Como esta instância teria que suportar múltiplas `threads` disputando processador para executar as transações, teve a necessidade de criar a classe `TransactionAplication` com a responsabilidade de sincronizar e delegar as requisições da implementação de `TransactionManager`. Nesta classe também foi encapsulado o escopo de replicação, podendo ser abstraída esta função na adoção de outra tecnologia de persistência, preocupando-se somente com a realização da interface `TransactionManager`. O Quadro 17 indica o modo utilizado na classe `TransactionAplication` para sincronizar as requisições de `HibernateTransactionManager`.

```
//Interface que mapeia a implementação da tecnologia de persistência
do framework
private static TransactionManager transaction;
//Classe que representa o escopo de replicação
private static TransactionSender txsender;

public static void beginTransaction(String name) throws
TransactionManagerException{
     //Sincroniza a requisições dos serviços receptores de mensagens
     synchronized(TransactionManager.class){
     try{
         transaction.beginTransaction();
         if (name != null)
             txsender = new TransactionSender(name);
         }catch(TransactionSenderException ex){
              throw new TransactionManagerException("Erro ao criar o
escopo de replicação.",ex);
          }
     }
}
```

Quadro 17 – Sincronização das requisições de HibernateTransactionManager

Havendo necessidade de alterar futuramente a tecnologia de persistência, não será necessário alterar o *framework*, pois são completamente configuráveis as implementações das interfaces de persistência de dados.

Na camada de replicação foi empregado o padrão de projetos *Singleton*. Este padrão garante que somente um objeto exista independente do número de requisições que receber para criá-lo. Realmente este padrão já foi utilizado de certa forma na camada de persistência no gerenciamento de transações, sendo evidenciado com enfoque maior na camada de replicação.

O diagrama de seqüência representado na Figura 21 demonstra a continuação da troca de mensagens realizada entre a camada de persistência e a camada de replicação, presumindo que a transação foi executada com sucesso no SGBD de origem e foi iniciado o processo de replicação a partir da classe `TransactionAplication` conforme já demonstrado na Figura 20.

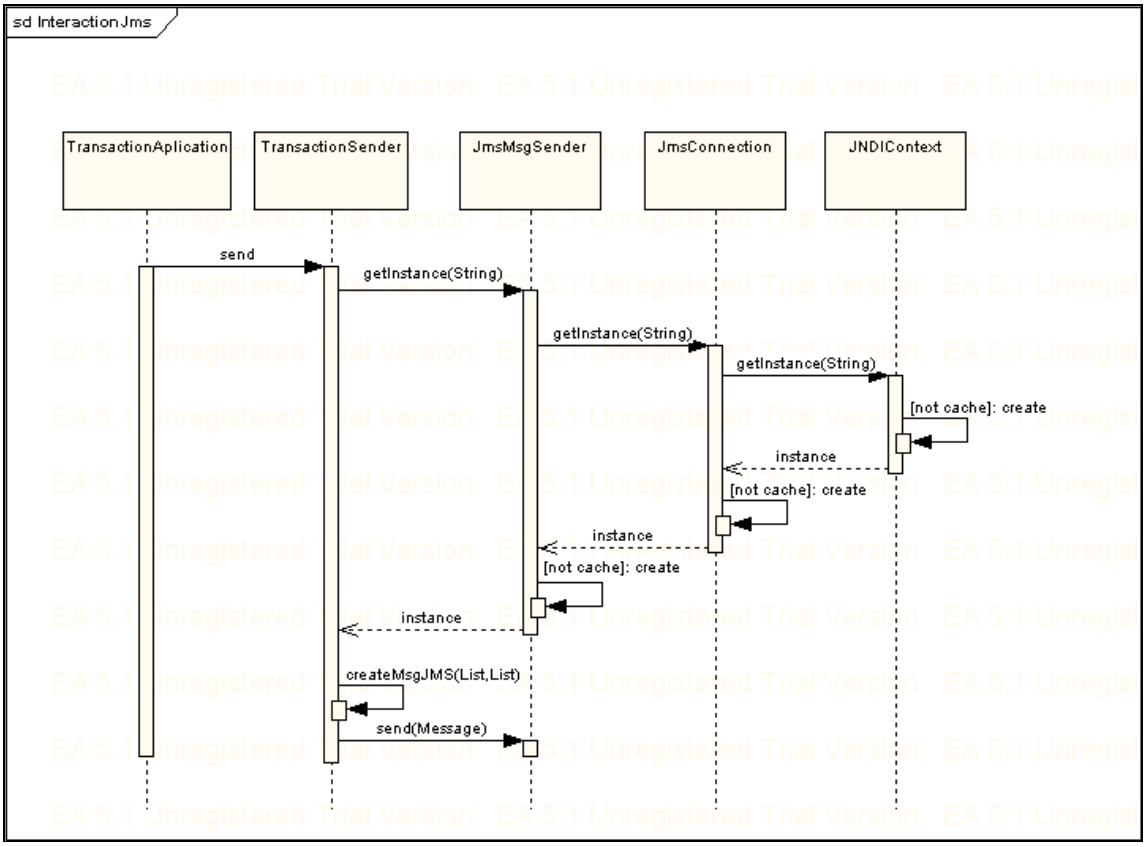


Figura 21 – Diagrama de seqüência da camada de replicação

Devido a problemas de performance e também sobrecarga das conexões com o *provider*, teve a necessidade de restringir que várias conexões com o mesmo *provider* fossem abertas simultaneamente. Partindo deste problema, o modificador dos construtores das classes que realizam a conexão foi alterado para privado. Dentro da própria classe de conexão foi implementado um método estático que retorna uma instância com o *provider* requisitado, armazenando em uma lista que caracterizou um *cache* de conexões. Sempre que algum outro processo solicitar conexão com algum *provider* em específico e já estiver contido no *cache*, será retornada a instância armazenada, criando assim um compartilhamento de conexão entre processos. O Quadro 18 indica o modo utilizado para restringir as conexões com os *providers* da classe `JmsConnection`.

```
public static JmsConnection getInstance(String host) throws
JMSException,CommunicationException,Exception{
     //Sincroniza a requisições dos serviços receptores de
//mensagens
     synchronized(JmsConnection.class){
         //Busca a conexão no cache
         JmsConnection con = (JmsConnection)connections.get(host);
         //Conexão não existe em cache?
         if (con == null){
                //Cria uma nova conexão
               con = new JmsConnection();
               InitialContext jndi = JNDIContext.getContext(host);
               if(jndi == null)
                    throw new CommunicationException();
                con.factory= (QueueConnectionFactory)jndi.lookup
               (Config.getInstance().getNameConectionFactory());
                con.connection=con.factory.createQueueConnection();
               connections.put(host,con);
           }
     return con;
}}...
```

Quadro 18 – Abertura de conexões da classe JmsConnection

Os serviços receptores de mensagem foram implementados de tal forma para garantir a execução, mesmo que não exista uma conexão estabelecida com o *provider*, garantindo modo assíncrono de consumo de mensagens. Os servidores tanto local como central não precisam estar conectados na rede para que o serviço de consumos de mensagens seja iniciado. A própria classe do serviço de consumo, se encarrega de restabelecer a conexão no momento em que o servidor conectar na rede ou iniciar o *provider*.

Foram implementadas configurações para os serviços de consumo, referente ao tempo de espera entre processamento, tempo de re-conexão, tempo tolerante do consumo (*time-out*) e também quantidade máxima de mensagens a serem consumidas em cada iteração da `Thread`. Estas configurações foram criadas para distribuir de forma igualitária o direito de processamento, não deixando algum serviço no estado de espera demasiado. No Apêndice D é mostrado o método `run()` implementado na classe `JmsMsgReceiver` realizando a interface `Runnable` responsável pela execução da `Thread`. Esta classe também foi nomeada de serviço receptor de mensagem.

A Figura 22 demonstra o diagrama de seqüência da execução de uma transação que foi consumida de uma determinada fila pelo serviço receptor de mensagem.

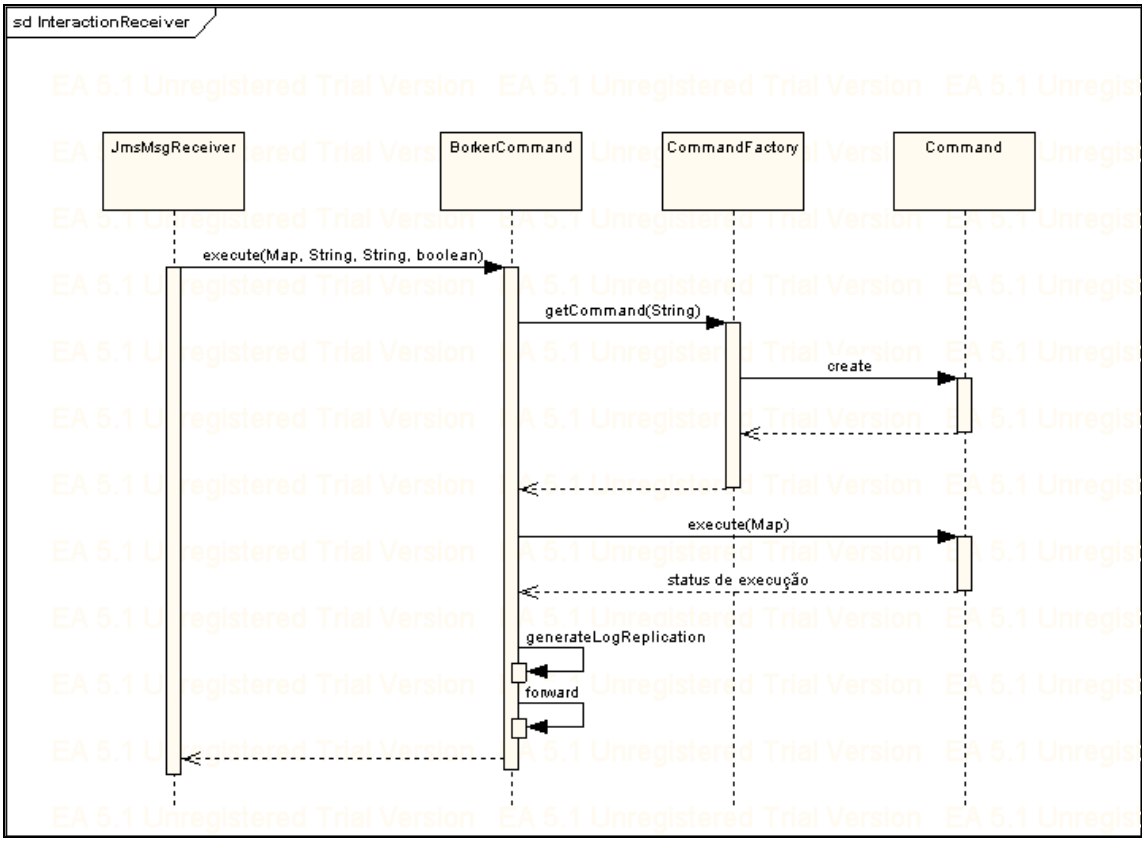


Figura 22 – Diagrama de seqüência da execução de uma transação replicada

Os serviços delegam a execução da transação para `BrokerCommand`. Esta classe é responsável por criar uma instância da classe `Command` conforme transação empacotada na mensagem. Toda transação pode optar por utilizar o modo padrão de execução de transação que respeita a seqüência em que os comandos foram aplicados, ou utilizar um modo específico de executar a transação. As implementações de `Command` relacionadas com o nome da transação são configuradas no arquivo command.xml. São instanciadas através da classe `CommandFactory` através método estático `getCommand(String)` que retorna o objeto da transação passada como parâmetro. Caso não existir o nome da transação no arquivo, será utilizada a classe configurada como `default`. A partir do momento em que existir uma instância de qualquer implementação da interface `Command` é chamado o método `execute(Map)` para executar a transação conforme definido. Não importando o status da execução é mantido um log de replicação em uma tabela de banco de dados contendo mensagem de erro ou sucesso da execução do `Command`. Retornando sucesso na execução, a mensagem é disponibilizada para todas filas que possuem interesse na mesma transação através do método `forward()`.

3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Para demonstrar a operacionalidade do *framework*, inicialmente foi disponibilizada uma arquitetura de rede, caracterizando uma rede privada com dois endereços de rede distintos. Tendo a arquitetura montada, foi classificado o servidor central como 192.168.226.1 e local como 192.168.226.2 para que pudesse demonstrar um caso real de replicação de dados entre dois servidores. Foi definido um simples modelo de dados contendo duas tabelas, USUARIO e GRUPO_USUARIO e um programa executado em modo console, que se beneficia dos recursos do *framework* para replicar os dados incluídos nas tabelas citadas. A Figura 23 demonstra um exemplo de arquitetura da replicação que será detalhada no decorrer desta sessão.

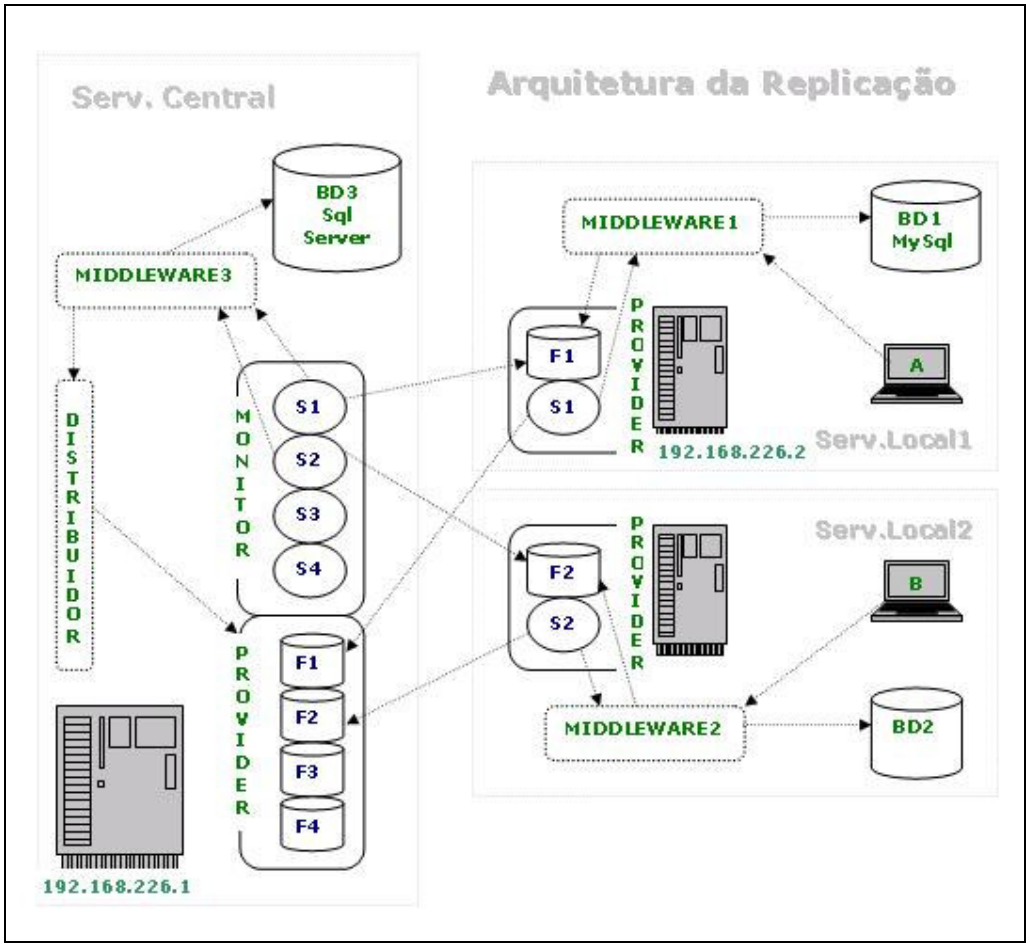


Figura 23 – Exemplo de arquitetura dos componentes participantes da replicação

Tendo em vista que um dos requisitos se tratava da heterogeneidade dos bancos de dados participantes da replicação, utilizou-se o MySQL no servidor local e MsSQL Server no

servidor central. Como padrão, foi mantido a base de dados Derby para persistência das mensagens, disponibilizada juntamente com o *provider* OpenJMS.

Baseando-se pelo esquema apresentado na Figura 22, no momento que o computador "A" finaliza uma transação, são realizadas as seguintes operações com o lote de transação interceptado:

a) computador "A" envia comando "finalizar transação", para "middleware1";
b) "middleware1" aplica comando "*commit*" na base de dados "BD1";
c) "middleware1" intercepta o lote da transação e cria mensagem JMS contendo o lote de comandos e objetos da transação;
d) "middleware1" disponibiliza mensagem na fila "F1" do *provider* local;
e) serviço "S1" do servidor central, consome a mensagem disponibilizada na fila do *provider* local e envia para "middleware3";
f) "middleware3" aplica o comando "*commit*" na base de dados "BD3" referente a transação empacotada na mensagem;
g) "middleware3" envia mensagem empacotada para distribuidor;
h) distribuidor disponibiliza a mensagem nas filas do *provider* central que possuem interesse na transação.

Para demonstrar estes passos foram realizadas as configurações necessárias do *framework* para cada servidor, iniciando os componentes necessários e também cadastrando as filas, nomes de transação e vínculo da fila e transação pelo gerenciador *web* de replicação central.

O primeiro passo é informar os parâmetros de configuração nos arquivos config.xml do servidor central conforme representado no Quadro 19.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
      <Persistence>
            <Hibernate file="hibernate.cfg.mssql.xml">
            </Hibernate>

            <TransactionManager  class="br.com.replication.dao.
            hibernate.HibernateTransactionManager">
            </TransactionManager>
      </Persistence>
      <Provider local="192.168.226.1" master="192.168.226.1">
            <QueueLocal name=""></QueueLocal>

            <ContextJNDI file="OpenJmsJndi.properties">
            </ContextJNDI>

            <QueryJMSUtil  class="br.com.replication.jms.server.
            QueryJmsUtilImpl">
            </QueryJMSUtil>

            <OpenJms derbyHome="C:\openjms-0.7.7-
            alpha-3\db" user="admin" password="openjms">
            </OpenJms>

            <ConnectionFactory name="ConnectionFactory">
            </ConnectionFactory>
      </Provider>

      <ServiceReceiver timeSleep="5000"
          toleranteTimeOut="3000" toleranceReconnect="20000"
          qtdMsgsFromIteration="20">
      </ServiceReceiver>
</Configuration>
```

Quadro 19 – Dados do arquivo config.xml de configuração do *framework*

As configurações foram separadas por *tags* entre as camadas de persistência e replicação e serviço receptor de mensagem. O Quadro 20 é apresenta o significado das *tags* presente no arquivo config.xml.

Nome da Tag	Conceito / Significado
Persitence	Camada de persistência.
Hibernate.file	Arquivo que contêm as configurações básicas do Hibernate.
TransactionManager.class	Classe que define a implementação de TransactionManager.
Provider	Camada de replicação
Provider.local	Endereço de rede do provider local.
Provider.master	Endereço de rede do provider central.
QueueLocal.name	Nome da fila definida como local, não obrigatório no provider central.
ContextJNDI.file	Nome do modelo utilizado pata conexão ao serviço de nomes do provider.
QueryJMSUtil.class	Classe que define a implementação de QueryJMSUtill. Esta classe deve possuir os comandos de consulta das tabelas do modelo de negócio da camada de replicação.
OpenJMS.derbyHome	*Path* do banco de dados Derby.
OpenJMS.user	Nome do usuário para conexão com o OpenJMS
OpenJMS.password	Senha do usuário para conexão com o OpenJMS
ConnectioFactory.name	Nome da fábrica de conexão com o provider.
ServiceReceiver.timeSleep	Tempo de espera entre iteração do serviço receptor de mensagem.
ServiceReceiver.toleranceTimeOut	Tempo de tolerância para consumo do serviço receptor de mensagem.
ServiceReceiver.toleranceReconnect	Tempo de tolerância para reconexão.
ServiceReceiver.qtdsMsgsFromIteration	Quantidade máxima de mensagens consumidas por iteração do serviço.

Quadro 20 – Significado das *tags* contidas no arquivo config.xml

Para o servidor local somente foi alterado o nome do arquivo de configuração do Hibernate, informando outro especificamente para o servidor MySQL. Também foi alterado a *tag* `Provider.local` para o endereço 192.168.226.2 e `QueueLocal.name` para o nome fila02. Os dados informados nos arquivos de configuração do Hibernate serão omitidos pois foram mantidas as configurações padrões da tecnologia.

O segundo passo foi iniciar os servidores de aplicação Tomcat do servidor central e local. Como o nome da fila “fila02” ainda não foi cadastrada e registrada no servidor central, o serviço receptor de mensagens do servidor local permanece no estado desconectado conforme representado na Figura 24 do arquivo de *log* do *framework*.

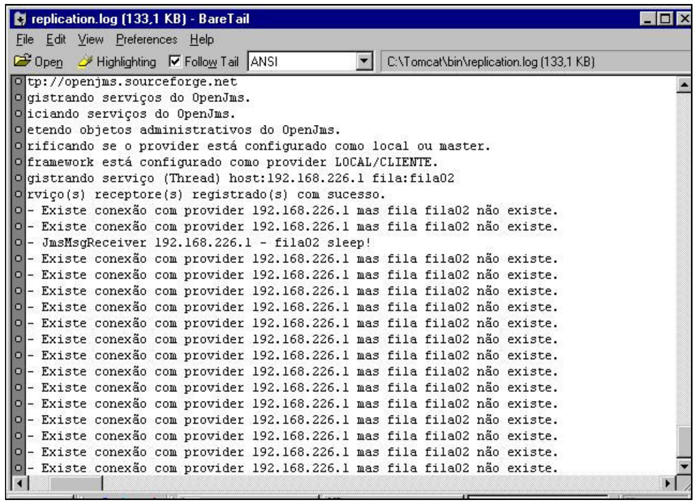


Figura 24 – Serviço local de recepção de mensagens desconectado

O arquivo de *log* é definido nas configurações de log4j.xml que possui uma *tag* `appender` contendo o nome “replication” e local do arquivo onde serão gerados os *logs* do *framework*.

O terceiro passo é cadastrar a fila no servidor central, para assim restabelecer a conexão entre o servidor local e central. Este processo também registra outro serviço no servidor central que consome as mensagens da fila do servidor local, essencial para que ocorra o processo de replicação. Após ter realizado o cadastro, o serviço local restabeleceu a conexão automaticamente conforme representando na Figura 25 do arquivo de *log* do *framework*.

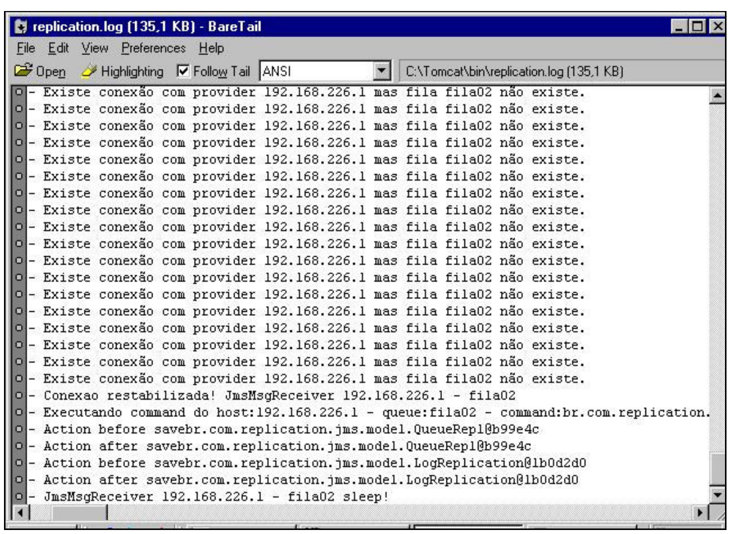


Figura 25 – Conexão restabilizada após cadastro da fila de mensagens no Servidor central

Com os cadastros realizados e também as conexões entre os servidores restabilizadas, o gerenciador exibe o serviço com status *on-line*. Para exemplificar, foram cadastradas outras

filas para endereços de rede distintos, mantendo os mesmos desconectados com mensagem pendente de consumo referente a transação de cadastro da fila. A Figura 26 representa o gerenciador exibindo os serviços com o status apropriado e também as mensagens pendentes de consumo.

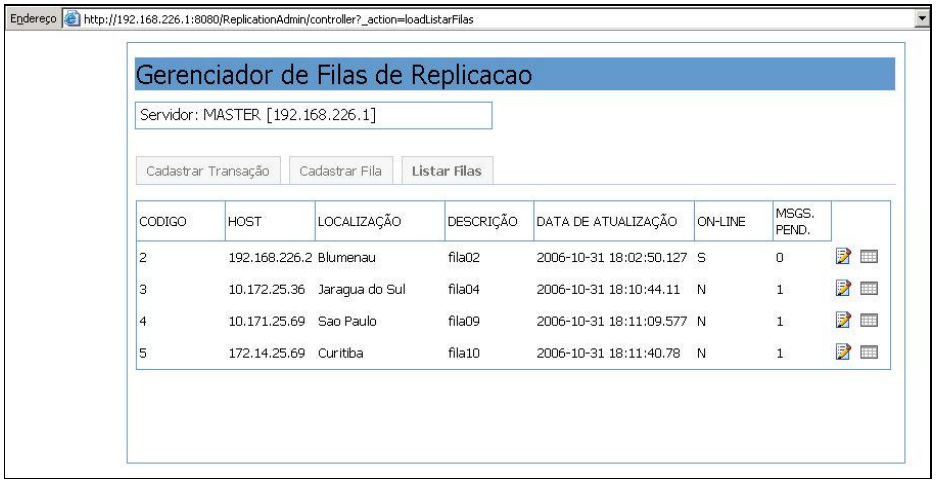


Figura 26 – Gerenciador central exibindo serviço *on-line* e *off-line* com mensagem pendente

O serviço referente ao endereço 192.168.226.2 está com quantidade de mensagens pendentes igual a zero, pois o serviço local já consumiu a mensagem referente a fila cadastrada. A partir do momento que o cadastro de fila é replicado para o servidor local, é possível gerenciar a fila localmente conforme demonstrado na Figura 27.

Figura 27 – Gerenciador local exibindo serviço on-line

O quarto passo refere-se o cadastro da transação que será utilizado para manipular os dados do modelo definido para testes. Para manipular o modelo, realizou-se o cadastro da transação com nome “manterUsuario” conforme demonstrado na Figura 28.

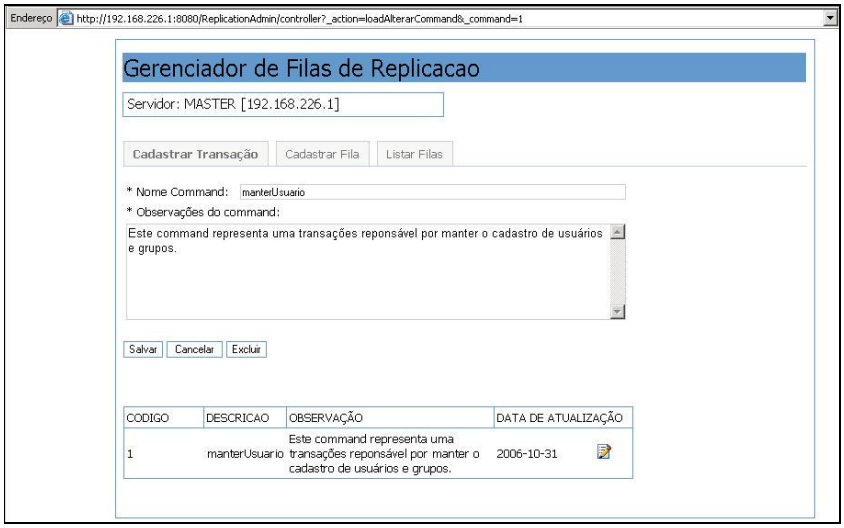


Figura 28 – Cadastro da transação “manterUsuario”

O quinto passo é vincular a transação cadastrada com a fila de mensagem do servidor local referente endereço de rede 192.168.226.2, para que possam ser replicados os dados do modelo de dados definido para testes. A Figura 29 demonstra o vínculo realizado entre a transação e a fila de mensagem.

Figura 29 – Vinculação da transação com a fila de mensagem

Todas estas operações de cadastros realizadas no gerenciador central são devidamente replicadas para as filas de mensagens. Desta forma, somente cabe ao administrador central realizar os cadastros, retirando esta responsabilidade do administrador do servidor local. A Figura 30 demonstra a réplica do vínculo de transação e fila no gerenciador de replicação local.

Figura 30 – Vínculo replicado sendo exibido no gerenciador de replicação local

Com todos os cadastros gerenciais realizados e devidamente replicados, neste momento será exemplificada uma aplicação que se beneficia do *framework* utilizando o modelo de dados definido para testes. Para melhor entendimento da aplicação, será detalhada a implementação realizada na aplicação de testes, levando em consideração a utilização do *framework*.

No início da aplicação são carregadas as configurações conforme demonstrado no Quadro21.

```
//Carregar configurações contidas em todos
//os arquivos XML da aplicação
Config.configure();
//Criar e iniciar configurações da
//instancia de TransactionManager
TransactionAplication.init();
```

Quadro 21 – Carregar configurações e iniciar gerenciador de transação

Para iniciar uma nova transação somente é necessário executar o comando `TransactionAplication.beginTransaction(String)` passando o nome da transação como parâmetro. Para finalizar deve ser chamado `TransactionAplication.commit()`.

Nesta aplicação foram utilizadas as classes `Dao` e `Command` disponibilizadas por padrão nas configurações do *framework*. O Quadro 22 demonstra a utilização da classe `Dao` para manipular um objeto da classe `Usuario`.

```
Usuario user = null;
Dao dao = DaoFactory.getDao(Usuario.class);
user = (Usuario)dao.findByPrimaryKey((Object)cod);
if (user != null){
      System.out.println("Usuario ja existe.");
}else{
      user = new Usuario();
      user.setCodUsuario(cod);
      user.setNomeUsuario(desc);
      dao.doObjectSave(user);
}
```

Quadro 22 - Utilização da classe `Dao` para manipular um objeto da classe `Usuario`

Sabendo como iniciar as configurações do *framework* e também a forma de atualizar os objetos, serão agora apresentadas as telas dos testes realizados com a aplicação. A Figura 31 apresenta o início da execução da classe `TesteReplica`.

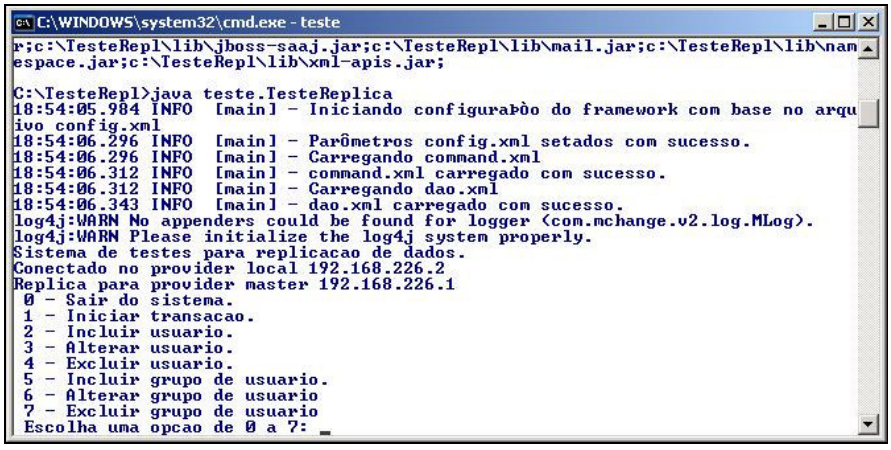


Figura 31 - Início da execução da classe `TesteReplica`

Neste momento é iniciada a transação com o nome “manterUsuario” que cria o escopo de replicação pois existe o vínculo com a fila local cadastrada. A Figura 32 exibe o início da transação.

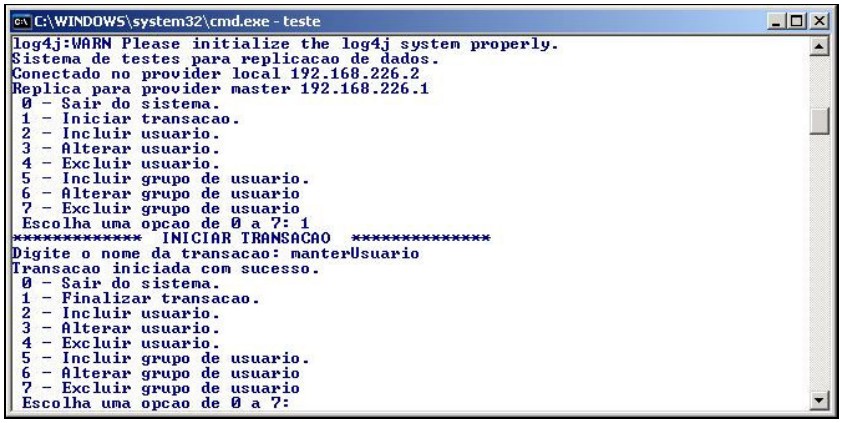


Figura 32 – Iniciando transação pela classe `TesteReplica`

Com a transação iniciada com escopo de replicação, é realizado o cadastro de dois registros de USUARIO e um registro de GRUPO_USUARIO. A Figura 33 exibe o cadastro dos registros de USUARIO pela classe `TesteReplica`.

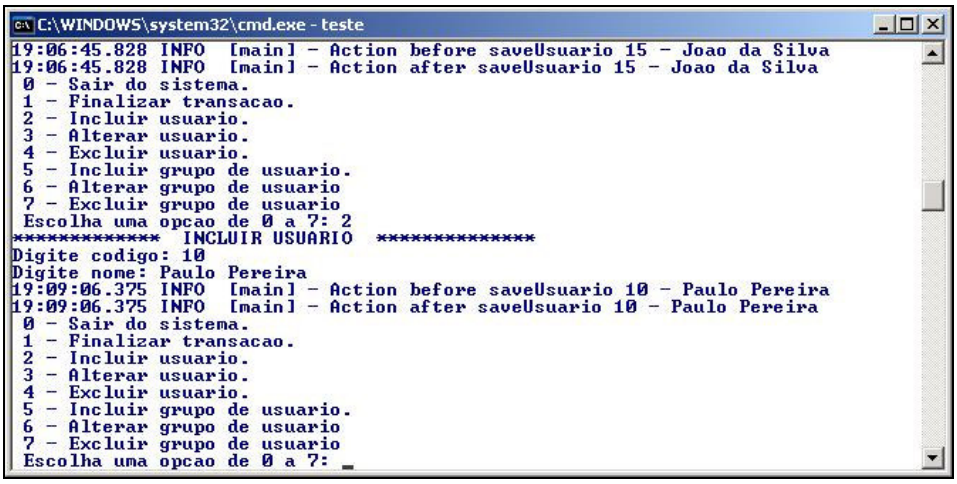


Figura 33 – Inclusão de USUARIO pela classe `TesteReplica`

A Figura 34 exibe o cadastro do registro de GRUPO_USUARIO pela classe `TesteReplica`.

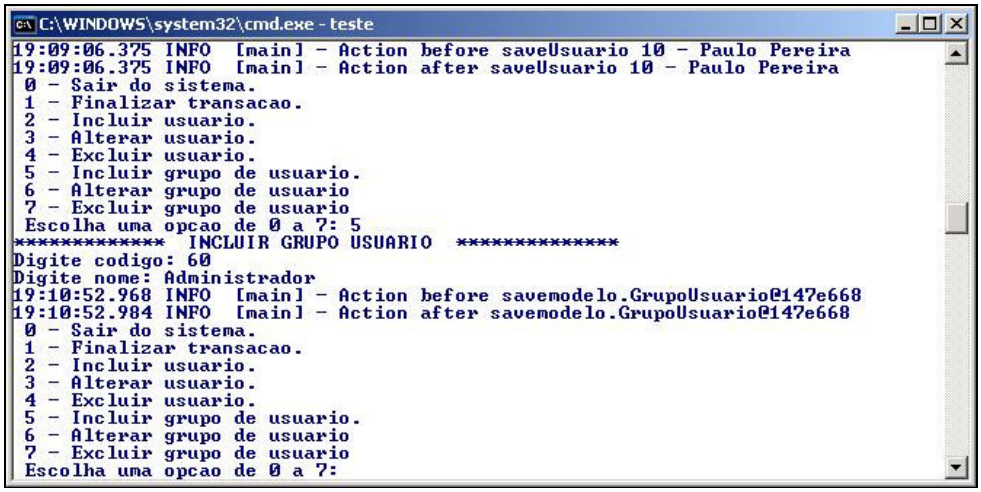


Figura 34 - – Inclusão de GRUPO_USUARIO pela classe `TesteReplica`

Contendo todos os registros atualizados na sessão, neste momento é finalizada a transação conforme apresentado na Figura 35.

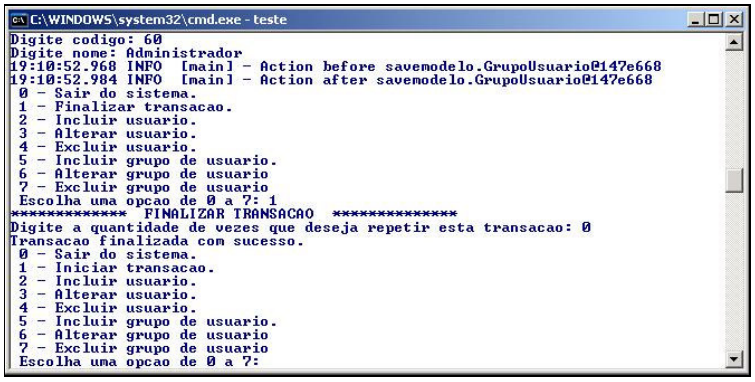


Figura 35 – Finalizar transação pela classe `TesteReplica`

Com a transação finalizada ocorreu o processo de replicação da mensagem contendo o lote de comandos e ações interceptados. A transação foi devidamente gravada na base de dados local e também replicada para a base remota. A Figura 36 apresenta o consulta realizada no MySQL para identificar os registros inclusos na base local.

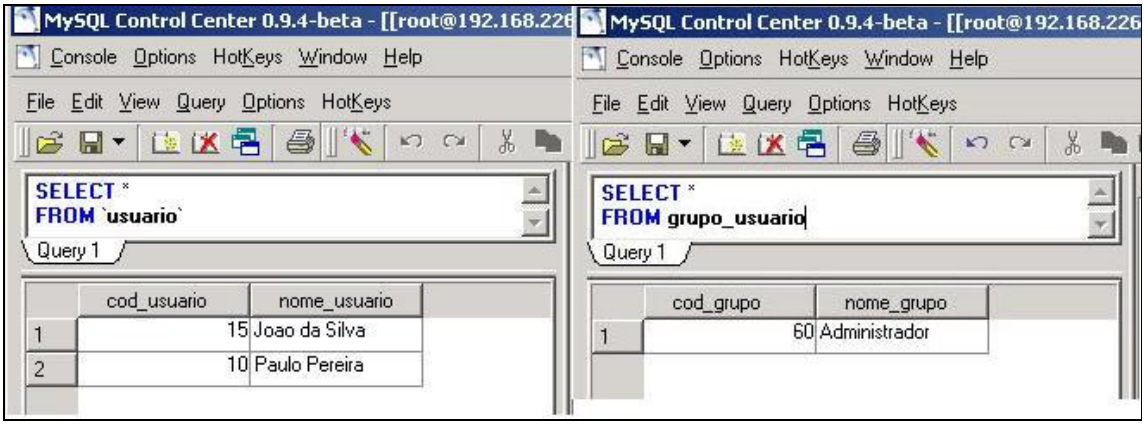


Figura 36 – Consulta MySQL, referente dados da transação “manterUsuario” inclusos.

Para garantir que os registros foram devidamente replicados para a base central, foram realizadas as mesmas consultas no MsSQL Server conforme apresentado na Figura 37.

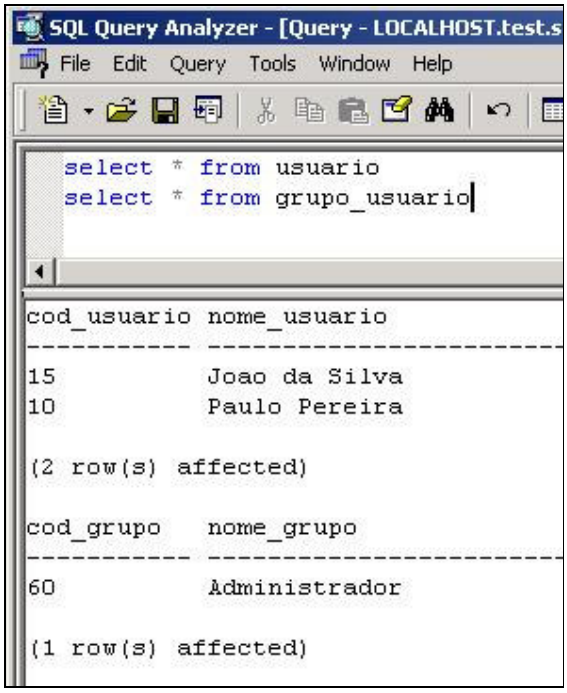


Figura 37 – Consulta MsSQL Server, referente dados da transação “manterUsuario” replicados

Neste exemplo a transação foi replicada pois existia cadastrado na base de dados o vínculo com a transação “manterUsuario”. Não existindo este vínculo cadastrado, apenas seria persistida na base local, ou seja, não teria criado o escopo de replicação.

3.5 RESULTADOS E DISCUSSÃO

A implementação do *framework* foi totalmente baseada no diagrama de classes definido previamente na fase de especificação. Isso foi um fator muito importante pois foi possível até mesmo prever problemas que poderiam ocorrer durante a fase de desenvolvimento. Ocorreram poucos casos que durante o desenvolvimento percebeu-se que seria necessário alterar a especificação, mas não representando muito esforço para realizar tais alterações.

Baseando-se nos requisitos de negócio apresentados na fundamentação teórica, é demonstrado a seguir um comparativo entre os requisitos de negócio e os resultados alcançados e não alcançados com o desenvolvimento do *framework*:

a) latência: os serviços receptores de mensagem foram desenvolvidos de forma que

pudesse ser configurado o tempo entre execução de suas iterações, tempo de re-conexão com os *sites* participante e quantidade máxima de mensagens que podem ser consumidas durante cada iteração. Estes recursos ajudam a manter a direito de processamento de cada serviços de forma igualitária, não deixando um serviço processando de forma demasiada e outro com um tempo de ociosidade muito grande. Estes recursos permitiram manter a latência configurável;

b) autonomia da réplica: A replicação é feita de forma assíncrona, não existe necessidade que os participantes da replicação estejam conectados no momento em que é finalizado uma transação no *site* de origem. Cada servidor possui um *provider* que gerencia as mensagens que estão sendo disponibilizadas localmente, como também distribuídas no servidor central. Assim sendo, os serviços apenas se comunicam entre os *providers*. Ocorrendo uma falha de conexão por alguma indisponibilidade de rede, o serviço se re-conecta de forma automática iniciando novamente o processo de replicação. As mensagens são sempre persistidas em uma outra base de dados utilizada pelo *provider*, somente é descartada no momento em que é consumida. O servidor pode ser desligado que as mensagens são persistidas em disco. Sendo novamente iniciado é carregada as mensagens em suas determinadas filas, até que outros serviços consomem as mesmas das filas. Estes recursos garantiram a autonomia da réplica;

c) disponibilidade física: Foi escolhido o tipo de arquitetura que somente envia os dados para o *site* de destino quando ocorre uma atualização no *site* de origem. É respeitada a ordem em que os comandos de atualização foram aplicados, para manter o mesmo comportamento da transação no destino. Não existindo atualização, também não ocorre replicação. Esta arquitetura tornou-se adequada referente a problemas que podem ocorrer, como sobrecarga de rede e processamento. Com este tipo de arquitetura não é necessário uma disponibilidade física muito elevada para introduzir a replicação de dados;

d) conflitos de atualização: Caso existirem dois *sites* atualizando o mesmo dado, podem ocorrer problemas pois as alterações que foram replicadas por primeiro serão perdidas. Este é um dos objetivos que não foi alcançado. Nenhum recurso foi implementado para que não ocorresse este conflito. Como o ambiente foi projetado no modelo assíncrono, tornou-se inviável trabalhar com bloqueios e outras alternativas. Pode ser evitado realizando um trabalho com a definição do modelo de dados prevendo determinados registros que possam ocorrer este conflito.

Para realizar os testes foi desenvolvido um aplicativo que se beneficiou do *framework*, para simular o processo de replicação de dados entre dois pontos de rede distintos. Estes pontos de rede estavam alocados em uma rede privada utilizando o protocolo TCP para comunicação entre os *providers*. Durante os testes foram encontrados alguns problemas como sobrecarga de processamento e memória prejudicando a performance.

Para realizar o teste de performance foi desenvolvido no aplicativo de testes um recurso para possibilitar repetir a execução de uma transação varias vezes consecutivas, gerando um número elevado de mensagens replicadas. Neste mesmo teste foram incluídas várias filas de mensagens e vinculada o nome da transação respectiva do teste com as filas cadastradas, sendo possível garantir a distribuição das mensagens para as filas interessadas na transação.

Com os cadastros realizados no gerenciador, foram realizados alguns testes de performance executando uma única transação várias vezes consecutivas utilizando outras bases de dados para persistir as mensagens JMS. Durante os testes foram constatados problemas como *deadlock,* sobrecarga de conexão e lentidão na bases MsSQL Server e MySQL. O Derby disponibilizado junto com o OpenJMS teve os melhores resultados mesmo no processamento de muitas mensagens.

No início do desenvolvimento estava sendo executado o OpenJMS e Tomcat em diferentes JVM, o que estava sobrecarregando o processamento e memória da máquina e também dificultando o processo para iniciar todos os serviços necessários para replicação. Diante deste problema, foi iniciado um estudo sobre o aplicativo OpenJMS e agregado conhecimento para iniciar o OpenJMS diretamente no momento em que é iniciado o gerenciador de replicação no próprio Tomcat. Esta modificação tornou o *framework* mais fácil de iniciar os componentes de replicação e também aumentou a performance.

Para constatar a integração com sistemas operacionais diferentes na infra-estrutura da replicação utilizou-se o Windows 98 e Windows XP. Como no servidor local, nem sempre existirá uma disponibilidade física muito grande, utilizou-se o Windows 98. O Windows XP foi utilizado no servidor central, pois necessitam uma disponibilidade física maior, referentes vários serviços sendo executados em paralelo disputando processamento entre si. Estes testes apresentaram resultados positivos, não ocorrendo problemas de incompatibilidade entre sistemas operacionais.

O *framework* desenvolvido possui características marcantes do software Postgres-R, como somente replicar atualizações e preservar o evento transacional para reduzir o tráfego de rede. O Postgres-R somente permite replicar dados entre base de dados Postgres, sendo

vantajoso utilizar o *framework* pois possui integração com o Hibernate que é compatível com diversos SGBDs. Como o Postgres-R trabalha diretamente com as bases de dados, e não como um mediador entre a aplicação e o SGBD, neste caso é possível desenvolver aplicativos em diversas linguagens. Já com o *framework*, deve somente ser desenvolvidos aplicativos na linguagem Java, caso contrário não será possível sua utilização.

Em relação ao software Heros, pode ser comparado com o recurso de heterogeneidade entre as base de dados envolvidas e também por ser multiplataforma. Os diversos protocolos de comunicação como RMI, HTTP e TCP compatíveis com o *provider* OpenJMS, também é uma das características do Heros.

É importante salientar que todos os testes se limitaram na replicação entre os SGBD`s MsSQL Server e MySQL, deixando a responsabilidade para tecnologia Hibernate, fornecer compatibilidade com os outros bancos de dados que não foram utilizados nos testes.

4 CONSIDERAÇÕES FINAIS

O *framework* desenvolvido conseguiu alcançar os objetivos desejados, demonstrando a importância e praticidade de se ter uma estrutura previamente definida capaz de replicar dados entre dois pontos distintos.

A heterogeneidade entre as bases de dados envolvidas na replicação foi alcançada aderindo a utilização da tecnologia Hibernate para persistência dos dados. O Hibernate é compatível com vários SGBD`s, permitindo que sejam facilmente configurados em sua arquitetura.

Com a utilização de padrões de projetos como DAO e *Singleton* foi possível fornecer uma estrutura que facilitou a construção de sistemas que necessitam replicar dados. Através de classes contendo métodos estáticos foram encapsuladas todas as operações necessárias para que ocorra a replicação de dados.

A camada de persistência foi completamente desenvolvida realizando a implementação de interfaces previamente definidas contendo os métodos básicos para gerenciamento de transação e persistência. A partir destas interfaces é possível utilizar outras tecnologias de persistência de dados não ficando acopladas ao *framework*. Todas as implementações destas interfaces podem ser informadas nos arquivos de configuração.

Os processos de replicação de dados se tornaram transparentes após ter desenvolvido toda configuração do *framework* em arquivos XML, sendo carregado e iniciado todos os serviços com base nestas configurações. Sendo assim não demanda muito esforço do desenvolvedor ao optar por utilizar o *framework* como *middleware* para replicação de dados.

O gerenciador de replicação também se demonstrou muito útil, pois o administrador facilmente cadastra regras de replicação diretamente no servidor central impedindo que certas transações sejam replicadas, pois em determinados casos não existe necessidade de replicar as transações que não são de importância do *site* de destino. Como estas regras são replicadas para os *sites* de destino, tornou-se prático, a configuração do ambiente de replicação.

O JMS apresentou ser uma API para gerenciamento de troca de mensagens muito robusta, oferecendo todos os requisitos necessários para cobrir por exemplo a implementação do domínio *point-to-point*.

O Hibernate apresentou características como praticidade no desenvolvimento, que enfatizou a produtividade do *framework*, economizado código e mantendo as preocupações na lógica do negócio que diz respeito a replicação de dados.

4.1 EXTENSÕES

O desenvolvimento foi praticamente voltado em criar uma estrutura na linguagem Java que fornecesse recursos necessários para uma aplicação que necessita replicar dados. Esta estrutura possui as características que encapsula todos os processos de replicação de dados.

Com a adoção do desenvolvimento desta estrutura, a aplicação obrigatoriamente deve ser desenvolvida na linguagem Java, o que impede que outros aplicativos de diferentes linguagens possam se beneficiar de seus recursos.

Como sugestão para extensão deste *framework* poderia ser transformada em um software desacoplado da aplicação. Este software representaria um *middleware* que teria a responsabilidade de ler os dados que foram atualizados na base de dados local e replicar para os *sites* de destino, podendo assim ser desenvolvida qualquer aplicação em diferentes linguagens. Para manter os eventos transacionais deveria ser pensado em regras que seriam aplicadas nos modelos de dados da aplicação para poder interagir com o *middleware*.

Como o OpenJMS apresentou problemas de persistência de dados nos SGBDs citados, tratando-se de uma aplicação *open source* poderia ser alterada a camada de persistência do *provider* para aderir a camada de persistência do próprio *framework*.

O *framework* desenvolvido não possui recurso para tratar erros que possam vir a ocorrer durante a gravação da transação no SGBD destino, como por exemplo, a replicação de registros órfãos, violação de *constraints*, alteração indevida do esquema no banco de dados e outros erros ocorridos devido as validações realizadas pelo próprio SGBD. Como extensão, poderia ser implementado um agente autônomo que monitorasse estas consistências de base, para que os erros possam ser tratados ou delegados para algum outro recurso, a fim de executar novamente as transações que foram inválidas no processo anterior.

REFERÊNCIAS BIBLIOGRÁFICAS

ALBERTO, Carlos. **Banco de dados com orientação em objetos em ASP**. Brasília, 2006. Disponível em: <http://www.imasters.com.br/artigo/4414/asp/banco_de_dados_com_orientacao_a_objetos_em_asp//imprimir/>. Acesso em: 25 jun. 2006.

BAUER, Christian; KING, Gavin. **Hibernate em ação**. Rio de Janeiro: Ciência Moderna, 2005.

CASANOVA, Marco A.; MOURA, Arnaldo V. **Princípios de sistemas de gerência de bancos de dados distribuídos**. Rio de Janeiro: Campus, 1985.

DATE, C. J. **Banco de dados:** fundamentos. Rio de Janeiro: Campus, 1985.

HIBERNATE. [S.l.], 2003. Disponível em: <http://www.hibernate.org/260.html>. Acesso em: 09 abr. 2006.

JOHANN, Eduardo; KROTH, Eduardo. Um middleware para replicação entre bancos de dados usando sistemas de comunicação de grupo. In: ESCOLA REGIONAL DE BANCO DE DADOS, 1., 2005, Porto Alegre. **Anais eletrônicos...** Porto Alegre: UNISC, 2005. Não paginado. Disponível em: <http://www.inf.ufrgs.br/~erbd2005/Artigos/7928.pdf>. Acesso em: 09 abr. 2006.

KEEME, Bettina. **Database replication.** Montreal, 2000. Disponível em: <http://www.cs.mcgill.ca/~kemme/disl/replication.html>. Acesso em: 08 abr. 2006.

LORÊDO, Heliomar Q.; FERREIRA, Lindemberg N.; ASSIS, Guilherme T. Replicação assíncrona entre bancos de dados heterogêneos. In: CONGRESSO BRASILEIRO DE COMPUTAÇÃO, 4., 2004, Itajaí. **Anais eletrônicos...** Itajaí: UNIVALI, 2004. p. 53-58. Disponível em: <http://www.niee.ufrgs.br/cbcomp/cbcomp2004/html/pdf/Banco_Dados/t170100218_3.pdf >. Acesso em: 04 abr. 2006.

MOSCARDINI, Claudete. Replicando dados com Microsoft SQL Server. **SQL Magazine**, Minas Gerais, n. 33, p. 34-47, 2006.

OPENJMS. [S.l.], 2005. Disponível em: <http://openjms.sourceforge.net/>. Acesso em: 27 out. 2006.

PAGE-JONES, Meilir. **Fundamentos do desenho orientado a objeto com UML**. São Paulo: Makron Books, 2001.

SPARX SYSTEMS. **Enterprise Architect**. [S.l.], 2005. Disponível em: <http://www.sparxsystems.com/products/ea.html>. Acesso em: 25 jun. 2006.

UCHÔA, Elvira M. A.; MELO, Rubens N. Integração de sistemas de bancos de dados heterogêneos usando frameworks. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 14., 1999, Florianópolis. **Anais eletrônicos...** Florianópolis: UFSC, 1999. p. 01-02. Disponível em: < http://www.inf.ufsc.br/sbbd99/anais/SBBD-Completo/32.pdf>. Acesso em: 09 abr. 2006.

WAENY JR., José C.; NUMAZAKI, Emílio Y. **Java message service:** teoria e prática. Florianópolis: VisualBooks, 2004.

WAYNE, José C.; LIMA, Gleydson. **Programando com JMS.** Florianópolis, 2004. Disponível em: <http://www.j2eebrasil.com.br/jsp/tutoriais/tutorial.jsp?idTutorial=012_003>. Acesso em: 05 abr. 2006.

APÊNDICE A – Implementação da classe OpenJMSServer

A seguir é apresentado o código fonte da classe `OpenJMSServer` que implementa a interface `Server`. Esta interface possui os métodos básicos para inicialização e finalização dos serviços do *Provider*, como também os componentes necessários para replicação.

```
public class OpenJmsServer implements Server{

        private Services _services;
   private Configuration _config;
   private JmsAdminServerIfc adminOpenJms;
   private Logger log =  Logger.getLogger(OpenJmsServer.class);

   public OpenJmsServer(){
   }

   /*
    * Método que inicia as configurações do OpenJMS
    */
   public void init() throws ServerJmsException{
         try{
               _services = new ServiceManager();
               ConfigurationLoader loader = new ConfigurationLoader();
               InputStream in =
OpenJmsServer.class.getResourceAsStream("/conf/openjms.xml");
               InputStreamReader fin = new InputStreamReader(in);
               _config = loader.load(Configuration.unmarshal(fin));
               fin.close();
               in.close();
      }catch(Exception e){
          throw new ServerJmsException("\\conf\\openjms.xml Erro ao carregar arquivo de
         configuração do provider OPENJMS.", e);
      }
   }

   /*
    * Método utilizado somente para testes.
    */
   public static void main(String args[]){
      try{
         Config.configure();
         OpenJmsServer server = new OpenJmsServer();
         server.version();
         server.init();
         server.startService();
         TransactionAplication.init();
         MonitorReceiver.startService();
         MonitorReceiver.registerAllReceiver();
      }
      catch(Exception exception)
```

```
        {
            exception.printStackTrace();
            System.exit(-1);
        }
    }

    /*
     * Método que escreve no log a versão do OpenJMS.
     */
    public void version()
    {
            log.info(Version.TITLE + " " + Version.VERSION);
            log.info(Version.COPYRIGHT);
            log.info(Version.VENDOR_URL);
    }

    /*
     * Método que registra os serviços do OpenJMS necessário antes de aplicar o start
     */
    protected void registerServices() throws ServiceException,Exception{

            System.setProperty("derby.system.home",Config.getInstance().getDerbyHomeOpenJ
ms());
            _services.addService(_config);
        _services.addService(org.exolab.jms.threads.ThreadPoolManager.class);
        _services.addService(org.exolab.jms.events.BasicEventManager.class);
        _services.addService(org.exolab.jms.persistence.DatabaseService.class);
        _services.addService(org.exolab.jms.scheduler.Scheduler.class);
        _services.addService(org.exolab.jms.lease.LeaseManager.class);
        _services.addService(org.exolab.jms.gc.GarbageCollectionService.class);
        _services.addService(org.exolab.jms.authentication.AuthenticationMgr.class);
        _services.addService(org.exolab.jms.authentication.UserManager.class);
        _services.addService(org.exolab.jms.messagemgr.DestinationCacheFactory.class);
        _services.addService(org.exolab.jms.messagemgr.DestinationManagerImpl.class);
        _services.addService(org.exolab.jms.messagemgr.ConsumerManagerImpl.class);
        _services.addService(org.exolab.jms.server.ServerConnectionManagerImpl.class);
        _services.addService(org.exolab.jms.messagemgr.ResourceManager.class);
        _services.addService(org.exolab.jms.server.AdminConnectionFactory.class);
        _services.addService(org.exolab.jms.server.AdminConnectionManager.class);
        _services.addService(org.exolab.jms.messagemgr.MessageMgr.class);
        _services.addService(org.exolab.jms.server.NameService.class);
        _services.addService(org.exolab.jms.messagemgr.DestinationBinder.class);
        _services.addService(org.exolab.jms.messagemgr.DestinationConfigurator.class);
        _services.addService(org.exolab.jms.server.ConnectorService.class);
    }

    public void stopService() throws ServerJmsException{
            try{
                    this._services.stop();
            }catch(Exception e){
```

```
                throw new ServerJmsException("Erro ao finalizar servicos do provider
OpenJms", e);
        }
    }

        /*
         * Método que inicia os serviços do OpenJMS
         */
    public void startService() throws ServerJmsException {
        try{
                this.version();
                log.info("Registrando serviços do OpenJms.");
                this.registerServices();
                log.info("Iniciando serviços do OpenJms.");
                this._services.start();
                log.info("Obetendo objetos administrativos do OpenJms.");
                String fileJndi = Config.getInstance().getFileContextJndi();
                String url =
        JNDIContext.getUrlConnection(Config.getInstance().getHostProviderLocal());
        adminOpenJms =
        AdminConnectionFactory.create(url,Config.getInstance().getUserOpenJms(),
        Config.getInstance().getPassOpenJms());
        log.info("Verificando se o provider está configurado como local ou master.");
        if(!Config.getInstance().isProviderMaster()){
                log.info("O framework está configurado como provider LOCAL/CLIENTE.");
                if (!this.existsQueue(Config.getInstance().getNameQueueLocal())){
                        log.info("Não existe a fila no provider, está sendo criado
                        automaticamente.");
                        this.createQueue(Config.getInstance().getNameQueueLocal());
                }
        }else{
                log.info("O framework está configurado como provider
                MASTER/CENTRAL.");
        }
      }catch(Exception e){
                throw new ServerJmsException("Erro ao iniciar servicos do provider
OpenJms", e);
      }
        }

        /*
         * Método que retorna a quantidade de mensagens pendente na fila a serem
        consumidas.
         */
        public Integer getCountMessageQueue(String nameQueue){
                try{
                        int count = adminOpenJms.getQueueMessageCount(nameQueue);
                        return new Integer(count < 0 ? 0 : count);
                }catch(JMSException e){
```

```
                        log.error("Erro ao retornar quantidade de mensagens da fila
"+nameQueue+", será retornado zero",e);
                        return new Integer(0);
                }catch(Exception e){
                        log.error("Erro ao retornar quantidade de mensagens da fila
"+nameQueue+", será retornado zero",e);
                        return new Integer(0);
                }
        }

        /*
         * Método que cria a fila passada como parâmetro no provider.
         */
        public void createQueue(String nameQueue) throws ServerJmsException {
                try{
                        log.info("Criando fila "+nameQueue+" no Servidor JMS");
                        adminOpenJms.addDestination(nameQueue,Boolean.TRUE);
                }catch(Exception e){
                        new ServerJmsException("Não foi possível criar a fila
"+nameQueue,e);
                }
        }

        /*
         * Método que remove uma determinada fila passada como parâmetro.
         */
        public void removeQueue(String nameQueue) throws ServerJmsException {
                try{
                        log.info("Removendo fila "+nameQueue+" do Servidor JMS");
                        adminOpenJms.removeDestination(nameQueue);
                }catch(Exception e){
                        new ServerJmsException("Não foi possível remover a fila
"+nameQueue,e);
                }

        }

        /*
         * Método que verifica se existe nome da fila passada como parâmetro no provider.
         */
        public boolean existsQueue(String nameQueue) throws ServerJmsException {
                try{
                        return adminOpenJms.destinationExists(nameQueue);
                }catch(Exception e){
                        new ServerJmsException("Não foi possível verificar se exsite a fila
"+nameQueue,e);
                }
                return false;
        }}
```

APÊNDICE B– Implementação da classe HibernateDao

A seguir é apresentado o código fonte da classe `HibernateDao` que implementa a interface `Dao`. Esta interface possui os métodos básicos necessários para persistência dos objetos.

```
public abstract class HibernateDao implements Dao {

	private Class classConceito = null;

	public HibernateDao(){
	}



	/*
	 * Método genérico para exclusão de registro
	 */
	public void doObjectDelete(Object conceito) throws TransactionManagerException{
		try{
			validadeTransaction();
			beforeDelete(conceito);
			((Session)TransactionAplication.getSession()).delete(conceito);
			afterDelete(conceito);
		}catch(HibernateException e){
			throw new DaoManagerDeleteException(e);
		}
	}

	/*
	 * Método genérico para inclusão de registro
	 */
	public void doObjectSave(Object conceito)throws TransactionManagerException{
		try{
			validadeTransaction();
			beforeSave(conceito);
			((Session)TransactionAplication.getSession()).save(conceito);
			afterSave(conceito);
		}catch(HibernateException e){
			throw new DaoManagerSaveException(e);
		}
	}

	/*
	 * Método genérico para atualização de registro
	 */
	public void doObjectUpdate(Object conceito)throws TransactionManagerException{
		try{
			validadeTransaction();
			beforeUpdate(conceito);
```

```
                ((Session)TransactionAplication.getSession()).update(conceito);
                afterUpdate(conceito);
            }catch(HibernateException e){
                throw new DaoManagerUpdateException(e);
            }
        }

        /*
         * Método genérico para atualização ou inclusão de registro
         */
        public void doObjectSaveOrUpdate(Object conceito) throws
TransactionManagerException {
            try{
                validadeTransaction();
                beforeSaveOrUpdate(conceito);
                ((Session)TransactionAplication.getSession()).saveOrUpdate(conceito);
                afterSaveOrUpdate(conceito);
            }catch(HibernateException e){
                throw new DaoManagerSaveException(e);
            }

        }

        /*
         * Método que retorna um objeto conforme o identificador do POJO passado como
parâmetro
         */
        public Object findByPrimaryKey(Object pkObject) throws
TransactionManagerException{
            try{
                Object obj =
(Object)((Session)TransactionAplication.getSession()).get(this.classConceito,(Serializable)pk
Object);
                return obj;
            }catch(HibernateException e){
                throw new TransactionExecuteQueryException(e);
            }finally{
                TransactionAplication.closeSession();
            }
        }

        /*
         * Método que valida se existe uma transação iniciada.
         */
        public void validadeTransaction() throws TransactionManagerException{
            if (!TransactionAplication.existsTransaction()){
                throw new TransactionNotInitialize();
            }
        }
```

```
	/*
	 * Método que configura a classe que será utilizada no Dao.
	 */
	public void setClassConceito(Class cls){
		this.classConceito = cls;
	}

	/*
	 * Pode ser implementado na sub-classe como um pós-precessamento do método
doObjectDelete
	 */
	public abstract void afterDelete(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pós-precessamento do método
doObjectSave
	 */
	public abstract void afterSave(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pós-precessamento do método
doObjectUpdate
	 */
	public abstract void afterUpdate(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pós-precessamento do método
doObjectSaveOrUpdate
	 */
	public abstract void afterSaveOrUpdate(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pré-precessamento do método
doObjectDelete
	 */
	public abstract void beforeDelete(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pré-precessamento do método
doObjectSave
	 */
	public abstract void beforeSave(Object conceito);

	/*
	 * Pode ser implementado na sub-classe como um pré-precessamento do método
doObjectUpdate
	 */
	public abstract void beforeUpdate(Object conceito);

	/*
```

```
         * Pode ser implementado na sub-classe como um pré-precessamento do método
doObjectSaveOrUpdate
         */
        public abstract void beforeSaveOrUpdate(Object conceito);
}
```

APÊNDICE C– Implementação da classe HibernateTransactionManager

A seguir é apresentado o código fonte da classe `HibernateTransactionManager` que implementa a interface `TransactionManager`. Esta interface possui as assinaturas dos métodos básicos para gerenciamento de sessão e transação da camada de persistência.

```
public class HibernateTransactionManager implements TransactionManager {

	private static ThreadLocal threadSession = new ThreadLocal();
	private static ThreadLocal threadTransaction = new ThreadLocal();
	private SessionFactory factory;
	private Logger log = Logger.getLogger(HibernateTransactionManager.class);

	/**
	 *
	 */
	public HibernateTransactionManager() {
		super();
	}


	/*
	 * Inicia uma nova transação.
	 */
	public void beginTransaction() throws TransactionManagerException {
		Transaction tx = (Transaction)threadTransaction.get();
		try{
		   if (tx == null){
			tx = ((Session)this.getSession()).beginTransaction();
			threadTransaction.set(tx);
		   }
		}catch(HibernateException ex){
			throw new TransactionManagerException(ex);
		}
	}

	/*
	 * Retorna a sessão
	 */
	public Object getSession() throws TransactionManagerException {
		Session s = (Session) threadSession.get();
		try{
			if (s == null){
				//Interceptor desenvolvido para iterceptar os métodos save,
update e delete do Hibernate.
				ReplicateInterceptor interceptor = new ReplicateInterceptor();
				s = this.factory.openSession(interceptor);
				threadSession.set(s);
			}
```

```
            }catch(HibernateException ex){
                    throw new TransactionOpenSessionException(ex);
            }
            return (Object)s;
    }

    /*
     * Finaliza a transação
     */
    public void commit() throws TransactionManagerException{
            Transaction tx = (Transaction)threadTransaction.get();
            try{
                    if (!tx.wasCommitted() && !tx.wasRolledBack())
                            tx.commit();
                    threadTransaction.set(null);
            }catch(ConstraintViolationException ex){
                    this.rollback();
                    throw new TransactionConstraintException(ex);
            }catch(HibernateException ex){
                    this.rollback();
                    throw new TransactionCommitException(ex);
            }finally{
                    this.closeSession();
            }
    }

    /*
     * Inicia a SessionFactory
     */
    public void init(HashMap mapCfg) throws TransactionManagerException {
            try{
                    String pathFile = null;
                    if (mapCfg != null)
                            pathFile = (String)mapCfg.get("fileCfg");
                    if (pathFile == null)
                            pathFile = Config.getInstance().getFileHibernateCfg();
    Document d = parseXML(pathFile);
                    factory = new Configuration().configure(d).buildSessionFactory();
            }catch(HibernateException e){
                    log.error(e.getMessage(),e);
                    throw new TransactionInitException(e);
            }catch(Exception e){
                    log.error(e.getMessage(),e);
    throw new TransactionInitException(e);
}
    }

    /*
     * Retorna o estados dos registros antes de demarcar a transação
     */
```

```
	public void rollback() throws TransactionManagerException {
		Transaction tx = (Transaction)threadTransaction.get();
		try{
			threadTransaction.set(null);
			if (tx != null && !tx.wasCommitted() && !tx.wasRolledBack()){
				tx.rollback();
			}
		}catch(HibernateException e){
			throw new TransactionRollbackException(e);
		}finally {
			this.closeSession();
		}
	}

	/*
	 * Fecha a sessão
	 */
	public void closeSession() throws TransactionManagerException {
		Session s = (Session) threadSession.get();
		try{
			threadSession.set(null);
			if (s == null && s.isOpen()){
				s.close();
			}
		}catch(HibernateException ex){
			throw new TransactionManagerException(ex);
		}
	}

	/*
	 * Completa uma intrução HQL com base nos parâmetros passados, executa e retorna
uma
	 * lista do resultado.
	 */
	public List executeQuery(String query, HashMap parameters) throws
TransactionManagerException {
		try{
			Query q = ((Session)this.getSession()).createQuery(query);
			if (parameters != null && !parameters.isEmpty()){
				Iterator it = parameters.keySet().iterator();
				while (it.hasNext()){
					String key = (String)it.next();
					Object obj = parameters.get(key);
					if (obj instanceof String){
						q.setString(key,(String)obj);
					}else if (obj instanceof Integer){
						q.setInteger(key,((Integer)obj).intValue());
					}else if (obj instanceof Double){
						q.setDouble(key,((Double)obj).doubleValue());
```

```
                                        }else if (obj instanceof Long){
                                                q.setLong(key,((Long)obj).longValue());
                                        }else if (obj instanceof Date){
                                                q.setDate(key,(Date)obj);
                                        }else if (obj instanceof Conceito){
                                                q.setSerializable(key,(Conceito)obj);
                                        }
                                }
                        }
                        return q.list();
                }catch(HibernateException e){
                        throw new TransactionExecuteQueryException(e);
                }catch(Exception e){
                        throw new TransactionExecuteQueryException(e);
                }finally{
                        this.closeSession();
                }

        }

        /*
         * Método utilizado para desativar algumas validações de TAG do arquivo de
configuração
         * do Hibernate e retornar um Document para ser carregada a SessionFactory
         */
        private static Document parseXML (String filename) throws Exception {
      InputStream is =
HibernateTransactionManager.class.getResourceAsStream("/conf/"+filename);
      InputSource source = new InputSource(is);
      DOMParser parser = new DOMParser();
      parser.setFeature("http://xml.org/sax/features/validation", false);
      parser.setFeature("http://apache.org/xml/features/nonvalidating/load-dtd-grammar",
false);
      parser.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",
false);
      parser.parse(source);
      return parser.getDocument();
   }

        /*
         * Verifica se existe uma transação iniciada
         */
        public boolean existsTransaction() throws TransactionManagerException {
                Transaction tx = (Transaction)threadTransaction.get();
                if (tx == null)
                        return false;
                return true;
        }

        /*
```

```
         * Os métodos abaixo não foram implementados pois a tecnologia Hibernate possui os
atributos
         * de conexão encapsulados no arquivo de configuração
         */
        public void readFileCfg(File file) throws TransactionManagerException {

        }
        public void setDriverJdbc(String nome) throws TransactionManagerException {
        }
        public void setPassword(String paswd) throws TransactionManagerException {
        }
        public void setUrl(String nomeUrl) throws TransactionManagerException {
        }
        public void setUsuario(String nome) throws TransactionManagerException {
        }
}
```

APÊNDICE D– Laço de execução do método run() da classe JmsMsgReceiver

A seguir é apresentado parte do código fonte da classe `JmsMsgReceiver`, representando o laço de execução do método `Run()`, referente implementação da interface `Runnable`.

```
public void run() {
	boolean logSleep = true;
	while (!isStopReceiver()){
	 try{
		this.connect();
		Message msg = null;
		int qtdMsgsFromIteration =
Config.getInstance().getQtdMsgsFromIteration().intValue();
		int contMsgsFromIteration = 0;
while(!isStopReceiver() && this.receiver != null && (msg =
this.receiver.receive(Config.getInstance().getToleranteTimeOutServiceReceiver().longValue(
))) != null){
		ObjectMessage objMsg = (ObjectMessage)msg;
		contMsgsFromIteration++;
		if (objMsg.getStringProperty("Command") != null){
			boolean forward = objMsg.getBooleanProperty("Forward");
			HashMap map = (HashMap)objMsg.getObject();
		BrokerCommand.execute(map,this.host,this.nameQueue,forward);
		logSleep = true;
}
if(contMsgsFromIteration == qtdMsgsFromIteration)
	break;
}
if (!this.isConected())
	this.reconnect();

if (logSleep){
	log.info(this+" sleep!");
	logSleep = false;
}

this.sleep(Config.getInstance().getTimeSleepServiceReceiver().longValue());
}catch(JMSException ex){
	this.reconnect();
}catch(CommunicationException ex){
	this.reconnect();
}catch(NameNotFoundException e){
	this.reconnect();
}catch(Exception ex){
	ex.printStackTrace();
	this.setStopReceiver();
}
}
log.info("Serviço "+this+" entrou no estado inativo");
}
```