CESAR FERNANDO MORO

SUPORTE À PERSISTÊNCIA DE ARTEFATOS PARA O AMBIENTE DISTRIBUÍDO DE DESENVOLVIMENTO DE SOFTWARE *DiSEN*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, Curso de Pós-Graduação em Informática, do Setor de Ciências Exatas, Universidade Federal do Paraná, em convênio com o Departamento de Informática da Universidade Estadual de Maringá.

Orientadora: Elisa Hatsue Moriya Huzita

CURITIBA
2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Cesar Fernando Moro , avaliamos o trabalho intitulado, "Suporte à Persistência de Artefatos para o Ambiente Distribuído de Desenvolvimento de Software DiSEN", cuja defesa foi realizada no dia 22 de agosto de 2003, às quatorze horas, no Anfiteatro B do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato. (Convênio número 279-00/UFPR de Pós-Graduação entre a UFPR e a UEM - ref. UEM número 1331/2000-UEM).

Curitiba, 22 de agosto de 2003.

Profª. Dra. Elisa Hatsue Moriya Huzita
DIN/UEM – Orientadora

Profª. Dra. Líria Matsumoto Sato
POLI/USP – membro externo

Prof. Dr. Marcos Sfair Sunyé
DINF/UFPR

AGRADECIMENTOS

Agradeço ao Pai.

Agradeço à toda minha família pelo carinho, compreensão e respeito com que abrilhantam a minha vida.

Em especial agradeço ao meu pai pelo exemplo, à minha mãe pelo aconchego e à minha esposa pela paciência de quem aceitou ser companheira de caminhada.

Agradeço, também, a todos que colaboraram para realização desta pesquisa, em especial pelo apoio da minha orientadora Elisa e dos meus colegas e companheiros de trabalho.

Dedico este trabalho às duas provas vivas de que a Vida não acaba, mas continua e se renova a cada dia, aos meus amados filhos Pedro e Clara

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E LISTA DE SIGLAS

DiSEN	- Distributed Software Engeneering Environment
DTD	- Document Type Definition
JCP	- Java Community Process
Jini	- Jini Network Technology
JMI	- Java Metadata Interface
MDA	- Model Driven Architecture
MDR	- MetaData Repository
MDSODI	- Metodologia para Desenvolvimento de software Distribuído
MOF	- Meta Object Facility
OMG	- Object Management Group
UML	- Unified Modeling Language
W3C	- World Wide Web Consortium
XMI	- XML Metadata Interchange
URI	- Universal resource Identifier
IDE	- Integrated Development Environment
XSL	- eXtensible Stylesheet Language
XSLT	- eXtensible Stylesheet Language for Transformations

RESUMO

O advento e o crescimento da utilização das redes de computadores, tem levado as empresas a adotar, cada vez mais, tecnologias de sistemas distribuídos para implementação de seus sistemas. *DiSEN* é um ambiente distribuído de desenvolvimento de software, no qual a *MDSODI* está inserida, que tem como um de seus objetivos, permitir que vários desenvolvedores, atuando em locais distintos, possam trabalhar de forma cooperativa no desenvolvimento de software. No processo de desenvolvimento de um software, os desenvolvedores, geralmente, utilizam diferentes ferramentas de apoio, cada qual aplicada a uma ou mais fases do processo cujos artefatos precisam ser integrados. *Meta Object Facility - MOF* e, *XML Metadata Interchange - XMI* são especificações de padrões produzidas pela *OMG* que têm como um de seus objetivos facilitar a integração de diferentes ferramentas de desenvolvimento de software. Estas especificações podem ser utilizadas para proporcionar o intercâmbio de metadados entre diferentes aplicações e fazem parte do núcleo da *Model Driven Architecture - MDA* proposta pela *OMG*. No ambiente corporativo é possível verificar a crescente utilização de repositórios de metadados no armazenamento de artefatos no suporte ao trabalho cooperativo de desenvolvedores de software. O presente trabalho aborda o desenvolvimento de um repositório distribuído de artefatos para o ambiente *DiSEN*, baseado em um repositório de metadados com suporte aos padrões *MOF* e, *XMI*.

ABSTRACT

The growing of the use of computer networks, have taken the companies to adopt, more and more, technologies of distributed systems in the implementation of their systems. *DiSEN* is a software development distributed environment, in which the *MDSODI* is inserted, that has as one of its goals, to allow developers, acting in distinct places, to be able to work in a cooperatively form in the software development. Generally, in the software development process, developers use different tools, each one applied to one or more of the process phases, whose artefacts need to be integrated. Meta Object Facility - MOF and, XML Metadata Interchange - XMI are standards specifications produced by *OMG* that have as one of their goals to facilitate the integration of different software development tools. These specifications can be used to provide the interchange of metadata between different applications and are part of the Model Driven Architecture - MDA kernel, proposed by OMG. In the enterprise environment it is possible to verify the increasing use of metadata repositories to storage artefacts in the support of the cooperative work of software developers. The study, here presented, approaches the development of a distributed artefacts repository, to be used in the *DiSEN* environment, based on a metadata repository with support to *MOF* and, *XMI* standards.

1 INTRODUÇÃO

Nos tempos atuais, com o advento e o crescimento da utilização das tecnologias de rede, as empresas têm cada vez mais adotado tecnologias de sistemas distribuídos para implementação de seus sistemas.

A Metodologia de Desenvolvimento de Software Distribuído - *MDSODI* (GRAVENA, 2000) (HUZITA, 1999) é uma metodologia de desenvolvimento de software que oferece suporte à especificação de alguns aspectos relacionados a sistemas distribuídos desde as fases iniciais de projeto. A notação desta metodologia baseia-se na *Unified Modeling Language – UML* (OMG, 2003a), (OMG, 2003b), mas adota também algumas extensões para a representação de sistemas distribuídos. Dessa forma é possível abordar de forma clara os aspectos relacionados à distribuição do sistema desde, as fases iniciais do processo de desenvolvimento de software.

Como a *MDSODI* é uma metodologia que cobre todo o processo de desenvolvimento de software, é natural imaginar-se um cenário onde serão utilizadas várias ferramentas de apoio ao desenvolvimento de software cada qual criando e, possibilitando a manutenção necessária dos vários artefatos relativos às diversas fases do processo. Estas ferramentas poderão ter sido criadas especialmente para esta metodologia, ou então serem ferramentas já existentes no mercado, produzidas por terceiros, cada uma com sua especialidade e potencialidade.

Na Universidade Estadual de Maringá, membros do Grupo de Estudos e Pesquisa de Sistemas de *Software* Distribuídos - *GESSD* desenvolvem vários trabalhos relacionados à metodologia *MDSODI* e, o ambiente distribuído de desenvolvimento de software, denominado *Distributed Software Engineering Environment - DiSEN* (PASCUTTI, 2002).

DiSEN foi concebido com vistas à utilização da metodologia *MDSODI*, enquanto que na sua arquitetura, pode-se observar que o mesmo oferece suporte a agentes. Entende-se, que neste ambiente, estarão sendo utilizadas várias ferramentas de desenvolvimento de software, onde cada uma delas estará produzindo artefatos como resultado do trabalho do desenvolvedor de software.

Segundo (OMG, 2003c), na realidade, não existe uma única ferramenta que seja capaz de abranger todas as etapas do desenvolvimento do software, desde a sua modelagem até a sua documentação. Uma combinação de ferramentas de diferentes fabricantes é necessária, porém difícil de se alcançar, uma vez que as ferramentas geralmente não conseguem fazer de forma facilitada o intercâmbio das informações que utilizam (figura 1).

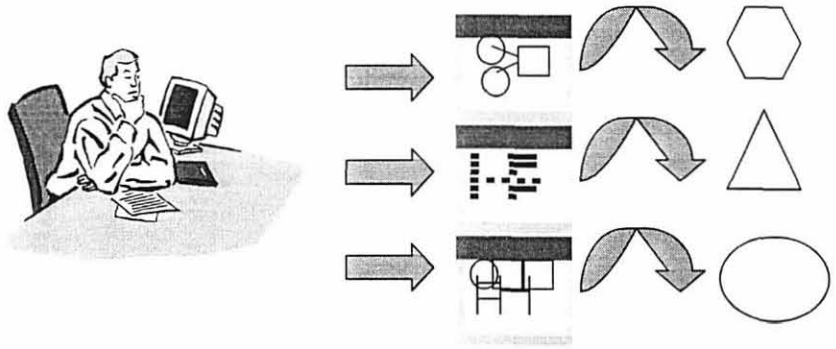

Figura 1- Desenvolvedor de software gerando artefatos diferentes

Ao trazer este exemplo para o contexto de sistemas distribuídos e de processos de desenvolvimento de sistemas distribuídos, é possível expandir o cenário de desenvolvimento de software descrito no parágrafo anterior, imaginando que cada ferramenta poderia estar sendo executada em um nó distinto da rede e que estas ferramentas deveriam poder trabalhar de forma integrada, possibilitando o intercâmbio de informações entre elas. Cada ferramenta, então, deveria ser capaz de acessar os artefatos, produzidos pelas outras ferramentas ao mesmo tempo em que disponibilizaria o acesso aos artefatos que ela mesma produz.

Em um ambiente, com as características descritas acima, o intercâmbio de informações relativas aos artefatos, produzidos durante o processo de desenvolvimento de software, é muitas vezes prejudicado pela adoção de formatos proprietários por parte de cada fabricante de ferramentas. Cada um destes fabricantes, na grande maioria das vezes, opta por representar suas informações de uma maneira própria, gerando artefatos que são armazenados em formatos diferentes, dificultando, conseqüentemente, que uma ferramenta consiga ler os artefatos produzidos por outra ferramenta.

Além disso, particularmente em um ambiente distribuído de desenvolvimento de software, para que possa haver o intercâmbio de informações entre diferentes ferramentas localizadas, em diferentes nós da rede, seria necessário que elas pudessem obter acesso aos artefatos produzidos por outras ferramentas através da própria rede.

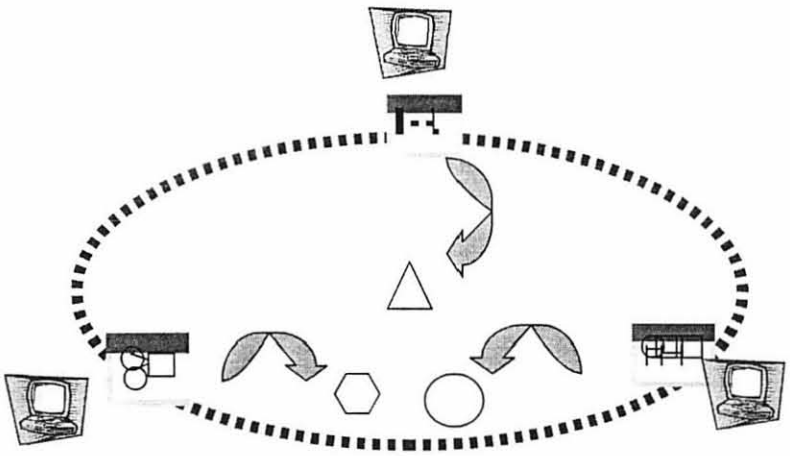

Figura 2 - Ferramentas distribuídas gerando artefatos diferentes

O formato das informações, manipuladas por uma ferramenta de desenvolvimento de software, pode ser descrita em um modelo de metadados. Assim, no cenário apresentado na figura 2, a dificuldade de integração e interoperabilidade entre as ferramentas de diferentes fabricantes pode ser vista como conseqüência da falta da utilização de um modelo comum de metadados. Sem a utilização de um modelo comum de metadados, cada fabricante adota um modelo particular, com estrutura, semântica e sintaxe próprias.

No presente trabalho, é adotada a definição de metadados de (OMG, 2002b), onde metadado é apresentado como qualquer dado que descreve outros dados. Na realidade, metadados descrevem tanto a estrutura de uma informação (sua sintaxe), como o que ela significa (sua semântica).

1.1 CONTRIBUIÇÃO DO TRABALHO, MOTIVAÇÃO E OBJETIVOS

Em (PASCUTTI, 2002), é apresentada uma proposta de uma arquitetura para o ambiente *DiSEN.* Um dos elementos desta arquitetura é o suporte fornecido pelo próprio ambiente *DiSEN,* para a persistência dos artefatos produzidos pelas diferentes ferramentas de desenvolvimento de software, conectadas ao ambiente. O ambiente *DiSEN,* bem como sua arquitetura, é apresentado em mais detalhes na seção 4.7.

Ainda no contexto de desenvolvimento distribuído de software, *Meta Object Facility - MOF* (OMG, 2002b) e *XML Metadata Interchange - XMI* (OMG, 2003c) são especificações produzidas pela *OMG* que têm como um de seus objetivos facilitar a integração de diferentes ferramentas de desenvolvimento de software, independente de seu fabricante. Estas especificações procuram estabelecer padrões que podem ser utilizados para proporcionar o intercâmbio de metadados entre diferentes aplicações. Estas especificações fazem parte, ainda, do núcleo da *Model Driven Architecture - MDA* (OMG, 2001c) proposta pela *OMG*.

Uma das aplicações do padrão *MOF* é, justamente, a construção de um repositório de metadados que seja capaz de armazenar modelos que serão acessados por diversas ferramentas, dando suporte ao desenvolvimento distribuído de sistemas de software.

> "Initially, one of the most likely applications of the *MOF* will be to support the development of distributed object-oriented software from high-level models. Such a software development system would typically consist of a repository service for storing the computer representations of models and a collection of associated tools. The latter would allow the programmers and designers to input the models, and would assist in the process of translating these models into software implementations." (OMG, 2001a) p. 26.

Além disso, para o suporte ao desenvolvimento cooperativo de software, vários fabricantes de ferramentas de modelagem têm adotado a utilização de repositórios de metadados no armazenamento dos artefatos produzidos pelas ferramentas de modelagem (ver capítulo 2).

Dentro deste contexto, o presente trabalho aborda justamente o desenvolvimento de um repositório de artefatos, para o ambiente *DiSEN,* baseado

em um repositório de metadados em conformidade com as especificações *MOF* e *XMI* da *OMG*. Este repositório de artefatos poderá ser usado tanto no armazenamento dos artefatos produzidos pelas diversas ferramentas de desenvolvimento de software, utilizadas no ambiente, bem como, no armazenamento dos metadados que descrevem estes artefatos e será denominado *Distributed Artefact ReposiTory – DART.*

Enquanto que, um repositório de artefatos para o ambiente distribuído possibilitará aos desenvolvedores utilizarem o *DiSEN* para persistir seus artefatos, o acesso aos metadados que descrevem os artefatos possibilitará, também, que :

- ferramentas possam manipular estes artefatos, mesmo sem conhecimento prévio de seu formato, tomando como base a descrição presente nos seus metadados;
- seja facilitada a construção de ferramentas de conversão entre um formato, ou tipo de artefato, para outro;
- os artefatos tenham o formato de suas informações verificados de acordo com a descrição dos seus metadados.

Além disso a adoção de padrões abertos como *MOF* e *XMI*, possibilitará:

- a utilização do repositório em conjunto com ferramentas de desenvolvimento de software já existentes, uma vez que várias delas já oferecem suporte à importação e exportação de artefatos no formato *XMI* (ver seção 4.3.1);
- flexibilidade nos formatos de artefatos que poderão ser armazenados, uma vez que será possível armazenar qualquer artefato cujo modelo possa ser descrito em *MOF.*

Não obstante, o armazenamento conjunto dos dados e metadados dos artefatos em um único mecanismo de persistência, impedirá a possível ocorrência de inconsistências entre estes dados e seus metadados. Falha esta encontrada em outros repositórios de artefatos existentes para ambientes de desenvolvimento de software, como por exemplo, o repositório de artefatos *OSCAR* apresentado no capítulo 2.

O presente trabalho, então, dá continuidade aos trabalhos apresentados em (PASCUTTI, 2002) e (HUZITA, 1999). Ele, também, vai ao encontro das necessidades dos trabalhos em andamento relacionados à metodologia *MDSODI,* que estão sendo desenvolvidos pelos membros do *GESSD,* no sentido que oferece ao ambiente *DiSEN* um repositório de artefatos.

Ao mesmo tempo, estando em conformidade com os padrões *MOF* e *XMI,* coloca estes trabalhos no contexto da *MDA* e proporciona uma base para realização de trabalhos futuros com relação à possível integração de diferentes ferramentas de desenvolvimento de software no ambiente *DiSEN.*

Outra característica do presente trabalho é a preocupação com aspectos relacionados à distribuição do repositório de artefatos, desde as fases iniciais de seu desenvolvimento. Na seção 4.8, são apresentadas várias características, desejáveis em um repositório, que foram consideradas no seu desenvolvimento.

Apesar de existirem outros trabalhos sobre a utilização de repositórios de artefatos no armazenamento dos dados e metadados dos artefatos em um ambiente distribuído (ver capítulo 2), o presente trabalho contribui na medida em que adota como requisito para o desenvolvimento do repositório de artefatos pelo menos as seguintes características:

- conformidade com padrões abertos propostos pela *OMG* (*MOF, XMI*);
- preocupação com aspectos relacionados à distribuição do repositório como escalabilidade, disponibilidade, concorrência, balanceamento de carga, recuperação de falhas, etc, desde as fases iniciais de desenvolvimento do repositório;
- adoção de um mecanismo único de persistência tanto para os dados dos artefatos, como para seus metadados, evitando inconsistências entre eles.

Uma lista completa das características consideradas para o repositório de artefatos é apresentada na seção 4.8.

O presente trabalho, portanto, contribui para as pesquisas na área de engenharia de software, propondo uma abordagem para o desenvolvimento de um repositório de artefatos, baseado em um repositório de metadados, com as

características relacionadas na sessão 4.8, para ser integrado ao ambiente distribuído de desenvolvimento de software *DiSEN*.

Esta dissertação esta organizada em 7 capítulos. O capítulo 2 apresenta alguns trabalhos relacionados. Os capítulo 3 e 4 apresentam o embasamento necessário para o entendimento das tecnologias relacionadas e utilizadas no presente trabalho. Ainda nas seções 4.7 e 4.8 é descrito o suporte à persistência de artefatos necessária ao *DiSEN* e são apresentados as características desejáveis ao repositório de artefatos. No capítulo 5 é apresentada a abordagem utilizada no desenvolvimento do repositório de artefatos bem como a descrição de sua execução. No capítulo 6, são apresentados os resultados obtidos e os testes realizados com o repositório de artefatos para o *DiSEN*. Finalmente, o capítulo 7 apresenta as conclusões e sugestões de trabalhos futuros.

2 ESTADO DA ARTE

No ambiente corporativo é possível verificar que, vários fabricantes de ferramentas de modelagem e de consultoria na área de desenvolvimento de software têm adotado a utilização de repositórios de metadados, os quais têm sido incorporados às ferramentas de desenvolvimento de software.

> Repositories are being packaged with development and data modeling tools and even being managed by development tools. (GATES, 2001)

Exemplos destes repositórios são: *Adaptive Repository* (ADAPTIVE, 2003), *Meta Integration Repository Server* (META, 2003), *MetaBase Repository* (METAMATRIX, 2003) e *dMOF (DISTRIBUTED SYSTEMS TECHNOLOGY CENTRE, 2002)*.

Todos estes repositórios têm em comum o fato de serem produtos comerciais, que são oferecidos pelos respectivos fabricantes, juntamente com outras ferramentas voltadas para o gerenciamento dos dados e metadados de uma empresa.

Entre as características destes produtos está o suporte às especificações dos padrões *MOF* e *XMI da OMG*.

O *MetaBase Repository* e as ferramentas, que o acompanham, possibilitam que sejam criados modelos que representem as diferentes fontes de dados de uma empresa e que estes modelos sejam usados no acesso unificado a estes dados.

Já o *Adaptive Repository* e o *Meta Integration Repository Server* permitem o armazenamento de artefatos provenientes de diversos projetos e ferramentas. Ambos os repositórios utilizam bancos de dados como *backend* e disponibilizam o acesso aos repositórios através de páginas WEB.

Quanto ao *dMOF,* apesar de ser implementado de acordo com a versão 1.3 da especificação do *MOF* e a versão 1.1 da especificação *XMI*, ele não tem sofrido atualizações desde a sua versão 1.1. Assim, implementações do suporte às novas versões das especificações de *XMI*, *MOF* e mesmo *JMI* (DIRCKZE, 2002), especificada pelo *Java Community Process - JCP*, não foram implementadas no

produto. Assim, *dMOF* não apresenta algumas funcionalidades, conforme documentado em (DISTRIBUTED SYSTEMS TECHNOLOGY CENTRE, 2002).

Com exceção ao *dMOF,* os demais produtos citados incluem, também, as seguintes funcionalidades:

- controle de acesso;
- controle de versão;
- suporte à utilização do repositório através da rede.

Apesar das funcionalidades destes produtos os tornarem atrativos para serem utilizados em um ambiente, distribuído de desenvolvimento de software, todos eles têm um custo de utilização, muitas vezes elevado. É necessário a aquisição das licenças para o repositório e, também, de licenças para o banco de dados, utilizado como *backend*.

Outro fator limitante para a utilização destes repositórios é o fato de que muitos deles não são oferecidos isoladamente, mas sim como parte integrante de um conjunto maior de ferramentas, fornecidas pelo fabricante, como uma solução completa de desenvolvimento de software.

Estas limitações, encontradas nos repositórios de metadados disponíveis no mercado, acabam por dificultar a sua utilização na construção de um ambiente distribuído de desenvolvimento de software como parte integrante de sua arquitetura.

Já com relação a repositórios de artefatos de código aberto para ambientes distribuídos de desenvolvimento de software, é possível citar o *OpenSource Component Artefact Repository - OSCAR* que faz parte da plataforma *Generalized EnvironmEnt for Process Management In Cooperative* software *Engineering – GENESIS* (BOLDYREFF; SMITH; WEISS, 2003).

GENESIS é um projeto de uma plataforma de código aberto que oferece suporte à cooperação e à comunicação entre desenvolvedores de software distribuídos.

GENESIS foi projetado para inter-operar de forma não invasiva com sistemas de gerenciamento de *workflow*, ferramentas de desenvolvimento de *software* e sistemas de repositórios já existentes, oferecendo suporte ao trabalho

cooperativo de equipes distribuídas de engenharia de software (NUTTER; BOLDYREFF; RANK, 2002a).

OSCAR, por sua vez, tem como objetivo oferecer acesso distribuído a uma coleção de artefatos para que possam ser utilizados por diferentes equipes de desenvolvimento.

OSCAR representa toda informação sob seu controle como um artefato:

> Firstly, OSCAR represents all data under its control as an artefact, including users of the system, the process models and instances they are using, tools and services that the system employs directly, output from metrics and other monitoring procedures that act on the repository as well as data created by tools not directly integrated with OSCAR. (NUTTER; BOLDYREFF;RANK, 2002a)

Um artefato é representado como um documento *XML.* Existe uma *DTD* que descreve as caracteristivas mínimas de um artefato, ou seja, de um artefato do tipo *Default.* Novos tipos de artefatos são criados através da definição de novas *DTDs,* que estendem essa *DTD Default.* Um artefato é identificado como sendo de determinado tipo, de acordo com a sua conformidade com determinada *DTD.*

Um artefato, por exemplo, que contenha um modelo *UML*, pode ser composto por uma descrição do artefato em *XML* (são os metadados) e uma descrição do modelo *UML* em *XMI* (são os dados do artefato). Apesar de *OSCAR* não entender as informações descritas em *XMI,* ele é capaz de trabalhar com o artefato com base nos metadados, em *XML,* que o descrevem.

Além da *DTD* que descreve artefatos do tipo *Default*, *OSCAR* ainda oferece outras *DTDs* que descrevem outros tipos de artefatos. Estes tipos são: software, *Annotation, HumaResource* e *Project.*(NUTTER; BOLDYREFF;RANK, 2002a).

Apesar de *OSCAR* apresentar seus artefatos como documentos *XML* que contém tanto os dados do artefato como os metadados que descrevem estes dados, na realidade, estes são armazenados utilizando mecanismos de persistência distintos. Enquanto que os metadados de um artefato são armazenados em um banco de dados *PostgreSQL* (POSTGRESQL, 2003), os dados do artefato são armazenados em um sistema de controle de versões *CVS* (CONCURRENT, 2003).

Uma vez que um dos objetivos de projeto da plataforma *GENESIS* é

minimizar interferência, muitas das funções do *OSCAR* são fornecidas através da incorporação de outras aplicações de código aberto. Exemplos são, justamente, a utilização de *plugins* para a incorporação do *CVS* e do banco de dados *PostgreSQL* na persistência dos artefatos.

Para acessar um determinado artefato armazenado pelo *OSCAR*, é utilizada uma *URI* que pode conter ou não informações sobre a versão do artefato.

OSCAR permite que operações em artefatos gerem eventos. Estes eventos podem ser propagados para outros artefatos, ou mesmo, para membros de uma equipe de desenvolvimento.

OSCAR tem, ainda, uma preocupação com o suporte à pesquisa dos artefatos armazenados. Atualmente, porém, este suporte está limitado à pesquisa por palavras chave, utilizando *SQL* nos metadados armazenados no banco de dados *PostgreSQL*.

Apesar de poder ser utilizado gratuítamente e de código aberto, o *OSCAR* possui algumas limitações:

- diferentemente dos repositórios comerciais citados, *OSCAR* não apresenta suporte às especificações dos padrões *MOF* e *XMI* da *OMG;*
- apesar da praticidade em se usar uma URI para acessar diretamente uma cópia do artefato, essa abordagem pode nem sempre funcionar, uma vez que o *OSCAR* pode não estar no ar em um dado momento. Atualmente *OSCAR* ainda não aborda problemas de disponibilidade;
- uma das limitação do *OSCAR,* decorrente da utilização e da forma de incorporação do *CVS*, é que os elementos dos artefatos armazenados no *CVS* podem ser acessados e alterados externamente por cliente do *CVS;* assim, um cliente do *CVS* poderia alterar parte de um artefato, diretamente, sem passar pelo controle do *OSCAR,* causando, possívelmente, inconsistência entre as informações armazenados no *PostgreSQL* e no *CVS*.

Por outro lado, *Metadata Repository - MDR (SUN, 2002b),* é um outro repositório de metadados de código aberto, também, em conformidade com as especificações *MOF* e *XMI* da *OMG*.

MDR é disponibilizado como parte do projeto *NetBeans* (SUN, 2002a), e trabalha de forma integrada com o *IDE* de desenvolvimento *NetBeans*. Apesar disso ele pode também trabalhar isoladamente ou ser adaptado para trabalhar com outras ferramentas, uma vez que assim como o *IDE Netbeans*, o *MDR* é de código aberto.

A interface do repositório é baseada na especificação *Java Metadata Interface - JMI* (DIRCKZE, 2002), que define um mapeamento *Java* para o padrão *MOF*.

Apesar de poder trabalhar isoladamente do *IDE Netbeans*, *MDR* não tem nenhum suporte para comunicação com clientes através de uma rede de computadores. Outra característica deste repositório é ele não utilizar um sistema gerenciador de banco de dados como *backend*. Na realidade, o repositório utiliza o próprio sistema de arquivos e uma implementação de árvore-B na armazenagem e acesso dos elementos do repositório.

Na seção 4.5, são apresentados mais detalhes sobre o repositório de metadados *MDR* e, inclusive, alguns projetos e ferramentas que fazem uso de funcionalidades.

3 SISTEMAS DISTRIBUÍDOS E SISTEMAS DINÂMICOS

3.1 INTRODUÇÃO

DiSEN é um ambiente distribuído para desenvolvimento de software, proposto por (PASCUTTI, 2002). Atualmente, os pesquisadores do *GESSD* realizam pesquisa que envolvem a continuação da proposta apresentada em (PASCUTTI, 2002), inclusive com trabalhos de implementação do ambiente *DiSEN* e de ferramentas para serem utilizadas com ele (BATISTA, 2002), (PEDRAS, 2002).

O presente trabalho, por exemplo, apresenta a criação do suporte à persistência de artefatos para o *DiSEN* e, assim, como os demais trabalhos relacionados a este ambiente, precisa considerar os conceitos, as características e as tecnologias que envolvem sistemas distribuídos.

Neste capítulo são apresentados os conceitos e tecnologias ligados a sistemas distribuídos que foram adotados para a elaboração do presente trabalho.

3.2 SISTEMAS DISTRIBUÍDOS

Segundo (COULOURIS; DOLLIMORE; KINDBERG, 2001), um sistema *distribuído é aquele no qual os componentes estão localizados em computadores*, ligados a uma rede e que se comunicam e coordenam suas ações apenas através da passagem de mensagens. Com essa definição são identificadas as seguintes características de sistemas distribuídos:

- **concorrência:** em uma rede de computadores programas são executados de forma concorente. A coordenação destes programas concorrentes que compartilham recursos é um dos aspectos importantes em sistemas distribuídos;
- **a falta de um relógio global:** em uma rede de computadores existem limites para a precisão com que estes podem sincronizar seus relógios, e a coordenação das ações dos componentes depende, muitas vezes, de

uma noção comum de tempo;

- **independência à falhas:** todo sistema de computador pode falhar, portanto, é de responsabilidade do projetista do sistema planejar as conseqüências das possíveis falhas. Um sistema distribuído pode falhar de diversas maneiras mas a capacidade do sistema e de seus componentes continuarem operando, mesmo que a rede ou alguns componentes isolados apresentem falhas, é uma característica importante a ser considerada.

A principal motivação de um sistema distribuído é justamente o compartilhamento de recursos e a construção de tal sistema implica em uma série de desafios relacionados à heterogeneidade, abertura, segurança, escalabilidade, tratamento de falhas, concorrência e transparência (COULOURIS; DOLLIMORE; KINDBERG, 2001).

Ainda segundo (COULOURIS; DOLLIMORE; KINDBERG, 2001), com relação às arquiteturas dos sistemas distribuídos, pode-se identificar as seguintes camadas:

- camada de aplicação de serviços;
- *middleware*;
- sistema operacional;
- computadores e equipamentos de rede.

O sistema operacional juntamente com os computadores e os equipamentos de rede são geralmente referenciados como a plataforma do sistema.

Já o *middleware* consiste na camada de software, cujo propósito é mascarar a heterogeneidade, e prover um modelo de programação conveniente aos desenvolvedores. O objetivo do *middleware* é prover infra-estrutura para a construção de elementos de software que possam trabalhar uns com os outros em um sistema distribuído. Para tanto, o *middleware* eleva o nível de abstração da comunicação entre os elementos de software do sistema, fornecendo métodos de acesso remoto aos elementos de software e, também, aos serviços de infra-estrutura.

As implementações do padrão *CORBA* (ORFALI; HARKEY; EDWARDS, 1996) (OMG, 2002a) da *OMG* são exemplos de *middleware* que oferecem uma série de serviços para a implementação de sistemas distribuídos. Outro exemplo de *middleware* que possibilita o desenvolvimento de sistemas distribuídos é *Jini* (SUN, 2001).

3.3 SISTEMAS DINÂMICOS

O mundo encontra-se em constante mudança. No âmbito tecnológico, por exemplo, a todo momento estão sendo produzidas e disponibilizadas novas tecnologias. Já no âmbito comercial, é possível citar o processo de globalização da economia mundial como um fator motivador de mudanças. A própria Internet, que se baseia essencialmente nos avanços tecnológicos da área de comunicações, redes e informática, tem proporcionado um ambiente adequado à criação e desenvolvimento de novos modelos de negócios na área comercial e de novas formas de interação nas áreas acadêmica e pessoal.

Os exemplos, acima, reforçam a noção de que o mundo tem um caráter dinâmico. É natural, portanto, que este dinamismo esteja presente também na área de desenvolvimento de software e que a disciplina de engenharia de software tenha que abordar também esta característica.

(DEIFEL; SALZMANN, 1999) afirmam que o escopo das estruturas de software está mudando: cada vez mais companhias vêem as vantagens de se utilizar *middleware* como *CORBA*, *DCOM* ou *RMI* e de começar a integração das estruturas distribuídas no ambiente empresarial. Em especial, os quesitos de adaptabilidade e dinamismo de sistemas distribuídos prometem vantagens no projeto de software.

Segundo (SALZMANN, 2000), com o aumento da infra-estrutura de rede e a necessidade de sistemas que estejam constantemente disponíveis, têm levado os sistemas distribuídos a oferecer estruturas mais dinâmicas e com maior adaptabilidade. *Jini Network Technology - Jini* é um dos exemplos de tecnologia de *middleware,* que está oferecendo os conceitos de dinamismo em um nível técnico, ou seja, está fornecendo suporte à construção de sistemas distribuídos com

características dinâmicas.

A seguir, *Jini* será apresentado descrevendo-se as suas características e o seu funcionamento.

3.4 *JINI NETWORK TECHNOLOGY - JINI*

Jini Network Technology (SUN, 2001) (JINI, 2001) é uma arquitetura aberta para a criação de sistemas distribuídos (figura 3).

Jini adota o termo **federação** para denominar uma coleção de elementos autônomos, que podem perceber a presença de outros elementos em uma rede, e cooperar com estes para realização de tarefas. Um elemento pode ser um componente de software, hardware ou ambos.

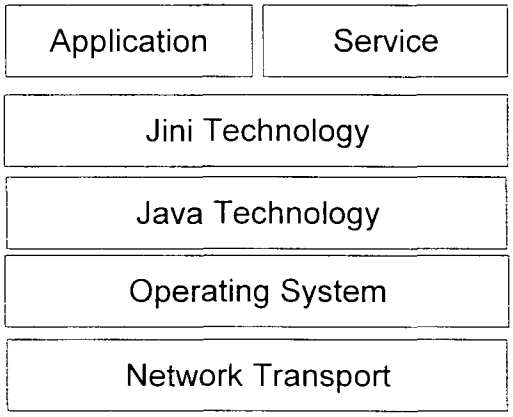

Figura 3 - Arquitetura do *Jini* (SUN, 2003d)

Jini permite, então, a criação de grupos federados de elementos autônomos na rede, de forma que alguns destes elementos são usuários e outros são serviços que oferecem os recursos requeridos por estes usuários. É importante ressaltar que um serviço pode, em determinado momento, ser também usuário de outro serviço.

Jini permite que serviços sejam incluídos ou excluídos de uma federação na rede de uma maneira flexível e dinâmica a qualquer momento. Imaginando que tenha sido criado um sistema distribuído, utilizando *Jini,* serviços podem ser incluídos, removidos ou substituídos da rede em tempo de execução do sistema.

Uma das características importantes do *Jini* é a capacidade de mobilidade de código. Essa mobilidade traduz-se na habilidade de se baixar e executar código dinamicamente, sendo parte essencial da arquitetura do *Jini* e chave para vários outros atributos da arquitetura como, por exemplo, o serviço de *Lookup* ou mesmo o serviço *JavaSpaces*.

A tecnologia *Jini* é constituída, basicamente, de elementos que podem ser organizados de acordo com a seguinte divisão:

- um conjunto de **elementos de software** que provêem uma infra-estrutura para uma federação de serviços em um sistema distribuído;
- um **modelo de programação** composto por um conjunto de interfaces que são utilizados na construção de serviços, tanto daqueles que fazem parte da infra-estrutura, quanto daquelas que se associaram a uma federação de serviços;
- **serviços** que podem fazer parte de uma federação *Jini* e que oferecem funcionalidades para qualquer outro membro da federação.

A tabela 1 apresenta os elementos que constituem a tecnologia *Jini* de acordo com a divisão acima:

Tabela 1 - Elementos constituintes do *Jini*

Infra-estrutura	*Discovery/Join* e *Lookup Service*, *Distributed Security*, e *Remote Method Invocation (RMI) Extensions*
Modelo de Programação Jini	*Distributed Transactions, Distributed Leases, e Distributed Events.*
Serviços	*JavaSpaces* e *Transaction Manager*

Mais detalhes com relação a cada um dos elementos constituintes da arquitetura do *Jini* podem ser encontrados em (SUN, 2001).

O conceito mais importante na arquitetura *Jini* é o de **serviço**. Um serviço é uma entidade que pode ser usada por uma pessoa, um programa ou, por outro serviço. Um exemplo poderia ser um serviço de impressão que aplicações clientes poderiam utilizar, enviando impressões através da rede.

A infra-estrutura do *Jini* e o seu modelo de programação foram construídos para permitir que os serviços sejam oferecidos/encontrados em uma federação da rede. Estes serviços fazem uso da infra-estrutura para fazer chamadas uns aos outros, para descobrirem-se uns aos outros e, ainda, para anunciar sua presença na rede.

A interação entre elementos de software que oferecem serviços (provedor de serviço) e usuários (aplicação cliente) em uma rede, utilizando *Jini*, pode ser descrita da seguinte forma:

Inicialmente o provedor de serviço procura um serviço de *Lookup* na rede utilizando o protocolo *discovery*.

Uma vez encontrado o serviço de *Lookup*, o provedor de serviço utilizará o protocolo *join* para se registrar nele. O provedor de serviço enviará ao serviço de *Lookup* um objeto contendo a interface de acesso ao serviço que ele está oferecendo, além de atributos descritivos do serviço (figura 4).

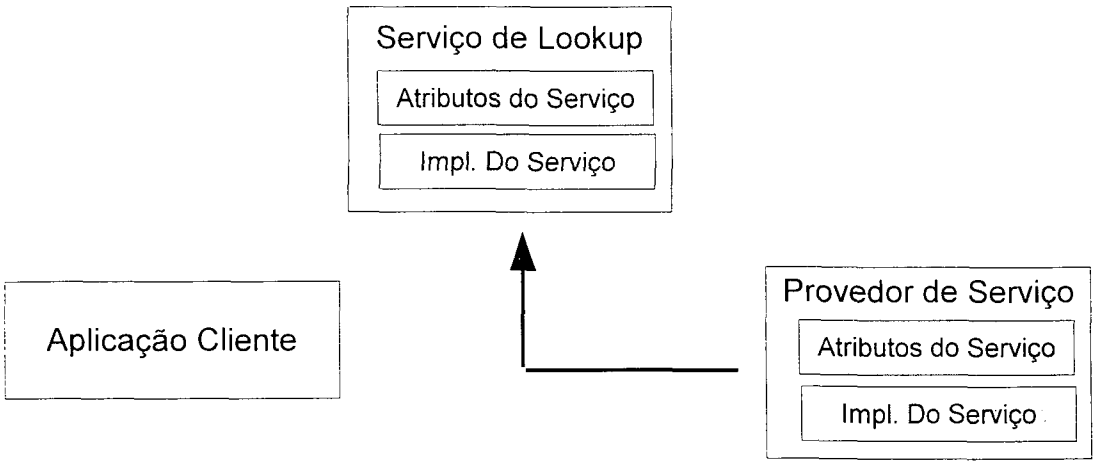

Figura 4 - Registro de um serviço no *Lookup* (SUN, 2001)

O registro de um provedor de serviços em um serviço de *Lookup* tem um tempo de expiração associado. Este suporte é fornecido pelo recurso de *Distributed Leasing* do *Jini*. Assim, é possível diminuir a possibilidade de que seja encontrado, em um serviço de *Lookup*, um registro de um provedor de serviço que já não se encontre mais disponível.

Uma aplicação cliente, então, para encontrar um determinado tipo de

serviço de que precisa, faz uma pesquisa no serviço de *Lookup,* verificando a interface e os atributos descritivos dos objetos que foram registrados nele.

Uma vez encontrado, no serviço de *Lookup*, um objeto com a descrição do serviço desejado, este objeto é enviado para a aplicação cliente e carregado por esta. Este objeto, além de conter a interface e os atributos descritivos do serviço, contém também, a implementação da forma de acesso ao serviço remoto, fornecido pelo provedor de serviço (figura 5).

Finalmente, a aplicação cliente utiliza o objeto carregado do serviço de *Lookup* para realizar o serviço desejado.

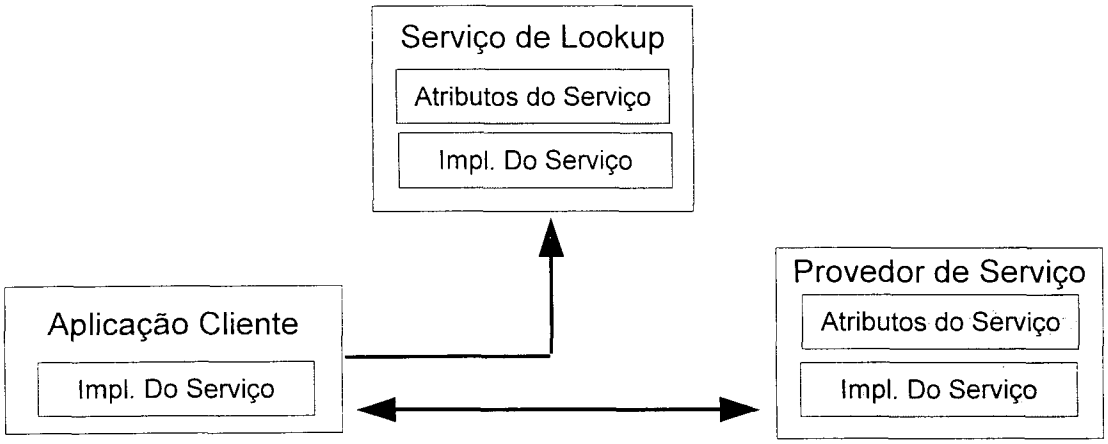

Figura 5 - Aplicação cliente recebe objeto e utiliza serviço (SUN, 2001)

Um cliente não precisa conhecer de antemão detalhes a respeito da implementação de um serviço remoto, nem da forma como acessará este serviço. Ele sabe apenas que estará lidando com a implementação de uma determinada interface escrita na linguagem *Java,* e que receberá, do serviço de *Lookup*, um objeto contendo a implementação de como ocorrerá a comunicação com o serviço remoto desejado.

Esta habilidade de mover objetos e código do provedor de serviços para o serviço de *Lookup* e, finalmente, para o cliente provê liberdade na escolha do padrão de comunicação entre os serviços de seus clientes. Esta movimentação de código garante, também, que a porção (o objeto) a ser executada no cliente esteja sempre atualizada e sincronizada em relação ao serviço remoto que está sendo

acessado. Isso por que, este objeto, a ser executado no cliente, é justamente aquele objeto que o próprio serviço remoto publicou no servidor de *Lookup* e que o cliente irá baixar do servidor de *Lookup*. Em outras palavras, a porção a ser executada no cliente é, na verdade, fornecida pelo serviço remoto que será acessado.

O objeto que é carregado pelo cliente pode ser implementado como um objeto que utiliza um protocolo de comunicação para fazer referência a um objeto remoto que implementa um serviço, ele pode ser um objeto que fornece um serviço ao ser executado localmente pela aplicação cliente ou, ainda, uma combinação das duas técnicas. Tal combinação, chamada também de *smart proxies*, implementa algumas das funções de um serviço localmente, enquanto mantém acesso a algumas funções remotas, disponíveis em um serviço da rede.

Além disso, no caso do objeto, sendo carregado pela aplicação cliente, prover acesso a um serviço remoto através da rede, a comunicação entre o objeto carregado do serviço de *Lookup* e o serviço remoto poderá ser implementado, utilizando-se *RMI,* ou outra tecnologia de comunicação qualquer, desde que haja suporte à sua implementação na linguagem *Java.*

Dessa forma, os métodos do objeto que implementa a comunicação com o serviço remoto podem ser escritos, utilizando um protocolo particular. Implementações diferentes de uma mesma interface de serviço podem utilizar protocolos diferentes para sua interação.

Do ponto de vista de uma aplicação cliente, não existe distinção entre os serviços que são implementados por objetos remotos, os serviços que são baixados e executadas localmente e os serviços que são implementados em hardware. Todos esses serviços irão tornar-se disponíveis na rede como se fossem objetos escritos na linguagem *Java*, de forma que a implementação de um serviço pode ser alterada, ou mesmo substituída por outra, sem que uma aplicação cliente precise tomar conhecimento deste fato.

3.5 *JAVASPACES*

JavaSpaces (SUN, 2002a), (FREEMAN; HUPFER; ARNOLD, 1999) é um

serviço *Jini,* baseado em um mecanismo de troca e persistência de objetos. Este serviço tem suas raízes no conceito de *TupleSpaces* originários das pesquisas relacionadas à linguagem de coordenação *Linda* (SCIENTIFIC COMPUTING ASSOCIATES, 2003).

> The concept of tuple space was pioneered by David Gelenrnter at the Yale Linda Group and was initially offered in Fortran Language and Ceel Language languages. However, various other TupleSpace systems exist today such as LiPS or ActorSpaces for many languages including SmallTalk, Java Language, Lisp Language and Ruby Language (DistributedRuby). For some reason, the Java Language is experiencing an explosion of generative systems including JavaSpaces, TSpace, PageSpaces, OpenSpaces, and so on. (TUPLESPACE, 2003)

Na abordagem tradicional de sistemas distribuídos, a interação entre os participantes é realizada através da utilização de protocolos, baseados na invocação de métodos remotos, ou seja, um processo envia uma mensagem diretamente a outro processo, requisitando a execução de determinada tarefa.

Por outro lado, muitas aplicações distribuídas podem ser modeladas como um fluxo de objetos entre participantes. Nesta abordagem, protocolos são baseados na movimentação de objetos para dentro e para fora de espaços compartilhados de troca como, por exemplo, a implementação da tecnologia *JavaSpaces* (SUN, 2002a) (neste trabalho o termo "espaço compartilhado" será utilizado para referenciar a implementação do serviço *JavaSpaces*).

Com *JavaSpaces,* os processos ao invés de se comunicarem diretamente, através de trocas de mensagens, eles passam a coordenar suas atividades através da troca de objetos utilizando um espaço compartilhado.

Quando processos comunicam-se e sincronizam suas atividades através deste espaço compartilhado, eles passam a ter um acoplamento fraco. Essa natureza dos programas, baseados em espaços compartilhados, é significativa e poderosa, colaborando para a implementação de aplicações distribuídas flexíveis e escaláveis.

Serviços *JavaSpaces* podem fornecer persistência distribuída de objetos na linguagem *Java*, podendo ser utilizados para prover um mecanismo de armazenamento simples de objetos que podem posteriormente ser pesquisados e recuperados. Apesar disso, os serviços *JavaSpaces* foram projetados para resolver

problemas na computação distribuída e não para serem usados como um repositório de dados. A funcionalidade do serviço *JavaSpaces* está entre um sistema de arquivos e um banco de dados, não sendo nenhum dos dois.

O serviço *JavaSpaces* (SUN, 2002a) trata de acesso concorrente, armazena e recupera entradas de forma atômica e provê a implementação de um mecanismo distribuído de transações, fornecendo, dessa forma, uma plataforma para o desenvolvimento de sistemas distribuídos que simplifica o seu projeto e implementação.

O serviço *JavaSpaces* armazena objetos do tipo *entry* (entrada), que são classes em *Java* que implementam a interface `net.Jini.core.entry.Entry`.

Uma entrada pode ser escrita (utilizando-se a operação *write*) em um serviço *JavaSpaces*, podendo depois ser recuperada. Para recuperação da entrada existem dois tipos de operações: *read* e *take*. Na execução das operações de recuperação, são utilizadas *templates,* que são objetos do tipo *entry* que têm todos ou, alguns de seus campos inicializados com valores específicos. Uma *template* será usada na recuperação de entradas, cujos valores de seus campos combinem exatamente com os valores dos campos da *template* (os campos com valores nulos funcionam como máscaras no estilo de *wildcards*).

Uma operação de leitura (*read*) retorna uma entrada que combine com a *template* utilizada ou, a indicação que nenhuma entrada foi encontrada. Uma operação de retirada (*take*) funciona como a operação de leitura, com a diferença de que se uma entrada que satisfaça a procura for encontrada, esta é retirada do espaço.

Pode-se, também, requisitar a um serviço *JavaSpaces* que este notifique um processo quando uma entrada que combine com determinada *template* for escrita no espaço.

O serviço *JavaSpaces* (SUN, 2002a) utiliza o suporte a transações, distribuídas da arquitetura do *Jini,* para garantir que uma série de operações sobre um ou mais espaços sejam realizadas de forma atômica.

JavaSpaces, também, faz uso dos recursos de *Distributed Leasing* do *Jini.* Assim, da mesma forma que os serviços registrados em um serviço de *Lookup*, em uma federação *Jini,* as entradas (objetos) publicadas no espaço *JavaSpaces*

possuem um tempo de expiração (*lease*).

Outro fator importante do serviço *JavaSpaces*, é que ele armazena objetos em seu espaço. Dessa forma, quando um objeto é recuperado, não se recupera apenas seus dados, mas também o seu comportamento. Um objeto recuperado por uma aplicação cliente, pode, por exemplo, ser executado pela própria aplicação cliente.

Portanto, a vantagem da utilização de *JavaSpaces* e, conseqüentemente , de espaços compartilhados, consiste na disponibilização de um plataforma simples que fornece suporte à solução de vários aspectos comuns à sistemas distribuídos. Na implementação de um sistema distribuído, a utilização do serviço *JavaSpaces* ajuda a simplificar a solução de questões clássicas como persistência, comunicação e sincronização entre processos, atomicidade de operações, etc., uma vez que o suporte à solução destes problemas, já se encontra incorporada ao próprio serviço e a arquitetura do *Jini*.

3.6 *RIO*

Rio (RIO PROJECT, 2001b) é um projeto desenvolvido pela comunidade *Jini* (ABOUT, 2003). Seu objetivo é disponibilizar aos desenvolvedores um *framework* que facilite o desenvolvimento de serviços, baseados na tecnologia *Jini*. O projeto *Rio* define uma arquitetura, baseada em *Java* e *Jini*, que adiciona recursos, serviços e ferramentas ao desenvolvimento de sistemas distribuídos e fornece, também, um *framework* que implementa justamente os recursos, serviços e ferramentas definidos na arquitetura.

Rio pode ser visto como uma camada superior à camada *Jini,* que fornece recursos para facilitar a programação de serviços *Jini*. Além disso, a arquitetura definida pelo projeto *Rio* estende a tecnologia *Jini* nas áreas de qualidade de serviço (*QoS*), entrega dinâmica e detecção e recuperação de falhas.

Rio provê um modelo de componentes chamados *Jini Service Beans* (*JSBs*), que tem por objetivo eliminar a complexidade, relacionada ao desenvolvimento de serviços *Jini*. *JSBs* são objetos *Java* que provêm um modelo de programação simples de se usar.

Já o *framework,* fornecido pelo projeto *Rio,* provê recursos ao desenvolvedor de software, para facilitar a programação de sistemas distribuídos com características dinâmicas (RIO PROJECT, 2001b). Além disso, o *framework* mantém o acesso à *API* do *Jini,* possibilitando que o desenvolvedor também faça uso das classes do próprio *Jini*.

Os serviços desenvolvidos, utilizando-se o modelo *JSB,* são executados na rede em nós chamados de *cybernodes*. Estes *cybernodes* podem ser vistos como containers dinâmicos (*Dynamic Container*), onde os *JSBs* podem ser carregados e instanciados, podendo, inclusive, existir vários *JSBs* em execução ao mesmo tempo.

O conceito de qualidade de serviço adotado pelo projeto *Rio,* fundamenta-se no fato de que dispositivos, que oferecem recursos computacionais, possuem capacidades, como por exemplo *CPU,* disco, largura de banda, etc., e elementos de software possuem requisitos, como por exemplo, tempo de resposta, taxa de transmissão, hardware mínimo, etc.. A arquitetura do *Rio* fornece mecanismos tanto para especificar os requisitos necessários ao funcionamento de um serviço, como para especificar as capacidades computacionais presentes em um nó da rede. Uma vez em funcionamento, um sistema baseado na arquitetura do *Rio,* procurará combinar os requisitos dos serviços com as capacidades dos nós no momento de distribuir e alocar os serviços na rede.

Rio também introduz o conceito de *Operational String* (RIO PROJECT, 2001a), que é basicamente um conjunto de serviços ou, elementos de software, que quando combinados provêm outro serviço na rede.

> An Operational String describes an aggregated collection of application and/or infrastructure software assets that when put together provide a specific coarse grained service on the network. The software assets are envisioned to be Java service objects. (RIO PROJECT, 2001a) p.3

A configuração de uma *Operational String* é feita através de um documento

XML, que especifica os serviços e/ou os elementos de software que a compõem, especificando, também, as características e requisitos de cada serviço e da própria *Operational String*.

Através do *Operational String* é possível identificar quais serviços e/ou elementos de software precisam ser alocados e instanciados, para que seja possível fornecer um determinado serviço em uma rede. Também é através dele que se obtém as informações a respeito das características e requisitos de cada serviço que, por sua vez, são utilizados no processo de escolha do nó (*cybernode*) mais adequado para sua execução.

Outro recurso, fornecido pelo *Rio,* é um modelo de eventos distribuídos que estende o modelo fornecido pelo *Jini*. O modelo de eventos distribuídos permite que objetos que sofram uma mudança em seu estado possam notificar outros objetos, espalhados no ambiente distribuído. Um objeto pode registrar, através dos serviços da rede, o interesse em receber notificações de determinado tipo.

O modelo de eventos distribuído funciona de acordo com o *pattern publish-subscribe* ou *observer-observable,* de forma que, são definidos produtores de eventos que enviam notificações quando seus estados sofrem mudanças e consumidores que recebem estas notificações. No modelo de eventos distribuídos do *Rio*, os produtores de eventos têm, ainda, a opção de escolher a política de entrega das notificações (RIO PROJECT, 2001b).

A arquitetura *Rio,* também, fornece mecanismos para auxiliar na coleta, análise e visualização de métricas das aplicações locais e distribuídas. Essas métricas devem ser definidas pelo desenvolvedor da aplicação. O *framework,* fornecido pelo projeto *Rio*, permite que o desenvolvedor defina as métricas, implementando o método que produz o cálculo das mesmas. *Rio,* por sua vez, dispõe de um serviço na rede que serve de repositório para as métricas calculadas e permite também a visualização gráfica das mesmas.

Outro recurso, encontrado no *framework* fornecido pelo *Rio,* é uma infra-estrutura abstrata para o gerenciamento de recursos locais aos nós, criando *resource pools,* ou seja, repositórios de recursos que podem precisar ser alocados freqüentemente ou que precisam ser, de alguma forma, limitados.

O *framework* fornecido pelo *Rio* define pelo menos três tipos de *resource*

pools (RIO PROJECT, 2001b):

- ***ThreadPool*** para gerenciar uma coleção de linhas de execução de programas *Java;*
- ***ObjectPool*** para gerenciar uma coleção de objetos *Java* genéricos;
- ***DatabaseConnectionPool*** para gerenciar conexões *JDBC* à servidores de banco de dados.

Além disso, existe também, uma série de serviços de infra-estrutura, responsáveis por proporcionar recursos chaves à arquitetura definida pelo *Rio.* Todos estes serviços são objetos gerenciáveis, no sentido em que todos eles fornecem uma interface gráfica para sua administração, implementada conforme a especificação proposta pelo projeto *ServiceUI* (VENNERS, 2003). Todos eles são configuráveis e fornecem suporte para a geração de registros de *Log* e acompanhamento de suas atividades para serem usados no diagnóstico do seu funcionamento. Exemplos destes serviços são: *cybernode, Provision Monitor, Watchsmith* e *Logger* (RIO PROJECT, 2001b).

O projeto *Rio* fornece, também, outras ferramentas administrativas que podem ser usadas para realizar tarefas, como a inicialização de serviços, leitura, entrega e visualização de *Operational Strings* e o acompanhamento do estado de uma *Operational String* e dos serviços que a compõem entre outros (RIO PROJECT, 2002).

3.7 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados conceitos e tecnologias ligadas à área de sistemas distribuídos. Em particular, foram apresentados a tecnologia *Jini*, o serviço *JavaSpaces* e o projeto *Rio.* Estes três últimos foram utilizados na implementação do protótipo do *DART,* conforme descrito na seção 5.2.

No próximo capítulo, são apresentados conceitos relacionados às especificações dos padrões *MOF* e *XMI* da *OMG* e ao suporte à persistência de artefatos para o ambiente *DiSEN.*

4 METADADOS E A PERSISTÊNCIA DE ARTEFATOS NO AMBIENTE *DiSEN*

4.1 INTRODUÇÂO

Como visto no capítulo 2, atualmente, vários fabricantes de ferramentas de modelagem têm adotado a utilização de repositórios de metadados no armazenamento de artefatos, produzidos por estas. Vários destes repositórios apresentam conformidade com as especificações dos padrões *MOF* e *XMI* da *OMG.*

O suporte a estas especificações da *OMG* têm sido incorporado, também, à diversas ferramentas de modelagem e desenvolvimento de software (ver seção 4.3.1).

Este capítulo apresenta os principais conceitos, relacionados às especificações *XMI e MOF* da *OMG,* e, também, ao suporte à persistência de artefatos para o ambiente *DiSEN*. Além disso, é apresentada, também, uma lista de características desejáveis a este suporte.

4.2 META OBJECT FACILITY – MOF

A especificação de *Meta Object Facility - MOF* (OMG, 2002b) estabelece um padrão para a definição de metadados. Segundo (OMG, 2003c), *MOF* fornece o suporte à descrição de qualquer tipo de metadados que possam ser descritos utilizando-se técnicas de modelagem de objetos.

A especificação do padrão *MOF*, assim como o presente trabalho, adota a definição de que metadados é qualquer tipo de dado que descreve outro dado. Conseqüentemente, um modelo é definido como uma coleção de metadados que descrevem uma coleção de dados relacionados. Nesta especificação, é apresentada uma arquitetura conceitual para descrição de metadados, chamada de arquitetura *MOF* e um modelo, chamado modelo *MOF*, que é na realidade uma linguagem de modelagem de objetos semelhante à *UML.*

Essa arquitetura de metadados *MOF* (OMG, 2002b) é apresentada como

uma arquitetura de 4 camadas, onde cada camada de metadados descreve os dados da camada seguinte.

As camadas são denominadas M0, M1, M2 e M3. A letra M de cada camada diz respeito ao nível de descrição de dados, por exemplo M0, é composta por dados, ou seja nível 0 de descrição de dados. Já a camada M1 é composta por metadados que descrevem os dados da M0, a camada M2 é composta por meta-metadados, e assim por diante. Uma breve descrição das camadas é apresentada a seguir:

- **camada 1, camada de dados ou M0:** é composta pelos dados, pelas informações;
- **camada 2, camada de modelos ou M1:** é composta pelos metadados que descrevem os dados da camada 1, ou seja, contém os modelos de dados enquanto que a camada 1 contém os dados;
- **camada 3, camada de metamodelos ou M2:** é composta por metadados que descrevem os modelos de dados da camada 2, ou seja, contém metamodelos;
- **camada 4, camada de meta-metamodelos, ou M3:** contém o modelo *MOF* que é na realidade um meta-metamodelo utilizado para descrever os metamodelos da camada 3.

Segundo a especificação *MOF* (OMG, 2002b), a camada 4 contém, justamente, uma linguagem de modelagem semelhante à *UML*, para ser usada na descrição destes metamodelos e (OMG, 2003c) afirma, ainda, que esta pode ser vista como uma linguagem abstrata para a definição de metamodelos.

A especificação *MOF* também define o chamado mapeamento *MOF IDL*. Esse mapeamento permite que a partir de uma especificação de um metamodelo *MOF* (um metamodelo descrito pelo modelo *MOF*), seja produzido um serviço de metadados *CORBA*. Segundo (OMG, 2003c), o mapeamento *MOF IDL* é um conjunto de templates padrão que faz o mapeamento de um metamodelo *MOF* para um conjunto de interfaces *IDL CORBA* correspondente, de forma que, dado um modelo de metadados *MOF* como entrada, tem-se como resultado um conjunto de interfaces *IDL CORBA* que podem ser usadas na representação dos metadados.

Essas interfaces permitem a criação, alteração e o acesso aos metadados na forma de objetos *CORBA*.

Além do mapeamento *MOF IDL*, a especificação *MOF* também fornece um conjunto de interfaces *IDL CORBA* para os objetos *CORBA* que representam um metamodelo *MOF*. Isso é possível, uma vez que o modelo *MOF* é definido utilizando-se o próprio modelo *MOF* na sua definição. Conceitualmente, o modelo *MOF* localiza-se na camada 4 ou M3 e este estaria em conformidade com um modelo de uma camada 5 ou M4 que seria isomórfica ao modelo *MOF*. Assim, é possível aplicar, também, o mapeamento *MOF IDL* ao próprio modelo *MOF* obtendo-se o conjunto de interfaces *IDL CORBA* que é fornecido na sua especificação.

Finalmente, a especificação *XMI* (OMG, 2003c) é definida com base no padrão *MOF*. Na realidade, *XMI* é definida como uma tecnologia de intercâmbio de metadados baseados no padrão *MOF*. *XMI* é, sucintamente, apresentada a seguir.

4.3 XML METADATA INTERCHANGE - XMI

XMI consiste basicamente de um formato de intercâmbio de metadados. Segundo (OMG, 2003c), o principal propósito da tecnologia *XMI* é possibilitar a troca de metadados entre ferramentas de modelagem que utilizem *UML* e repositórios de metadados em um ambiente distribuído e heterogêneo.

Segundo (SCHLAPBACH, 2001), *XMI* melhora o gerenciamento e interoperabilidade de metadados em ambientes distribuídos em geral e, particularmente, em ambientes distribuídos de desenvolvimento de software.

De uma forma geral, *XMI* tem como base três tecnologias tidas como padrão:

- ***XML - eXtensible Markup Language*** criada pelo consórcio *W3C* (W3C, 2000);
- ***UML - Unified Modeling Language,*** *um padrão de modelagem da OMG;*
- ***MOF - Meta Object Facility****, um padrão de repositório de metadados também da OMG.*

Enquanto a *UML* define uma linguagem de modelagem orientada à objetos adotada como padrão por uma grande variedade de ferramentas de modelagem do mercado, o padrão *MOF* define um *framework* extensível para definição de modelos de metadados e fornece ferramentas com uma interface programável para o armazenamento e acesso de metadados em repositórios. A tecnologia *XMI,* por sua vez, permite o intercâmbio de metadados na forma de cadeias de caracteres (*streams*) ou documentos em um formato padrão definido em *XML*.

Na realidade, *XMI* pode ser vista como um par de mapeamentos, sendo o primeiro entre metamodelos *MOF* e *Document Type Definition - DTDs* (W3C, 2000), e o segundo entre metadados *MOF* e documentos *XML* (OMG, 2003c).

Se considerado em um contexto mais amplo, ou seja, além do padrão *MOF*, *XMI* pode ser vista como um formato de intercâmbio de metadados independente da tecnologia de *middleware* (OMG, 2003c). Ela pode ser utilizada na troca de metadados entre qualquer ferramenta e repositório que seja capaz de codificar e decodificar documentos *XMI*. Isso significa que, apesar de existir um forte relacionamento entre *UML*, *MOF* e *XMI*, este último fornece uma tecnologia a ser usada no intercâmbio de metadados que não precisam ser baseados no padrão *MOF*, é necessário apenas que exista um mapeamento entre o metamodelo dos metadados e o documento *XMI* (OMG, 2003c).

A especificação do padrão *XMI* contém um conjunto de regras de produção para a geração de definições *DTD* à partir de metamodelos *MOF* e um conjunto de regras de produção para geração de documentos *XML,* baseados em metadados *MOF*.

4.3.1 FERRAMENTAS DE MODELAGEM E O SUPORTE À *XMI*

Nos dias de hoje, várias ferramentas de modelagem já dispõe de suporte à utilização de arquivos segundo a especificação *XMI*.

Em (JECKLE, 2003), é apresentada uma tabela comparativa de ferramentas de modelagem *UML* existentes, indicando, inclusive, aquelas que já possuem

suporte à *XMI*. Entre as muitas ferramentas de modelagem *UML* que, atualmente, apresentam suporte ao padrão *XMI* é possível citar:

- *Rational Rose* RATIONAL (RATIONAL, 2002);
- *Poseidon for UML* (GENTLEWARE, 2003a);
- *Borland Together Control Center* (BORLAND SOFTWARE CORPORATION, 2003);
- MagicDraw (NO MAGIC, 2003);
- *Visual Paradigm* (VISUAL, 2003), etc.

Apesar disso, verifica-se, na prática, que existem, ainda, algumas incompatibilidades entre os arquivos *XMI*, gerados pelas ferramentas de diferentes fabricantes.

Um dos motivos desta incompatibilidade é o fato de que as especificações *UML*, *MOF* e *XMI* ainda são relativamente novas e, portanto, incompletas estando em evolução.

Por exemplo, nenhum dos dois padrões, *UML* ou *XMI*, especificam claramente como lidar com as informações relativas à geometria dos modelos *UML*. A especificação *UML* define os elementos que podem ser utilizadas na modelagem de um sistema e, também, como cada um destes elementos podem ser apresentados graficamente. O que falta é, justamente, como ligar um elemento de modelagem de um modelo *UML* à representação gráfica de um diagrama do modelo (NENTWICH, 2000).

> UML is a modeling language for object-oriented software systems with a strong emphasis on a graphical representation. A mechanism for exchanging model information is included in the current UML standard. However, it only supports the definition of elements in a model, the graphical information how these elements are laid out and presented in diagrams is not included. (GENTLEWARE, 2003b)

Portanto, devido à falta de especificação com respeito à representação gráficas dos modelos, a saída encontrada pelos fabricantes de ferramentas têm sido a extensão do padrão *XMI* com *tags* proprietárias. Sem estas extensões nenhuma ferramenta é capaz de reconstruir a disposição original dos elementos de

modelagem em um diagrama.

Segundo (OMG, 2003b), apesar da *UML* definir uma linguagem precisa, isso não impede que sejam feitas melhorias nos conceitos de modelagem. Na verdade, espera-se que novas tecnologias influenciem as novas versões da *UML*. Ao mesmo tempo que, *UML* pode ser usada como base para definição de várias técnicas avançadas, ela pode ser estendida sem que para isso seja necessário redefinir-se o cerne da *UML*.

Assim, já existem esforços no sentido de se completar as especificações *UML* e *XMI* com padrões para a representação gráfica de elementos de modelagem e, também, para a transferência de diagramas (NENTWICH; EMMERICH; FINKELSTEIN; ZISMAN, 2000) (JÉZÉQUEL; HUSSMAN, 2002) (GENTLEWARE, 2003b).

Com o objetivo de verificar, na prática, a incompatibilidade entre diferentes arquivos *XMI*, foram realizados testes de leitura, a partir do repositório de metadados *MDR*, de artefatos gerados por diferentes ferramentas de desenvolvimento de software. Os resultados destes testes estão representados na tabela 2.

Tabela 2 - Testes com artefatos *UML* de diferentes ferramentas

Ferramenta	Armazenamento do artefato gravado em *XMI*	Leitura do arquivo *XMI* recuperado
Poseidon for UML	OK	OK
Rational Rose	OK com aplicação de *XSLT*	OK
MagicDraw	OK com aplicação de *XSLT*	Não

Verificou-se que alguns arquivos *XMI*, gerados por estas ferramentas, tiveram que passar por adaptações para que pudessem ser lidos pelo repositório e ter seu conteúdo armazenado sem erros. De qualquer forma, as alterações que se fizeram necessárias puderam ser realizadas através de um simples editor de texto, sendo facilmente automatizáveis, podendo, neste processo, ser utilizado por exemplo *XSLT*.

Com relação à aplicação de arquivos *XSLT* na transformação de arquivos *XMI*, é possível citar o trabalho de (LANGIANO, 2003) onde são apresentados

alguns arquivos *XSLT,* criados para eliminar erros no processo de leitura de arquivos *XMI* pelo *MDR,* gerados por diferentes ferramentas de modelagem. Ao mesmo tempo, o projeto *AndroMDA,* também, utiliza arquivos *XSLT* para fazer a leitura sem erros de arquivos *XMI,* gerados por diferentes ferramentas de modelagem, para o *MDR.* Estes arquivos estão disponíveis na área de *CSV* do projeto.

4.4 JAVA METADATA INTERFACE - JMI

A especificação de *Java Metadata Interface - JMI* define uma infra-estrutura dinâmica e neutra com respeito à plataforma, que possibilita a criação, armazenamento, descoberta e intercâmbio de metadados (DIRCKZE, 2002). *JMI* baseia-se na especificação *MOF* da *OMG* (OMG, 2002b) e define um mapeamento *Java* para o padrão *MOF.*

Enquanto *MOF* fornece um conjunto de construções para definição de metadados, juntamente com um mapeamento destas construções para *IDL CORBA*, *JMI* fornece um mapeamento *Java* para *MOF,* definindo dessa forma um modelo de programação *Java* para acesso à metadados.

Utilizando *JMI*, por exemplo, é possível que ferramentas com modelos baseados em *UML,* em acordo com o padrão *MOF*, tenham as interfaces *Java* para seus modelos geradas automaticamente. Além disso, o intercâmbio de metamodelos e metadados é possível através do suporte que *JMI* tem para o padrão *XMI* (DIRCKZE, 2002).

Através da tecnologia *JMI*, qualquer aplicação *Java* pode obter acesso à metadados e metamodelos em conformidade com o padrão *MOF.* Além disso, da mesma forma que *MOF* especifica um conjunto de regras para geração automática de *IDLs CORBA*, *JMI* também especifica um conjunto de regras que permitem a geração de um conjunto de interfaces de programação *Java* para a manipulação das informações específicas de uma instância de um metamodelo.

De uma forma geral, *JMI* fornece ao desenvolvedor *Java* pelo menos os seguinte benefícios:

- um *framework* para metadados que fornece um modelo de programação *Java* para acesso aos metadados;
- um *framework* para integração e interoperabilidade de aplicações *Java;*
- integração com a arquitetura de metadados e modelagem da *OMG*.

Segundo (DIRCKZE, 2002), espera-se que implementações da especificação *JMI* forneçam uma infra-estrutura de gerenciamento de metadados que facilite a integração de aplicações, ferramentas e serviços.

4.5 METADATA REPOSITORY - MDR

Metadata Repository - MDR (SUN, 2002b) é uma implementação de um repositório de metadados, baseado no padrão *MOF* para trabalhar de forma integrada à ferramenta de código aberto *Netbeans* (SUN, 2003a). A interface do repositório é baseada na especificação *JMI,* de forma a auxiliar a sua incorporação à ferramenta.

Entre as propriedades do *MDR,* pode-se destacar o suporte à importação e exportação de documentos no padrão *XMI* e a geração automática de interfaces *Java* para acessar metadados, descritos por metamodelos no padrão *MOF*.

Além de ter sido criada para trabalhar de forma integrada à ferramenta *Netbeans*, o *MDR* também pode ser executado individualmente e fornece suporte à criação, ao armazenamento e à recuperação de metadados.

Como parte de sua arquitetura, o *MDR* fornece um conjunto de interfaces que provêm métodos para indexação e persistência dos dados do repositório (SUN, 2002b). Apesar de contar apenas com implementações destas interfaces para persistência em memória e em um banco de dados próprio, utilizando o sistema de arquivos e árvore B, é possível adicionar outras implementações destas interfaces para incluir um esquema de persistência alternativo, baseada, por exemplo, em servidores de banco de dados relacionais através de *JDBC*.

Na utilização do repositório *MDR*, um usuário (desenvolvedor de software) pode criar um modelo *MOF* que descreva o modelo de metadados dos artefatos que

deseja armazenar. Ele pode, então, carregar este modelo *MOF* no repositório de metadados e criar dentro do próprio repositório, uma espécie de sub-repositório específico para o seu modelo de metadados. Esse sub-repositório pode ser utilizado para armazenar artefatos que sigam o modelo de metadados que foi especificado.

Tanto o modelo de metadados, seguindo a especificação *MOF*, quanto os artefatos a serem armazenados, podem ser lidos e gravados em arquivos *XMI*. Além disso, interfaces *Java* de acordo com a especificação *JMI*, podem ainda, ser geradas para acessar os elementos do repositório. Um exemplo, seria gerar as interfaces *JMI* para serem utilizadas por uma aplicação *Java* para que esta pudesse manipular os metadados de um determinado modelo dentro do repositório.

Dentre as ferramentas e projetos que utilizam o repositório de metadados *MDR* é possível citar a ferramenta *UML2MOF* (MATULA, 2003), e o projeto *AndroMDA* (BOHLEN; ANDROMDA TEAM, 2003).

A ferramenta *UML2MOF* possibilita que a partir de um modelo *UML* seja gerado um modelo *MOF* correspondente. A implementação da ferramenta é composta de apenas duas classes que, utilizando a *API* do *MDR*, realizam os seguintes passos:

- inicialmente é carregado no repositório, através da leitura de um arquivo *XMI*, um modelo *MOF* com a descrição da *UML* 1.4;
- é criado um sub-repositório para modelos *UML* 1.4, no qual é carregado, a partir de outro arquivo *XMI*, o modelo *UML* que se deseja converter;
- é criado um sub-repositório para o modelo *MOF* resultante;
- é feita a leitura de cada um dos elementos do modelo *UML* em questão e criam-se os elementos do novo modelo *MOF* correspondente;
- o novo modelo *MOF* resultante é gravado em um novo arquivo, também, no formato *XMI*.

UML2MOF, então, utiliza o repositório de metadados como um repositório temporário para fazer a conversão entre artefatos, baseados em especificações diferentes, ou seja com metamodelos diferentes (neste caso *UML* e *MOF*). *UML2MOF* faz uso, ainda, da capacidade que o repositório tem de ler e gravar arquivos no formato *XMI* e da *API* do *MDR* para ter acesso aos elementos dos dois

modelos.

A implementação da ferramenta *UML2MOF* é facilitada, justamente, pelas funcionalidades que ela utiliza do repositório de metadados, é por isso que a sua implementação consiste de duas pequenas classes.

Já o projeto *AndroMDA* permite que a partir modelos *UML* seja gerado código *Java* de classes, segundo a especificação *Enterprise Java Beans* (DEMICHIEL; YAÇINALP; KRISNAN. 2001). De forma semelhante à ferramenta *UML2MOF,* o projeto *AndroMDA* utiliza o repositório de metadados *MDR* para fazer a leitura de um modelo *UML,* armazenado em um arquivo no formato *XMI*, e com base neste, fazer a geração do código *Java* segundo a especificação *EJB*.

Além do *UML2MOF* e do *AndroMDA,* em (SUN, 2003b) é apresentada uma lista de alguns projeto e fabricantes de software que estão atualmente utilizando o *MDR* em seus produtos.

4.6 METODOLOGIA PARA DESENVOLVIMENTO DE SOFTWARE DISTRIBUÍDO MDSODI

A *Metodologia para Desenvolvimento de Software Distribuído - MDSODI* é uma metodologia dirigida a *use-case*, centrada na arquitetura e de desenvolvimento iterativo incremental. *MDSODI* leva em consideração já nas fases iniciais do projeto algumas características de sistemas distribuídos tais como: paralelismo/concorrência, comunicação, sincronização e distribuição. Mais detalhes podem ser encontrados em (GRAVENA, 2000).

MDSODI divide o processo de desenvolvimento de software em fases e para cada uma delas define seu objetivo, as atividades que devem ser realizadas e os artefatos que devem ser produzidos.

As fases descritas na *MDSODI* são:

- **requisitos**: tem como objetivo principal identificar as funcionalidades necessárias para o sistema, a ser desenvolvido, a partir das necessidades do usuário;
- **análise:** tem como objetivo analisar os requisitos descritos na fase

anterior, refinando e estruturando de forma a adquirir um entendimento mais preciso da descrição dos requisitos;

- **projeto:** tem como objetivo moldar o sistema, incluindo também os requisitos não funcionais e outras considerações não identificadas nas fases anteriores (ex: linguagem de programação, interface com usuário, reuso de componentes, etc.);
- **implementação:** o principal objetivo desta fase é a construção/implementação do sistema, tendo como base aspectos identificados nas fases anteriores, além de identificar aspectos que não foram totalmente cobertos na fase de projeto;
- **teste:** tem como objetivo os testes necessários para que se atinja um produto final de qualidade.

MDSODI faz uso da *UML* e define extensões, para elaboração de seus diagramas incorporando, além disso, algumas características da *MOOPP* (*Metodologia Orientada a Objetos para Processamento Paralelo*) (HUZITA, 1995) para representação gráfica de tipos de objetos e classes tendo como base o paralelismo. É importante notar que a *UML* define um padrão para linguagem de modelagem, não para processo de software, ficando livre ao desenvolvedor, a escolha do processo de software que mais se adeqüe à sua realidade.

Percebe-se que esta metodologia tem uma preocupação em fornecer mecanismos para representar, adequadamente, as características de sistemas distribuídos e, assim, abordar de forma clara estes aspectos durante o processo de desenvolvimento de software. Alguns destes mecanismos são extensões da notação *UML* que foram propostas em trabalhos de pesquisa anteriores (GRAVENA, 2000), (HUZITA, 1999).

4.7 *DISTRIBUTED SOFTWARE ENGINEERING ENVIRONMENT - DiSEN*

Em (PASCUTTI, 2002) é proposto um ambiente distribuído de desenvolvimento de software chamado *DiSEN,* com suporte à agentes e no qual a

MDSODI está inserida. Entende-se, que neste ambiente, estarão sendo utilizadas várias ferramentas de desenvolvimento de software, onde cada uma delas produziria artefatos como resultado do trabalho do desenvolvedor de software.

A arquitetura do *DiSEN* define os elementos de software que possibilitarão diferentes ferramentas trabalharem de maneira cooperativa, permitindo que desenvolvedores distribuídos em locais remotos possam colaborar no processo de desenvolvimento de software.

Na figura 6, é apresentado um desenho da arquitetura do *DiSEN,* que consiste em um detalhamento do desenho original apresentado em (PASCUTTI, 2002). Esta figura, apresenta os mesmos elementos ilustrados no desenho original, mas, ao mesmo tempo, apresenta mais detalhes a respeito dos elementos constituintes de cada camada da arquitetura.

Um dos elementos propostos na arquitetura do *DiSEN* (PASCUTTI, 2002) é o suporte à persistência de informações que possibilita às diversas ferramentas de desenvolvimento de software o armazenamento de seus artefatos.

Antecipa-se que em um cenário de desenvolvimento de software, baseado no *DiSEN*, poderão existir vários desenvolvedores, localizados em nós distintos da rede, cada qual utilizando uma ou mais ferramentas de desenvolvimento de software.

Utilizando estas ferramentas, os desenvolvedores de software produzirão artefatos, relacionados às diferentes fases do processo de desenvolvimento de software. A princípio, imagina-se que cada uma destas ferramentas será uma aplicação completa no sentido em que permitirá que o desenvolvedor trabalhe de forma isolada da rede, armazenando localmente os artefatos por ele produzidos.

Esta forma de trabalho é útil quando a máquina cliente, por um ou outro motivo, está impossibilitada de utilizar o ambiente *DiSEN*. Casos de perda, ou mesmo, de inexistência de conexão com uma rede, são situações nas quais será necessário existir a possibilidade de se trabalhar desconectado do ambiente *DiSEN*.

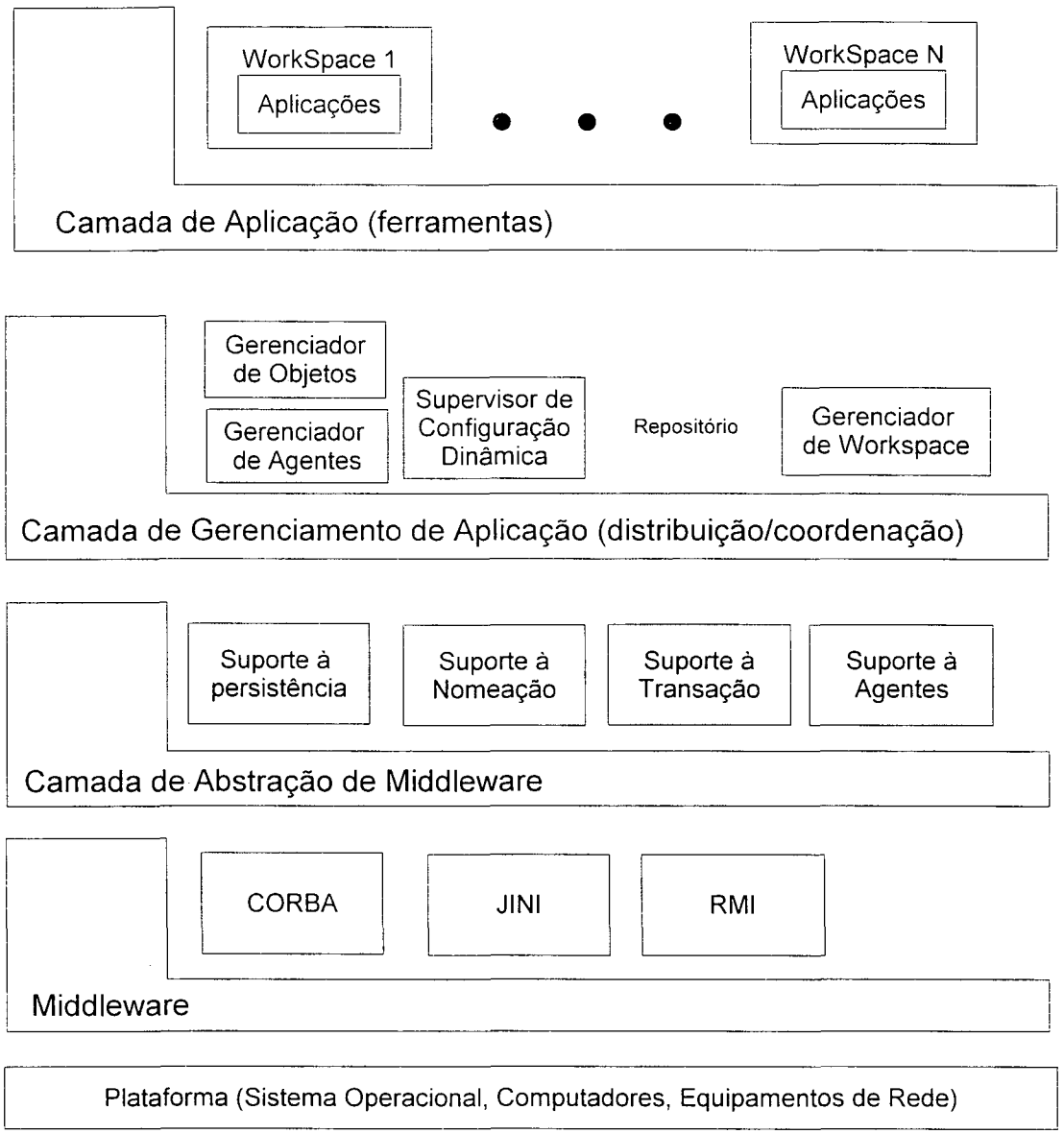

Figura 6 - Arquitetura do *DiSEN*

Apesar da necessidade de se poder trabalhar independentemente do ambiente *DiSEN*, a persistência local de artefatos traz dificuldades, pelo menos, nos seguintes casos:

- **se o desenvolvedor de** software **mudar de máquina de trabalho:** neste caso ele precisa ter o cuidado de transportar os artefatos nos quais está trabalhando da máquina anterior, para a máquina atual;
- **para poder compartilhar os artefatos por ele produzidos com outros desenvolvedores: p**ara que isso seja possível, é necessário oferecer

> acesso aos artefatos armazenados localmente a outros desenvolvedores. Essa tarefa pode ir desde a simples distribuição de cópias físicas dos arquivos, quanto ao compartilhamento de áreas do sistema de arquivo local, ou à publicação dos artefatos em servidores da rede. O fato é que se não existir uma forma padrão para o compartilhamento destes artefatos, cada desenvolvedor pode adotar sua própria abordagem tornando, praticamente impossível, o compartilhamento de forma controlada e segura das informações.

Portanto, quando uma ferramenta estiver conectada, através da rede, ao ambiente *DiSEN,* ela poderá utilizar os recursos, oferecidos pelo ambiente para fazer a persistência dos seus artefatos. Assim, será possível ao desenvolvedor de software fazer a persistência dos artefatos por ele produzidos, não apenas na máquina local, mas também no ambiente distribuído.

Com relação à persistência dos artefatos no ambiente distribuído, esta deve ocorrer de forma transparente ao desenvolvedor, no sentido em que este não precisa conhecer detalhes de como é realizado o processo de persistência. Detalhes como a localização do nó ou, nós da rede nos quais a persistência está sendo efetivamente realizada, não interessam ao desenvolvedor, usuário do suporte de persistência. A este basta saber que se necessário ele poderá utilizar o ambiente distribuído para armazenar e, posteriormente, recuperar os artefatos por ele produzidos. Isto está ilustrado na figura 7, no *use case* de persistência de artefatos.

A utilização de um repositório de metadados, conforme a especificação *MOF,* na implementação do suporte a persistência de artefatos do ambiente *DiSEN,* cria a possibilidade de armazenar não apenas um dado artefato, mas, também, armazenar os metadados que descrevem este artefato.

O presente trabalho, portanto, estende o trabalho apresentado em (PASCUTTI, 2002), no que se refere à utilização de um repositório de artefatos, no sentido em que propõe a adoção de um mecanismo de armazenamento de artefatos, para o ambiente *DiSEN*, que seja capaz de armazenar não apenas as informações contidas nos artefatos, mas também as informações que descrevem o formato destas informações.

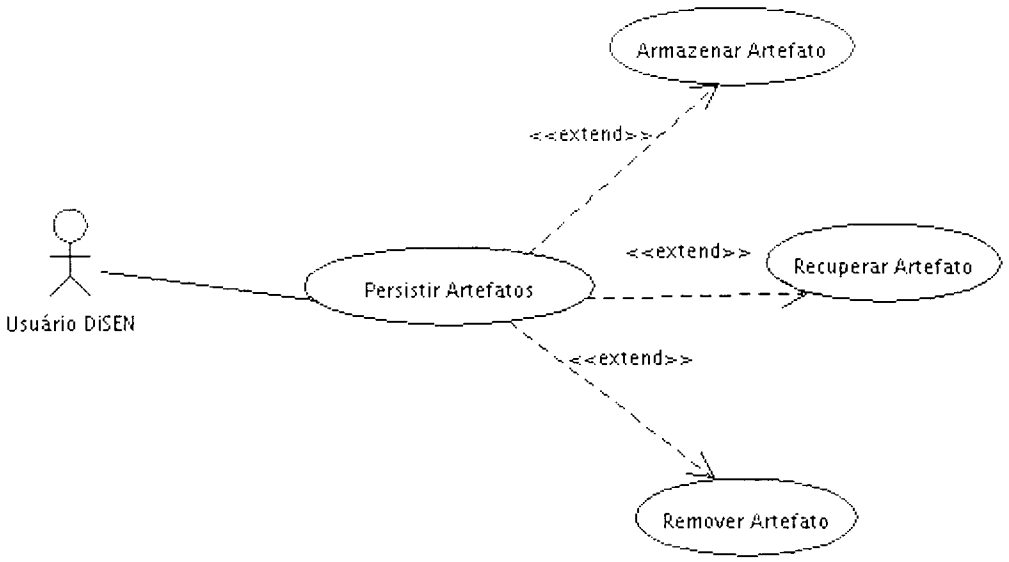

Figura 7 - *Use Case* persistir artefatos.

Uma vez que o *DiSEN* possua um repositório de metadados com informações sobre o formato dos artefatos, utilizados por cada uma das ferramentas presentes no ambiente, tais informações poderão ser utilizadas na construção de programas que façam tradução das informações contidas em um tipo de artefato, gerado por uma ferramenta X para outro tipo de artefato utilizado por uma ferramenta Y.

Outra possível utilização deste repositório, seria a construção de uma ferramenta de propósito geral, por exemplo, um navegador de artefatos, que fosse capaz de manipular as informações de determinado artefato com base nas informações contidas nos metamodelos, armazenados no repositório de metadados.

Como a proposta é disponibilizar tal repositório em um ambiente distribuído de desenvolvimento de software, deverão ser consideradas características como disponibilidade, recuperação de falhas, balanceamento de carga, etc.

A seguir, na seção 4.8, são apresentadas as características desejáveis ao suporte à persistência de artefatos, utilizando o repositório de metadados no ambiente distribuído de desenvolvimento de software *DiSEN*.

4.8 CARACTERÍSTICAS DESEJÁVEIS

Para que seja possível disponibilizar um repositório de metadados no ambiente de desenvolvimento distribuído de software *DiSEN*, é necessário que se considere uma série de fatores que dizem respeito à disponibilidade, escalabilidade, transparência de localização, recuperação de falhas, desempenho, atomicidade das operações e balanceamento de carga entre outros.

Abaixo são consideradas várias características desejáveis ao repositório de metadados, considerando-se os fatores acima citados e que dizem respeito ao seu funcionamento no ambiente *DiSEN:*

- **quanto às funcionalidades do repositório:** basicamente o repositório deverá permitir que um usuário (desenvolvedor de software) de uma ferramenta de desenvolvimento de software possa armazenar, recuperar e remover artefatos de um repositório;
- **quanto à compatibilidade e utilização de padrões:** deve permitir o armazenamento, manipulação e recuperação dos diferentes artefatos que serão produzidos por diferentes ferramentas de desenvolvimento de software. O repositório, então, deve adotar a utilização de padrões abertos que possibilitem diferentes ferramentas interagirem com ele;
- **quanto ao desempenho:** o repositório de metadados deve ser projetado, buscando otimizar o seu desempenho em um ambiente distribuído. Técnicas como utilização de um cache local pelo cliente e balanceamento de carga devem ser consideradas para se alcançar um melhor desempenho;
- **quanto à distribuição:** o repositório de metadados, dentro do ambiente de desenvolvimento distribuído de software, poderá estar localizado e em funcionamento em 1 ou mais nós da rede em um mesmo momento;
- **quanto à transparência:** deverá ser transparente aos clientes do repositório a localização dos repositórios de metadados, na rede, bem como o número deles em operação em um dado momento. Também deve ser transparente ao cliente se, em determinado momento, ele está utilizando o repositório que está no nó A ou B da rede. É necessário entretanto que o cliente sempre utilize informações consistentes e atualizadas;

- **quanto à sincronização dos repositórios:** havendo mais de um repositório de metadados presentes e em funcionamento no ambiente distribuído em um mesmo momento, estes deverão ter o seu conteúdo sincronizado e atualizado com relação as operações de armazenamento, solicitadas pelos clientes do repositório. Um cliente deverá sempre utilizar um repositório que contenha informações atualizadas. Assim é necessário um mecanismo que mantenha as informações entre diferentes repositórios sincronizadas e atualizadas;
- **quanto ao balanceamento de carga:** havendo mais de um repositório de metadados presente e em funcionamento no ambiente distribuído em um mesmo momento, as operações solicitadas pelos clientes do repositório deverão ser processadas de forma balanceada por estes repositórios;
- **quanto à disponibilidade do repositório e à detecção e recuperação de falhas:** a capacidade de armazenar e recuperar artefatos deve permanecer disponível mesmo com a ocorrência de eventuais falhas em um nó ou em um segmento da rede. Para tanto, havendo mais de um repositório na rede, se um deles se tornar indisponível outros deverão continuar executando as operações de forma transparente ao cliente. Deverá ainda ser possível detectar falhas na operação de um repositório de metadados e disparar, quando for o caso, as medidas necessárias para que o mesmo volte a operar ou, passe a operar em outro nó da rede. No momento em que um repositório entrar ou, voltar, à operação no ambiente distribuído, este deverá ter primeiro seu conteúdo atualizado para só depois poder atender às solicitações de clientes;
- **quanto à escalabilidade:** à medida em que a demanda por operações de armazenamento de artefatos no repositório de metadados crescer, deverá ser possível aumentar o número de repositórios de metadados no ambiente distribuído para atender a esta demanda;
- **quanto ao acesso aos dados:** o repositório deve permitir um acesso controlado aos elementos nele armazenados, tomando como base, o autor/criador das informações nele armazenadas e os direitos de acesso que ele atribui à sua criação. Por exemplo, a partir do momento que o

usuário X armazena um conjunto de informações, elas deverão poder ser acessadas ou alteradas por outros usuários apenas se o usuário X, ou outros usuários com permissão para alterar os direitos de acesso deste conjunto de informações, assim o permitirem;

- **quanto ao controle de versões:** o repositório deve permitir que sejam armazenadas diferentes versões de um mesmo elemento;
- **quanto à atomicidade de operações:** as operações de arrmazenamento, solicitadas pelos clientes do repositório, deverão ser atômicas no sentido em que, ou uma operação acontece com sucesso no ambiente distribuído ou ela não acontece. Operações de armazenamento, que sejam compostas por um conjunto de operações menores, devem ser todas elas executadas com sucesso, ou nenhuma delas deve ser executada.

4.9 CONSIDERAÇÕES FINAIS

Basicamente as características apresentadas acima serviram de base para a elaboração do presente trabalho. A seguir é apresentada a descrição de sua execução, bem como as abordagens adotadas para se alcançarem as características aqui apresentadas.

5 REPOSITÓRIO DE METADADOS PARA O AMBIENTE DiSEN

5.1 INTRODUÇÃO

Para a implementação do suporte a persistência de artefatos para o *DiSEN*, foi considerada a escolha de tecnologias e produtos que pudessem ser utilizados para implementar uma solução que atendesse às características desejáveis relacionadas na seção 4.8.

Uma decisão tomada com relação à implementação deste trabalho foi a de procurar utilizar produtos já existentes que pudessem ser integrados e incorporados à solução, ao invés de implementar-se toda a solução a partir do zero.

Outra decisão, com relação a escolha das tecnologias a serem utilizadas neste trabalho, foi a de dar preferência às tecnologias e produtos que sejam de código aberto, desde que isso não signifique prejuízo ao mesmo. Um dos motivos da segunda decisão é, justamente, a vantagem de, utilizando um produto de código aberto, poder ter acesso ao código fonte do mesmo, inclusive para implementar alterações.

Devido à complexidade que envolve a implementação de todas as características relacionadas na seção 4.8, foi estabelecida uma ordem de prioridade determinando as características que este trabalho poderia abordar em um primeiro momento e aquelas que serão abordadas em trabalhos futuros.

Das características constantes na seção 4.8, os itens abaixo não serão abordados neste trabalho e serão objeto de estudo em trabalhos futuros.

- acesso aos dados;
- controle de versões;
- desempenho.

Não foram implementados, por exemplo, recursos como *caches* locais para os clientes do repositório de metadados ou qualquer outra otimização com relação ao desempenho da utilização do suporte à persistência de artefatos, oferecido pelo ambiente *DiSEN*.

Em um primeiro momento, o protótipo desenvolvido para o presente trabalho pretende validar a implementação de estratégias, relacionadas às demais características apresentadas na seção 4.8.

A seguir são relacionadas as tecnologias e produtos que foram escolhidos, bem como a justificativa para cada um deles. Na seção 5.3 são relacionadas as características apresentadas anteriormente na seção 4.8, juntamente com a respectiva abordagem adotada para cada uma delas no presente trabalho.

5.2 TECNOLOGIAS E PRODUTOS UTILIZADOS

Inicialmente optou-se por utilizar o repositório de metadados *MDR* (SUN, 2002b) apresentado na seção 4.5.

Apesar dos recursos oferecidos pelo repositório *MDR*, ele não possui suporte para ser utilizado através de uma rede de computadores e, também, não oferece suporte para ser utilizado por múltiplos usuários ao mesmo tempo. Assim, para que seja possível utilizar este repositório em um ambiente distribuído de desenvolvimento de software, de forma concorrente por diversos clientes, é necessário a implementação de uma camada de software que integre o repositório ao ambiente.

Para a implementação desta camada de integração entre o repositório *MDR* e um ambiente de desenvolvimento distribuído de software foram escolhidas a tecnologia *Jini* (SUN, 2001) e o *framework* desenvolvido pelo projeto *Rio* (RIO PROJECT, 2001b).

Por estar disponível de forma gratuita à desenvolvedores, por possuir uma ampla base de desenvolvedores e documentação disponível (tutoriais, listas de discussão) e, finalmente, por basear-se na linguagem *Java* que tem sido aceita tanto no ambiente corporativo quanto no acadêmico, *Jini* foi escolhido como *middleware.*

Além disso, pesaram na escolha as características dinâmicas da tecnologia *Jini*, o seu serviço de *JavaSpaces* (SUN, 2002a), bem como, o suporte que o *framework* fornecido pelo projeto *Rio* (RIO PROJECT, 2001b) oferece no que diz

respeito à definição de serviços baseados em qualidade de serviço (*QoS*). Estes recursos oferecem uma base para o desenvolvimento de uma solução que seja capaz de detectar e recuperar eventuais falhas, indo ao encontro das características relativas à disponibilidade, detecção e recuperação de falhas listadas na seção 4.8.

Outro fator determinante da escolha de *Jini* e *Rio* foi a relação de semelhança existente entre alguns dos elementos constituintes da arquitetura descrita em (PASCUTTI, 2002) e os recursos oferecidos por estas tecnologias.

Por exemplo, os papéis do "supervisor de configuração dinâmica" definido em (PASCUTTI, 2002), se assemelha muito ao *Provision Monitor* presente no *framework* do projeto *Rio.* Enquanto que o "servidor de página amarela", também presente na arquitetura do *DiSEN,* se assemelha ao serviço de *Lookup* do *Jini.*

A utilização do repositório de metadados *MDR* adiciona à solução conformidade com as especificações *MOF* e *XMI.* É ele que fornece o suporte necessário ao armazenamento de artefatos, através do armazenamento de seus metadados e metamodelos correspondentes. Ele, também, proporciona recursos de leitura e gravação dos artefatos em arquivos *XMI* (na verdade seus metadados e metamodelos). Uma ferramenta de desenvolvimento de software, por exemplo, deverá ter o suporte necessário para conseguir, depois de gravar seus artefatos em arquivos *XMI*, enviá-los para que sejam armazenados no repositório de metadados.

Rio e *Jini* por sua vez são utilizados para disponibilizar, em um ambiente distribuído, o acesso ao repositório de metadados *MDR,* adicionando, à solução, características dinâmicas e suporte à *QoS,* possibilitando, por exemplo, a detecção e recuperação de falhas.

A seguir é apresentada a forma adotada para disponibilizar o repositório de metadados *MDR* em um ambiente distribuído de desenvolvimento de software *DiSEN,* utilizando-se *Jini*, *JavaSpaces* e *Rio.*

5.3 FUNCIONALIDADES DO SUPORTE À PERSISTÊNCIA

A tabela 3 relaciona as características desejáveis ao repositório de artefatos para o *DiSEN* que foram, inicialmente, apresentadas na seção 4.8 e a abordagem adotada para cada uma delas no presente trabalho.

Tabela 3 - Características X abordagem adotada

Características Desejáveis	Abordagem Adotada
Quanto à compatibilidade e utilização de padrões	O repositório *MDR* adotado oferece compatibilidade com os padrões abertos *MOF*, e *XMI* especificados pela *OMG* além de *JMI* do *JCP*.
Quanto ao desempenho	Em trabalhos futuros, a implementação do protótipo deverá ser revisada buscando-se minimizar a utilização da banda e as chamadas através da rede. Ao mesmo tempo um *cache* local deverá ser considerado para ser incorporado aos clientes.
Quanto à distribuição	Foi desenvolvida uma camada de integração do repositório de metadados com o ambiente distribuído utilizando-se a tecnologia *Jini* e o *framework* oferecido pelo projeto *Rio*.
Quanto à transparência	Para o cliente, tanto ferramenta quanto usuário, é transparente a localização do repositório de metadados e, também, o número de repositórios que estão em operação no ambiente em um dado momento.
Quanto à sincronização dos repositórios	Foram implementadas duas abordagens de atualização do conteúdo dos repositórios de metadados (apresentadas na seção 5.4.2). Estas atualizações são disparadas pelos próprios repositórios de forma transparente para o cliente dos mesmos.
Quanto ao balanceamento de carga	O balanceamento de carga é alcançado com a utilização do espaço compartilhado *JavaSpaces* juntamente com uma estrutura de dados distribuída e um protocolo distribuído (apresentados nas seções 5.4.1 e 5.4.2 respectivamente). Diferentes repositórios remotos podem executar tarefas, paralelamente, enviando o resultado de volta para o espaço assim que terminarem o processamento.
Quanto à disponibilidade do repositório e à detecção e recuperação de falhas	O *framework* oferecido pelo projeto *Rio*, fornece suporte à detecção e recuperação de falhas, no sentido em que possibilita que serviços em uma rede distribuída sejam monitorados (pelo *Provision Monitor*). Caso um dos serviços apresente falhas e fique indisponível, o *Provision Monitor* se encarrega de procurar e iniciar o serviço em outro nó da rede. Além disso é possível ter diversos repositórios remotos em operação em um mesmo momento trabalhando de forma concorrente com relação ao processamento de tarefas e de forma sincronizada com relação ao seu conteúdo.
Quanto à escalabilidade	A implementação adotada é escalável no sentido em que a qualquer momento pode ser aumentado o número de repositórios remotos no ambiente distribuído. O aumento ou diminuição de repositórios no ambiente distribuído é transparente para os clientes do mesmo.
Quanto à atomicidade de operações	Para garantir a atomicidade das operações executadas pelo repositório de metadados no ambiente distribuído, foi utilizado o recurso de transações distribuídas existente na tecnologia *Jini*. As transações distribuídas foram utilizadas nas operações que envolvem alteração da estrutura de dados distribuída armazenada no espaço compartilhado *JavaSpaces*.
Quanto ao acesso aos dados	Em trabalhos futuros, deverá ser adicionado o controle de acesso às informações armazenadas no repositório de artefatos.
Quanto ao controle de versões	Em trabalhos futuros, deverá ser adicionado o controle de versões no ambiente distribuído *DiSEN*.

Conforme a figura 6 apresentada na seção 4.7, um cliente que vá utilizar o repositório de metadados para a persistência de seus artefatos, precisará pelo menos poder armazenar e recuperar artefatos no repositório. O cliente deve poder também remover um artefato do repositório de metadados.

MDR é um repositório baseado na especificação *MOF,* assim, o seu funcionamento está relacionado com a arquitetura conceitual desta especificação, descrita na seção 4.2.

De uma forma geral, neste funcionamento, para que seja possível armazenar qualquer informação no repositório de metadados, é necessário que antes se armazene um metamodelo que descreva estas informações.

A *UML*, por exemplo, possui uma descrição baseada em *MOF*, assim, é possível dentro do *MDR* armazenar o modelo de metadados da *UML*. Este modelo não deve ser confundido com os artefatos em *UML* que são criados durante o desenvolvimento de um software. Esse modelo de metadados da *UML* descreve os elementos que compõem a *UML* e que podem ser usados na elaboração de um artefato *UML.*

A figura 8 dispõe o modelo *MOF*, o modelo *UML* e os artefatos *UML,* dentro da arquitetura conceitual do *MOF.*

Na figura 8, na camada M3 (ou camada 3 de metadados) está o modelo *MOF* que pode ser usado para descrever qualquer modelo de metadados, orientado a objetos. Já na camada M2 tem-se o metamodelo *UML*, que contém a descrição da especificação *UML.* Finalmente, na camada M1, tem-se os artefatos *UML.*

Como é possível observar, a camada de metadados N tem a descrição dos elementos da camada N-1, ou seja, os elementos da camada N-1 são descritos com base nos elementos presentes na camada N. Os artefatos *UML,* da camada M1 são representados, utilizando-se a notação do modelo *UML*, cuja descrição está na camada M2, enquanto que a especificação *UML* é descrita utilizando-se a notação do modelo *MOF* que por sua vez está descrito na camada M3.

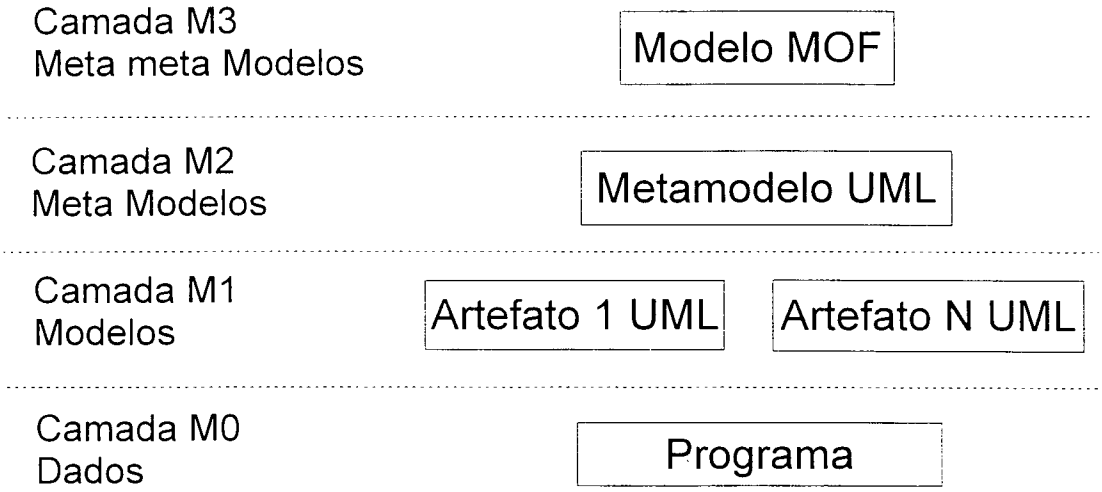

Figura 8 - Modelo *MOF*, *UML* e artefatos *UML*

Assim, para que seja possível armazenar um artefato *UML* no repositório de metadados, é necessário que tenha sido carregado nele, anteriormente, o metamodelo que descreve os elementos da *UML*. Uma vez carregado este metamodelo com a descrição da *UML*, é possível criar uma espécie de sub-repositório para modelos descritos por este metamodelo, no caso artefatos *UML*.

O metamodelo que descreve a *UML* pode ser carregado no repositório a qualquer momento, pois este estará descrito em *MOF* e por definição um repositório de metadados, baseado em *MOF,* deve ter o suporte ao modelo *MOF*. Na verdade, quando inicializado, uma das primeiras coisas que o repositório *MDR* faz, é carregar a descrição do modelo *MOF.*

A figura 9 ilustra alguns dos *use case* que um usuário do *MDR* pode executar.

Já a figura 10 apresenta uma combinação dos diagramas da figura 6 e da figura 9, mostrando os *use cases* necessários ao suporte à persistência de artefatos para o ambiente *DiSEN,* utilizando o repositório *MDR*.

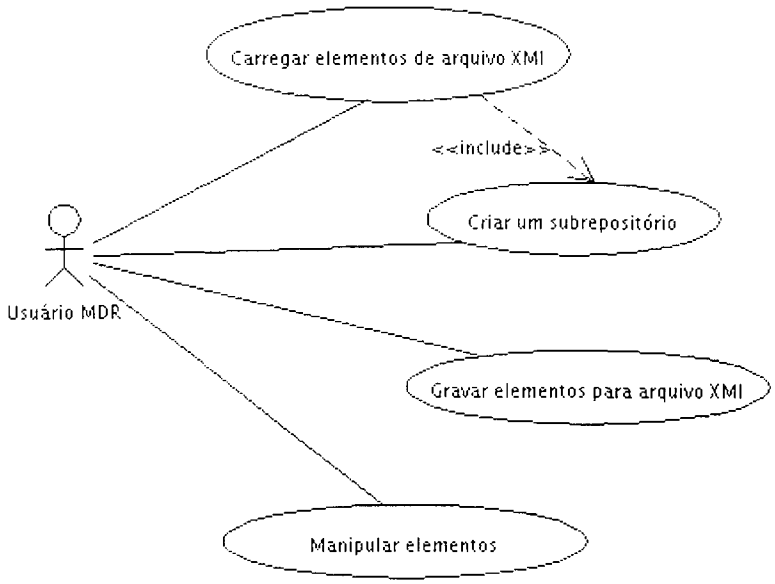

Figura 9 - Alguns *use case* que um usuário do *MDR* pode executar

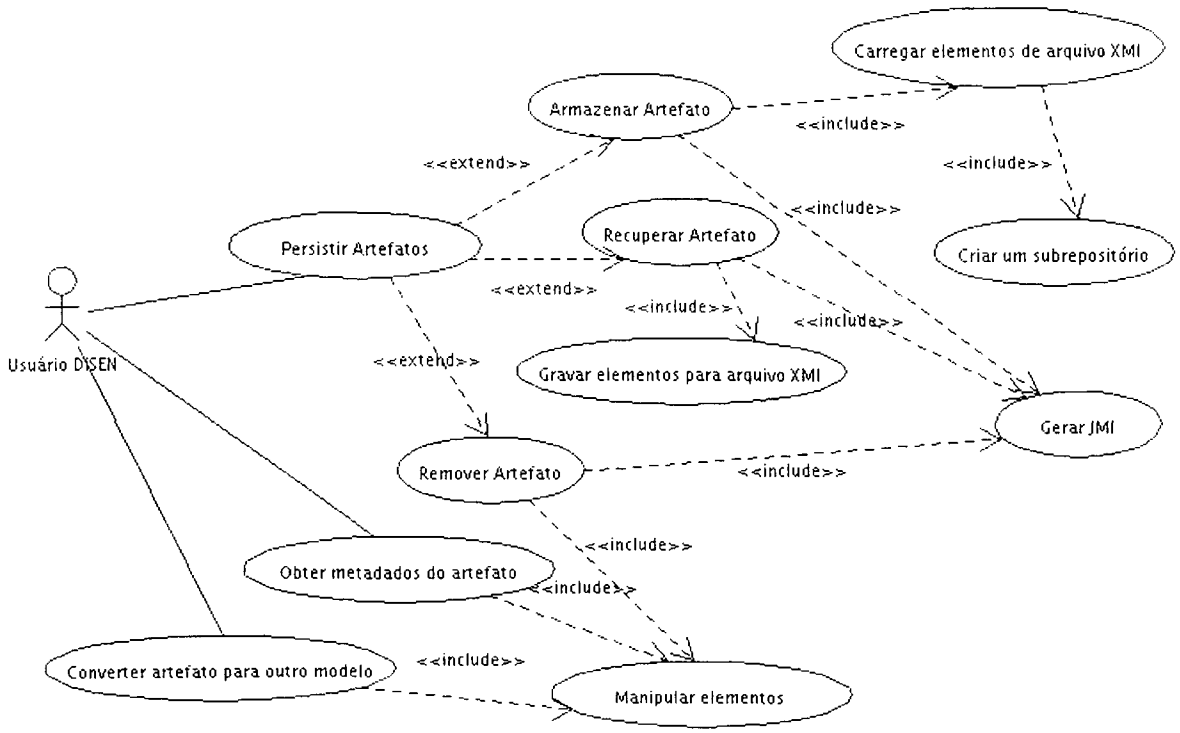

Figura 10 - *Use cases* da persistência de artefatos do *DiSEN* e do MDR

Por exemplo, o carregamento do modelo *UML* para o repositório é

executado através da leitura de um arquivo *XMI* que contém a descrição em *MOF* dos elementos deste modelo e a verificação se a descrição destes elementos está correta. Esta verificação só é possível pelo fato do repositório já conter o modelo *MOF*, armazenado anteriormente, utilizado para descrever o modelo *UML*.

A figura 11 apresenta um diagrama de seqüência para o *use-case* carregar elementos de arquivo *XMI*.

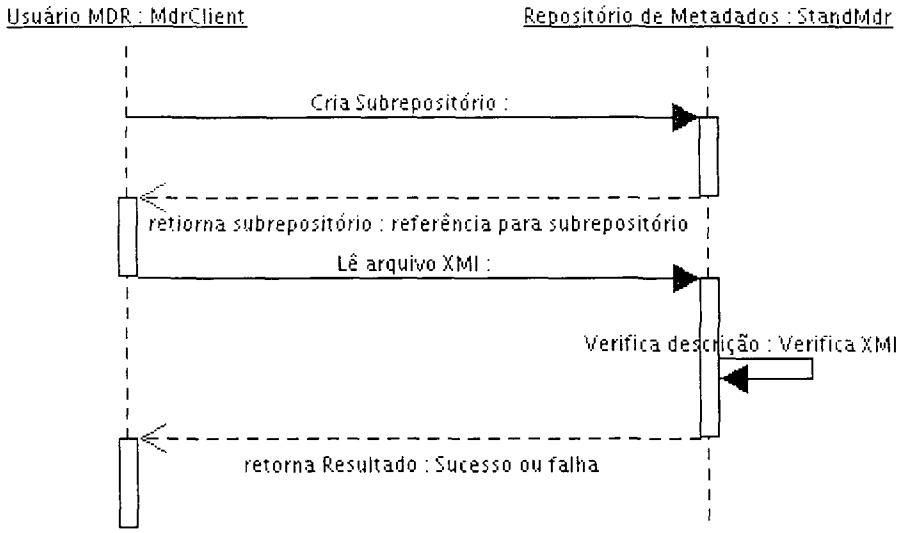

Figura 11 - Carregar elementos de arquivo *XMI*

O *MDR* pode tanto ler um arquivo *XMI* e armazenar os elementos nele descrito no repositório, como, ao contrário, escrever um arquivo *XMI*, contendo a descrição de elementos que estão armazenados no repositório.

A criação de um sub-repositório acontece por meio da criação de uma instância de um elemento específico de um modelo já armazenado no repositório. Assim para criar um sub-repositório para armazenar um modelo descrito em *MOF*, é necessário instanciar um elemento do modelo *MOF*, já existente no repositório.

Da mesma forma, para criar um sub-repositório para armazenar artefatos *UML* é necessário instanciar um elemento específico do modelo *UML*, que já deverá estar armazenado no repositório. A figura 12 apresenta um diagrama de seqüência, mostrando os passos necessários para que seja possível armazenar um artefato de um modelo, por exemplo *UML*, no *MDR*.

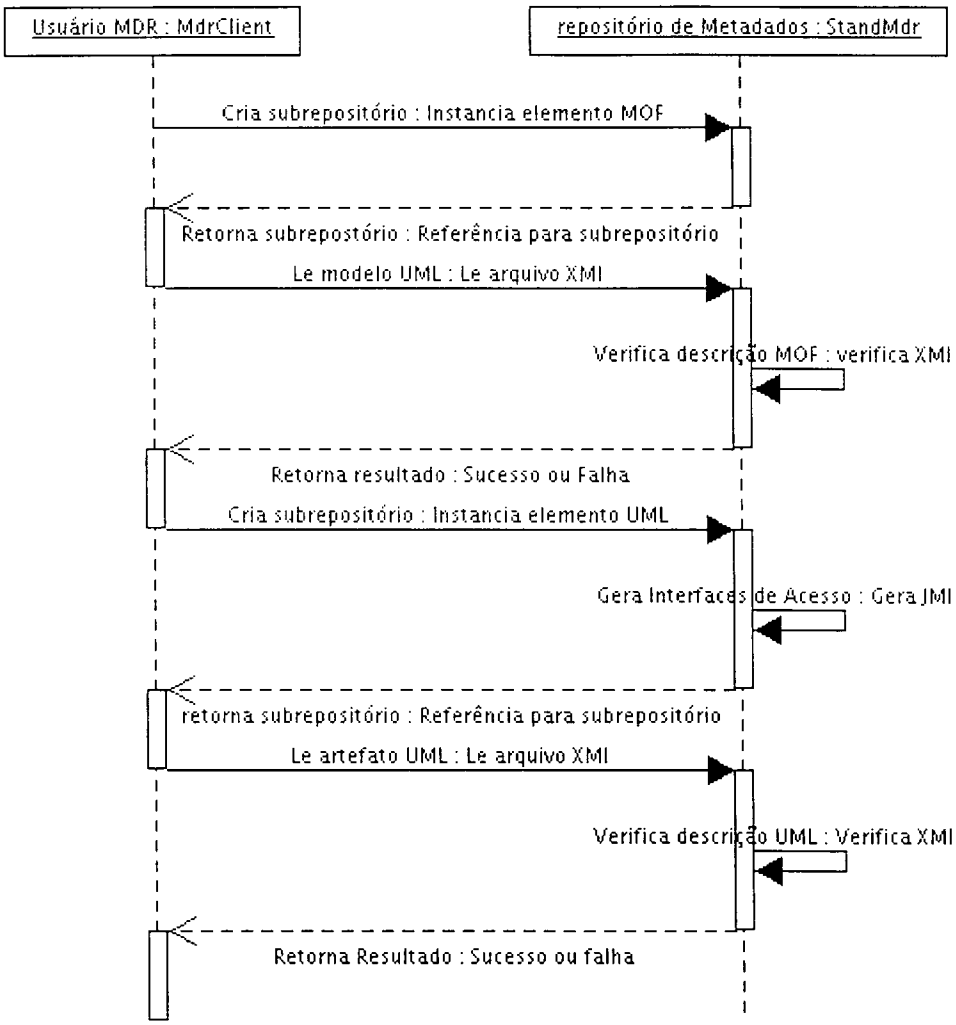

Figura 12 - Armazenar artefato

Finalmente, é importante saber que para que o *MDR* possa acessar os elementos armazenados em um sub-repositório ele precisa das interfaces *JMI,* correspondentes ao modelo que teve um elemento instanciado para a criação do sub-repositório.

O *MDR* é capaz de gerar estas interfaces *JMI* automaticamente e para que se possa manipular qualquer elemento armazenado em um sub-repositório, é necessário primeiro gerar estas interfaces. O mesmo vale para a criação de uma instância de um determinado elemento de um sub-repositório.

Esta geração automática das interfaces *JMI* acontece sempre que se tentar

acessar pela primeira vez em uma execução do *MDR,* um determinado sub-repositório.

Assim, apesar do praticidade da geração automática destas interfaces, uma boa abordagem é gerar as interfaces *JMI,* através do próprio *MDR* e incluí-las no caminho de bibliotecas da máquina virtual *Java* que executará o repositório. Neste caso, quando alguém for tentar acessar o sub-repositório, as interfaces *JMI* já estarão disponíveis para o *MDR*, não sendo necessário gastar tempo gerando-as. A figura 13 mostra um diagrama de seqüência da primeira tentativa de manipulação de um elemento de um modelo de um sub-repositório.

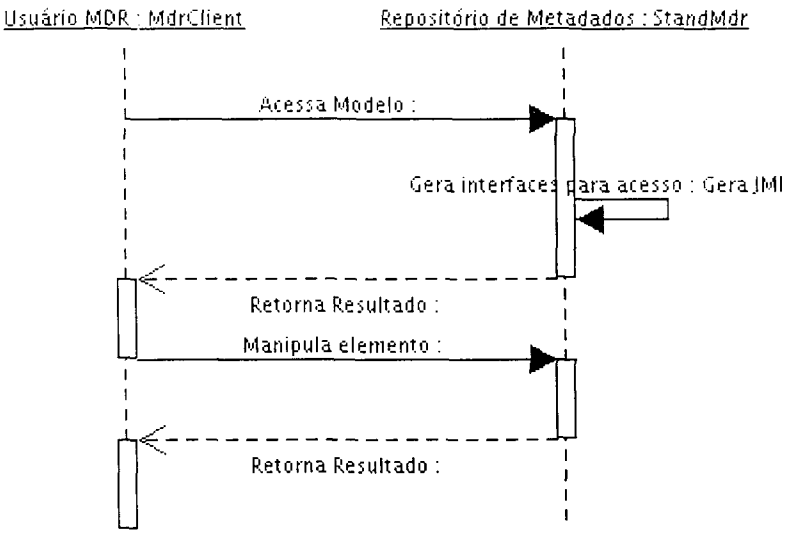

Figura 13 - Gerar *JMI*

Até o presente momento foi detalhado o exemplo do armazenamento de artefatos *UML* no repositório de metadados. Entretanto, é essencial que fique claro que o repositório de metadados pode armazenar artefatos de qualquer outro modelo, desde que esse modelo tenha uma descrição *MOF.*

A *OMG* disponibiliza uma descrição da *UML* em *MOF.* Para armazenar um outro tipo de artefato baseado, por exemplo, em um modelo estendido da *UML*, seria necessário modelar esse modelo estendido em *MOF* e utilizá-lo no repositório da mesma forma que foi descrita a utilização do modelo *UML.*

Em (SUN, 2003c) existem alguns modelo descritos em *MOF*, prontos para

serem carregados no *MDR* (*UML* 1.3, *UML* 1.4, *UML* 1.5, etc.). Além disso, é possível utilizar ferramentas de modelagem como o *Rational Rose* (RATIONAL, 2002), ou mesmo o *Poseidon for UML* (GENTLEWARE, 2003a) juntamente com a ferramenta *UML2MOF* (MATULA, 2003) para se criar um novo modelo em *MOF.*

Finalmente, o repositório *MDR* não possui suporte à utilização através da rede e nem à múltiplos usuários, assim, na próxima seção é apresentada a abordagem adotada para integrar o repositório de metadados *MDR* ao ambiente *DiSEN* e disponibilizar as funcionalidades necessárias ao suporte à persistência de artefatos, através da rede em um ambiente distribuído.

5.4 DISPONIBILIZAÇÃO DO REPOSITÓRIO *MDR* NO *DiSEN*

Para que um cliente do ambiente distribuído de desenvolvimento de software *DiSEN* possa utilizar o repositório de metadados *MDR*, este precisa de alguma maneira poder acessar as funcionalidades do mesmo através da rede.

Como o repositório *MDR* não tem suporte à utilização através da rede, foi criada uma camada de integração do repositório com o ambiente distribuído. Os elementos de software desta camada de integração tem como papel receber requisições de execução de tarefas de clientes em atividade na rede, fazer a chamada necessária ao repositório de metadados para que ele execute a tarefa requisitada e, finalmente, retornar o resultado ao cliente.

Para utilizar as capacidades do repositório de metadados, um cliente deve enviar uma requisição de execução de tarefa através da rede, e, então, esperar pela resposta. Esta requisição será recebida e processada por um repositório de metadados através dos objetos da camada de integração com o ambiente distribuído.

A partir deste ponto, para o restante do trabalho, será utilizada a expressão repositório remoto de metadados *RemoteMDR* (*Remote Metadata Repository*) para referenciar o repositório de metadados *MDR*, juntamente, com os elementos de software da camada de integração, que permitem que o *MDR* receba requisições de execução de tarefas e envie os resultados para a rede.

A disponibilização do repositório de metadados no ambiente distribuído foi implementada de forma que possam haver em um mesmo momento diversos clientes e, também, diversos repositórios remotos de metadados (*RemoteMDR*) espalhados em nós distintos da rede, trabalhando de forma conjunta. Para tanto, *JavaSpaces* é utilizado como mediador entre repositórios remotos de metadados que estão distribuídos em nós da rede e clientes que queiram utilizar as suas capacidades, conforme ilustrado na figura 14.

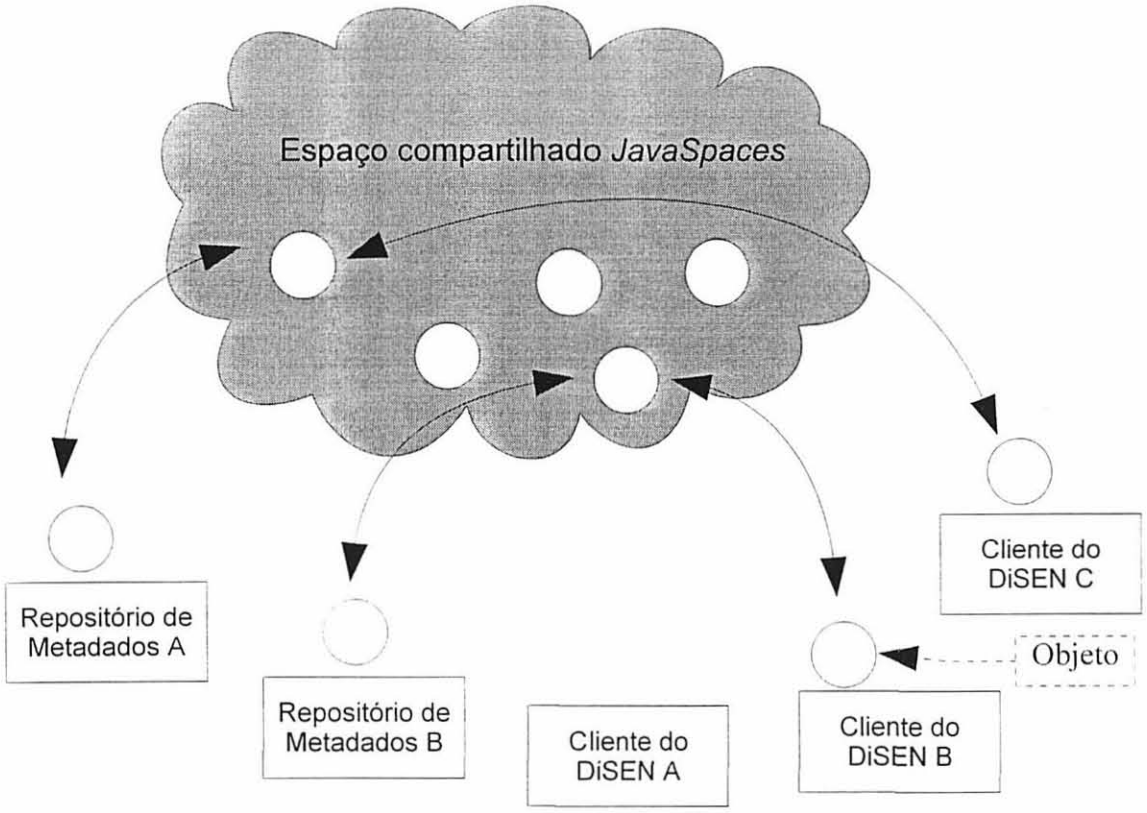

Figura 14 - Comunicação entre repositórios de metadados e clientes

Na verdade, toda comunicação entre os clientes e os repositórios remotos de metadados e, também, entre diferentes repositórios de metadados, ocorre através da troca de objetos, utilizando o espaço compartilhado na rede *JavaSpaces*.

Segundo (FREEMAN; HUPFER; ARNOLD, 1999), para criar aplicações baseadas em espaços compartilhados como o *JavaSpaces*, é necessária a criação

de estruturas de dados distribuídas e protocolos distribuídos que utilizam esta estrutura.

Na implementação do presente trabalho, então, foi criada uma estrutura de dados distribuída, apresentada a seguir e um protocolo distribuído descrito nas seções 5.4.1 e 5.4.2 respectivamente.

5.4.1 Estrutura de Dados Distribuída

JavaSpaces é utilizado neste trabalho, de tal forma, que as requisições de tarefas que são enviadas para ele são lidas pelos repositórios remotos de metadados na mesma ordem em que foram enviadas. É a estrutura de dados distribuída, que fica armazenada no espaço compartilhado de troca *JavaSpaces* na rede, juntamente com o protocolo distribuído, que determina a forma de acesso controlado aos elementos dessa estrutura, que possibilitam esse funcionamento.

Em (FREEMAN; HUPFER; ARNOLD, 1999), é apresentado o conceito de uma estrutura de dados distribuída chamada *channel,* que funciona como uma fila *FIFO.*

> Think of a channel as an information conduit that takes a series of messages at one end and delivers them at the other end, while maintaining their ordering. (...) The channel is capable of exhibiting many different behaviors, depending on the protocols that the senders and receivers of the channel obey.(FREEMAN; HUPFER; ARNOLD, 1999, p. 116)

Esta estrutura de dados distribuída é bastante versátil e pode ser usada como base para construção de diferentes *patterns* de comunicação (FREEMAN; HUPFER; ARNOLD, 1999). Para o presente trabalho, por exemplo, foi criada uma variação desta estrutura de dados distribuída.

A estrutura de dados distribuída, criada, funciona basicamente como uma fila *FIFO* (*First In First Out*) composta por elementos que servem de apontadores para o início da fila (*top*) e o seu final (*tail*).

A diferença da estrutura de dados distribuída *channel*, apresentada em (FREEMAN; HUPFER; ARNOLD, 1999) para a utilizada neste trabalho, consiste na implementação de um terceiro apontador para os elementos da fila. Assim, além dos

apontadores, que foram chamados de *top* e *tail*, a estrutura conta ainda com outro apontador, que foi chamado de *head* e que indica a primeira requisição de tarefa ainda não processada.

Essa estrutura de dados distribuída permite que clientes incluam tarefas no espaço compartilhado no final da fila, utilizando o apontador *tail*, ao mesmo tempo que repositórios remotos de metadados podem ler tanto, a primeira requisição de tarefa existente na fila, utilizando o apontador *top*, quanto a primeira requisição de tarefa ainda não processada, utilizando o apontador *head.*

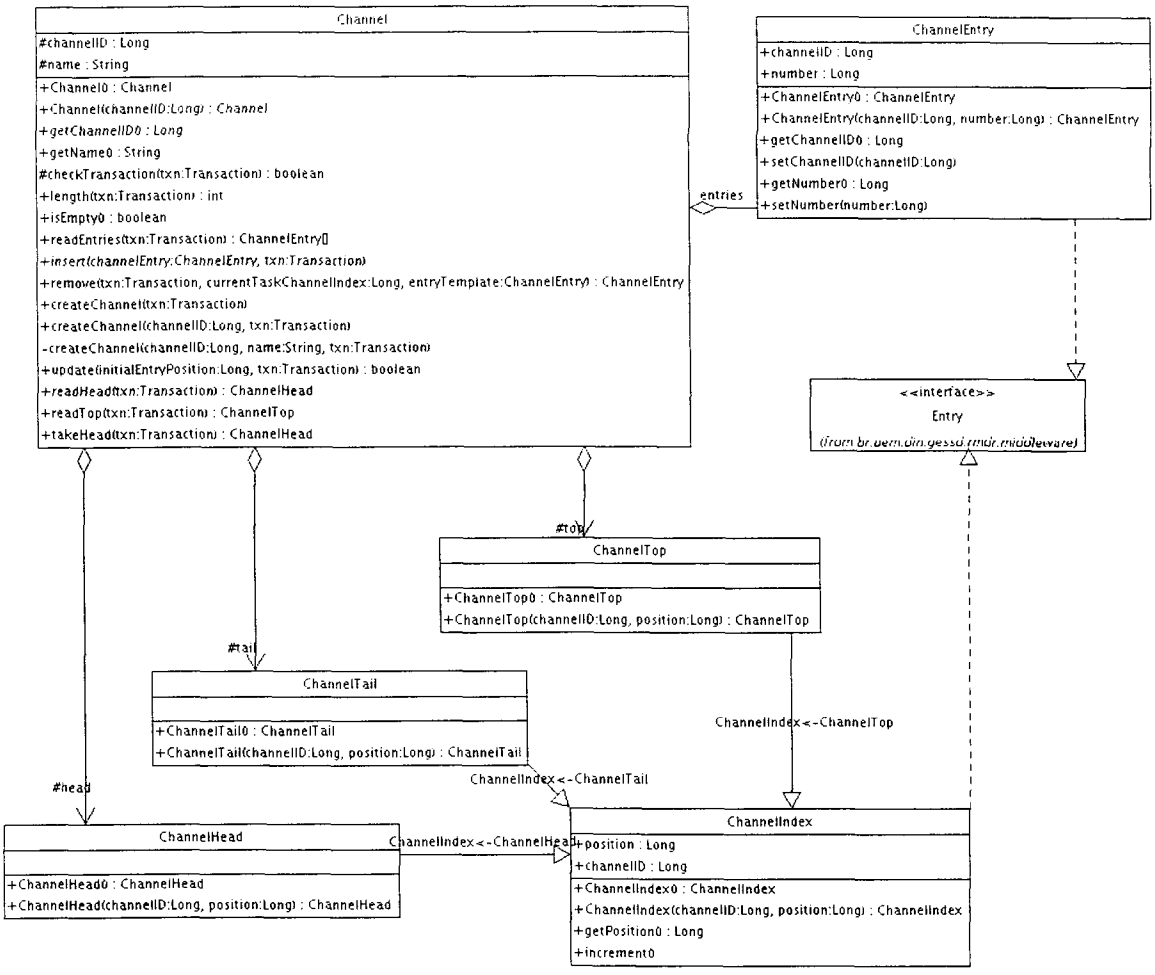

Figura 15 - Estrutura de dados distribuída

A estrutura de dados distribuída criada para o presente trabalho, também,

foi chamada de *channel* uma vez que consiste em uma pequena variação das estruturas apresentadas em (FREEMAN; HUPFER; ARNOLD, 1999) e é ilustrada na figura 15.

Quando a estrutura distribuída é criada pela primeira vez no espaço compartilhado, são criados os três apontadores *tail, top* e *head* com valores que indicam que não existem tarefas a serem processadas.

À medida que requisições de tarefas são enviadas para o espaço compartilhado *JavaSpaces*, o apontador de final da fila *tail* é incrementado. Ao mesmo tempo, à medida que requisições de tarefas são lidas e processadas por repositórios remotos de metadados, o apontador *head* que indica a primeira requisição de tarefa ainda não processada é incrementado. Finalmente, quando requisições de tarefas já processadas são retiradas da fila e do espaço compartilhado *JavaSpaces,* o apontador *top* é incrementado.

Assim, cada requisição de tarefa enviada ao espaço compartilhado recebe um número de seqüência que indica sua posição na fila e os repositórios remotos de metadados lêem e processam estas requisições de acordo com a ordem numérica desta seqüência. Desta forma, um repositório remoto só pode processar a requisição de uma tarefa que tenha como numeração de seqüência N, depois de ter processado a requisição da tarefa com numeração N-1.

A figura 16 representa a comunicação entre repositórios de metadados e clientes utilizando a estrutura de dados distribuída, armazenada no espaço compartilhado *JavaSpaces*.

Quando um cliente precisa utilizar uma das funcionalidades oferecidas pelo repositório de metadados, este envia para o espaço compartilhado na rede, *JavaSpaces*, um objeto que é uma requisição de execução de uma tarefa.

Para o presente trabalho foram criados diferentes tipos de objetos, cada qual representando um tipo de tarefa relacionada a uma ou mais funcionalidades do repositório de metadados *MDR*. De forma semelhante ao que foi feito com as tarefas, foram criados diferentes tipos de objetos para representar tipos de resultados diferentes que podem ser enviados como resposta por um repositório de metadados remoto.

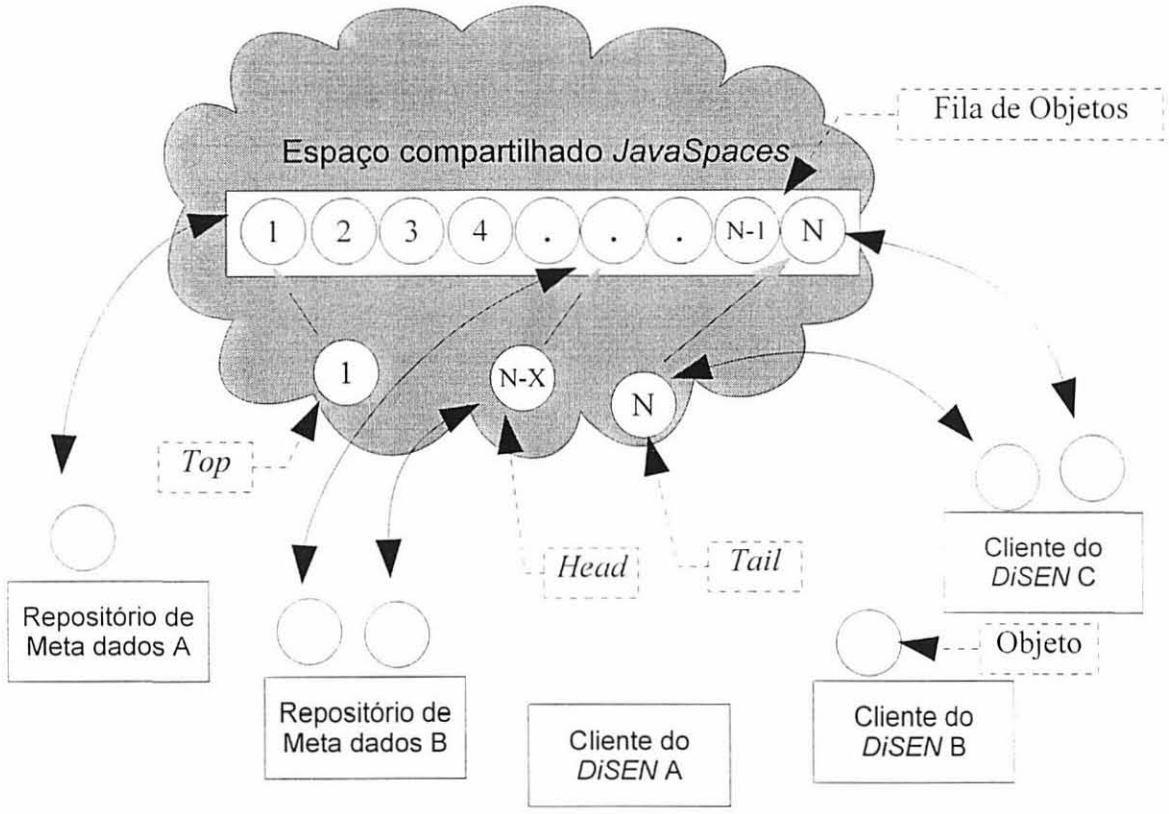

Figura 16 - Comunicação utilizando a estrutura de dados distribuída

O diagrama de classes da figura 17 apresenta uma estrutura de dados distribuída chamada *MDRChannel*, que herda as características da estrutura *Channel* apresentada no diagrama da figura 15 e adiciona as classes referentes aos objetos criados para representar as tarefas e resultados, os quais podem ser trocados através do *JavaSpaces* no processo de comunicação entre repositórios remotos de metadados e seus clientes.

A estrutura *MDRChannel*, então, é uma especialização da estrutura *Channel* que contém elementos voltados, especificamente, para a comunicação entre repositórios de metadados remotos e seus clientes através do ambiente distribuído *DiSEN*.

Enquanto que na estrutura distribuída *Channel* podem ser incluídos objetos genéricos, a estrutura *MDRChannel* pode receber apenas especializações de *MDRTaskEntry,* ou seja, objetos que sejam tarefas para repositórios de metadados remotos.

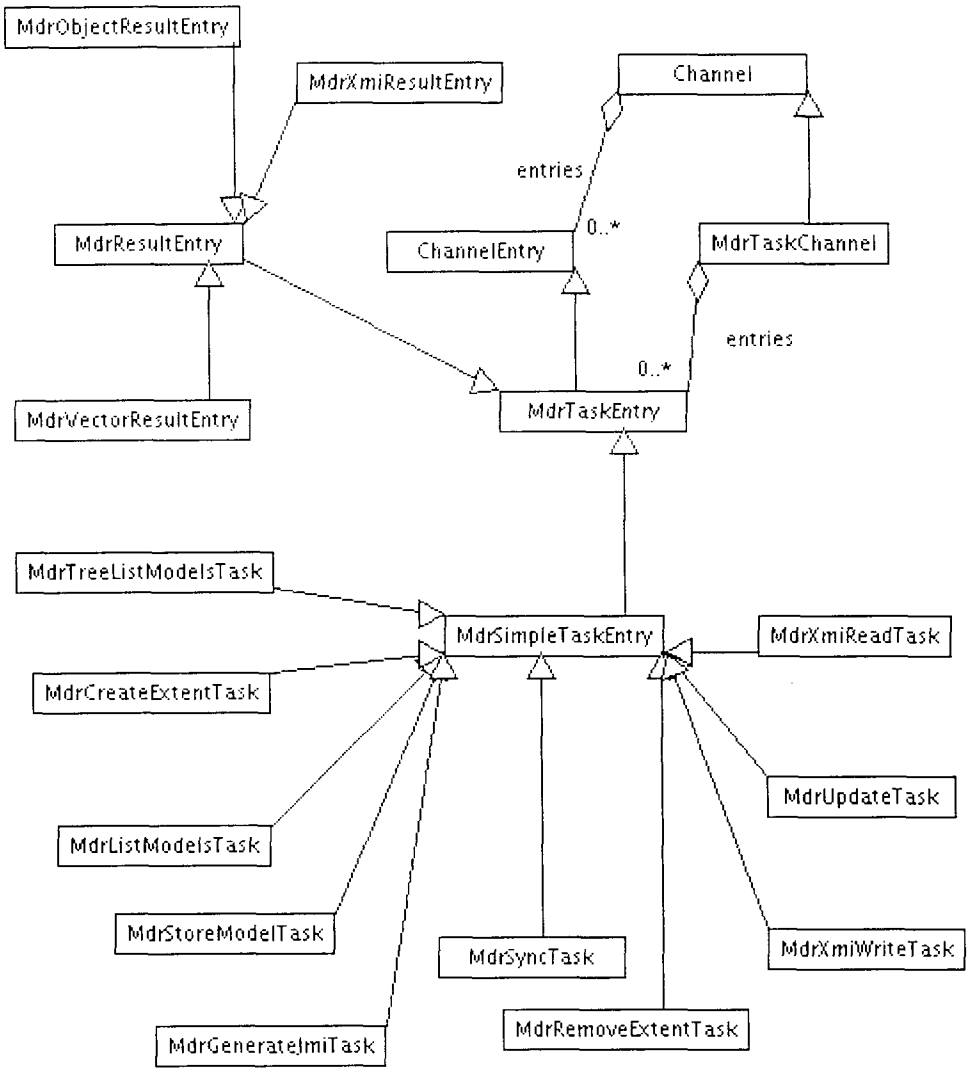

Figura 17 - Estrutura de dados distribuída com tarefas e resultados

A tabela 4, a seguir, apresenta as classes criadas para representar os diferentes tipos de tarefas que podem ser enviadas a um repositório de metadados remoto, enquanto a tabela 5 apresenta as classes que representam os resultados que podem ser enviados de volta pelos repositórios.

Tabela 4 - Tipos de tarefas e suas finalidades

Tipo de Tarefa	Finalidade
MdrTaskEntry	É usado para modelar os elementos comuns entre os tipos de tarefas e resultados relacionadas ao *RemoteMDR*.
MdrSimpleTaskEntry	É usado para modelar, somente, os elementos comuns aos tipos de tarefas relacionadas ao repositório de metadados remoto *RemoteMDR*.
MdrListModelsTask	Lista os modelos armazenados no repositório, retorna uma lista com os nomes dos modelos.
MdrTreeListModelsTask	Lista os modelos armazenados no repositório, retornando uma árvore com os modelos e seus elementos.
MdrCreateExtentTask	Cria um sub-repositório através da instanciação de um elemento de um modelo.
MdrGenerateJMITask	Gera as interfaces *JMI* de um modelo armazenado no repositório.
MdrXmiReadTask	Lê os elementos definidos em um arquivo *XMI* para dentro de um sub-repositório.
MdrXmiWriteTask	Grava em um arquivo *XMI* os elementos de um sub-repositório.
MdrRemoveExtentTask	Remove um sub-repositório.
MdrStoreModelTask	Armazena um modelo ou artefato dentro do repositório de metadados.
MdrUpdateTask	É usado pelo repositório remoto para indicar a necessidade de fazer a atualização do seu conteúdo através do processamento das tarefas que ainda estão presentes no espaço de compartilhado *JavaSpaces*..
MdrSyncTask	É usado pelo repositório remoto para fazer a atualização do seu conteúdo a partir do conteúdo de um outro repositório remoto atualizado e não através do processamento das tarefas que ainda estão presentes no espaço de compartilhado *JavaSpaces*..

Tabela 5 - Tipos de resultados e suas finalidades

Tipo de Resultado	Finalidade
MdrResultEntry	É usado para modelar os elementos comuns a todos os tipos de resultados relacionados as tarefas executadas pelos repositórios remotos.
MdrObjectResultEntry	É utilizado para retornar um objeto
MdrVectorResultEntry	É utilizado para retornar um vetor de objetos
MdrXmiResultEntry	É utilizado para retornar um arquivo *XMI*

A seguir, é apresentada a descrição do protocolo distribuído, implementado no presente trabalho, para acessar os elementos da estrutura de dados distribuída *MDRChannel*.

5.4.2 Protocolo Distribuído

No desenvolvimento do protocolo distribuído, utilizado no presente trabalho, foram utilizados dois *patterns,* relacionados ao desenvolvimento de aplicações distribuídas. Estes *patterns,* juntamente com a sua forma de utilização usando *JavaSpaces,* são apresentados em (FREEMAN; HUPFER; ARNOLD, 1999).

O primeiro *pattern* utilizado é o *application pattern* chamado *replicated-worker pattern,* também conhecido como *master-worker pattern* ou, *master-slave pattern* (BUSHMAN; MEUNIER; ROHNERT; SOMMERLAD; STAL, 1996). Basicamente, este *pattern* descreve um cenário onde haverá um ou mais elementos chamados de *masters* que criarão tarefas para serem executadas por vários *workers.*

Um *master,* então, cria uma tarefa e a envia para os *workers.* Os *workers,* por sua vez, processam a tarefa e devolvem o resultado para o *master.*

(FREEMAN; HUPFER; ARNOLD, 1999) destacam duas qualidades importantes deste *application pattern*:

- este *pattern* faz automaticamente balanceamento de carga, pois os *workers* processam uma tarefa após a outra. Tão logo tenham terminado uma tarefa estão prontos para processar outra. Digamos que um *worker* A recebeu uma tarefa muito demorada. Enquanto o *worker* A processa a tarefa outro *worker* B pode estar processando outras tarefas. Os *workers* executam um maior ou menor número de tarefas de acordo com sua capacidade e disponibilidade;
- as aplicações que utilizam este *pattern* escalam naturalmente, pois se houver um aumento ou diminuição da demanda de tarefas a serem executadas, basta adicionar ou remover *workers* da aplicação sem que seja necessário outras modificações na mesma;

É importante notar que estas qualidades vão ao encontro das características desejáveis ao suporte a persistência de artefatos apresentadas na seção 4.8.

No presente trabalho, os repositórios remotos de metadados *RemoteMDR* correspondem aos *workers* do *pattern*, enquanto que os clientes usuários dos

repositórios são os *masters*.

O segundo *pattern,* utilizado no protocolo distribuído, é o *command design pattern* (GAMMA; HELM; JOHNSON; VLISSIDES, 1995) (FREEMAN; HUPFER; ARNOLD, 1999). A aplicação deste *pattern* é simples e consiste no fato de que toda tarefa, a ser enviada para um repositório de metadados remoto deverá implementar uma interface específica que define o método que conterá os passos de execução da tarefa.

Com a utilização deste *pattern,* é possível criar vários tipos de tarefas diferentes, cada qual contendo os passos particulares de sua execução. O repositório de metadados remoto é responsável por receber a tarefa e executar os passos definidos nesta tarefa. Neste caso, as tarefas enviadas para o repositório de metadados remoto, contém não apenas dados, mas também comportamento, permitindo que um repositório de metadados remoto execute uma tarefa sem conhecê-la de antemão.

Com a utilização dos *patterns*, apresentados acima, foi possível a elaboração de um protocolo distribuído com o seguinte funcionamento:

- um ou mais repositórios buscam no espaço compartilhado objetos que contenham as requisições de tarefas a serem executadas. Ao receber um destes objetos, o repositório remoto *RemoteMDR* executa a tarefa especificada no objeto fazendo uma chamada à *API* (*Application Programming Interface*) do repositório de metadados *MDR*. Depois, o repositório de metadados remoto envia para o espaço compartilhado *JavaSpaces* um outro objeto contendo o resultado da execução da tarefa;
- finalmente, o cliente que enviou a requisição da tarefa, busca no espaço compartilhado *JavaSpaces* o objeto que contém o resultado da execução da mesma.

A partir deste ponto as interações entre um usuário do *DiSEN*, os repositórios remotos e o espaço distribuído, serão ilustradas com o auxílio de diagramas de seqüência. Por exemplo, a interação entre o usuário do *DiSEN* e o repositório remoto de metadados está representada no diagrama de seqüência da figura 18.

Havendo mais de um repositório remoto de metadados no ambiente distribuído, eles irão concorrer no processamento das tarefas, requisitadas pelos clientes. Aquele que, primeiro, terminar de processar a requisição, enviará o resultado para o espaço compartilhado *JavaSpaces.*

O repositório remoto que terminar a tarefa depois, irá perceber que já existe um objeto com o resultado no espaço compartilhado *JavaSpaces* e se limitará a comparar o resultado da execução da tarefa que ele processou, com o resultado que já foi enviado para o espaço, atualizando seus registros de *Log* com as informações da tarefa processada.

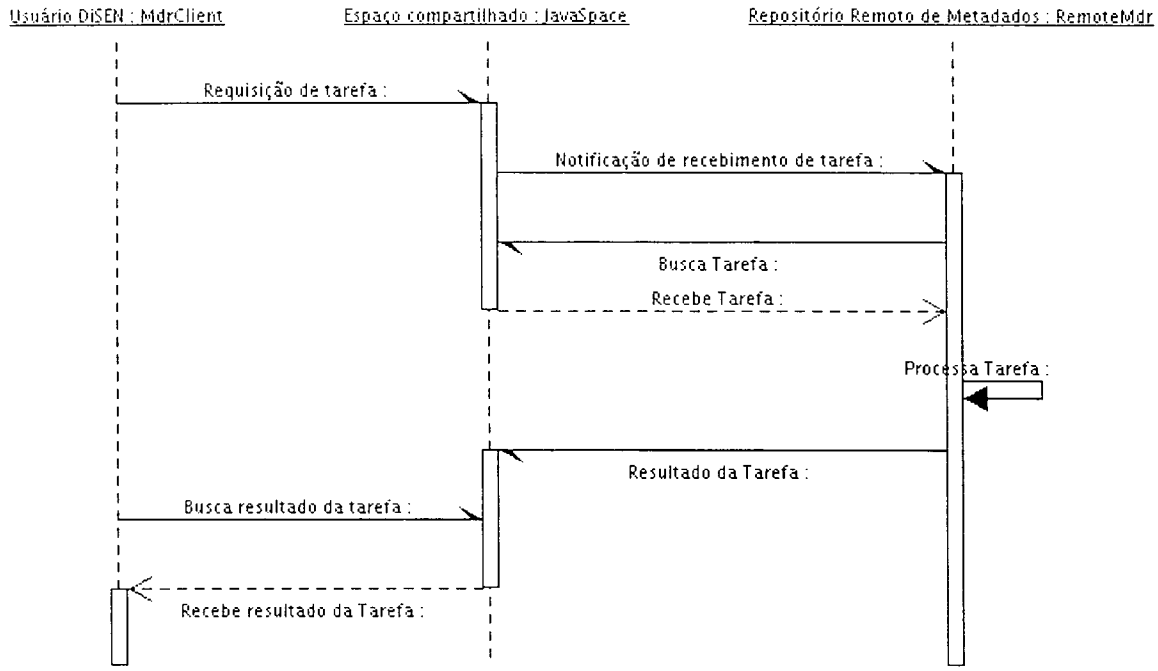

Figura 18 - Interação entre o cliente e o repositório remoto de metadados

Existem situações nas quais um repositório remoto A pode ser iniciado, ou voltar à atividade depois de um período de ausência e encontrar um cenário onde existem outros repositórios remotos em atividade, processando requisições de tarefas com a numeração de seqüência igual a N, enquanto ele mesmo, o repositório A, processou apenas tarefas até a numeração de seqüência N-X. X, então, é o número de requisições de tarefas que o repositório remoto A precisa

processar para estar apto a processar a requisição de tarefa com numeração de seqüência igual a N, juntamente com os outros repositórios remotos.

Pode-se dizer, então, que o repositório A está desatualizado com relação às tarefas que estão sendo enviadas para processamento pelos clientes. Para casos como este, foram implementadas duas abordagens para a atualização do repositório.

A primeira abordagem, representada no diagrama de seqüência da figura 19, pode ser utilizada quando no espaço compartilhado ainda existirem todas as X requisições de tarefas (de N-X até N) que o repositório remoto A precisa processar para poder, aí sim, continuar processando as requisições de tarefas com numeração de seqüência N em diante.

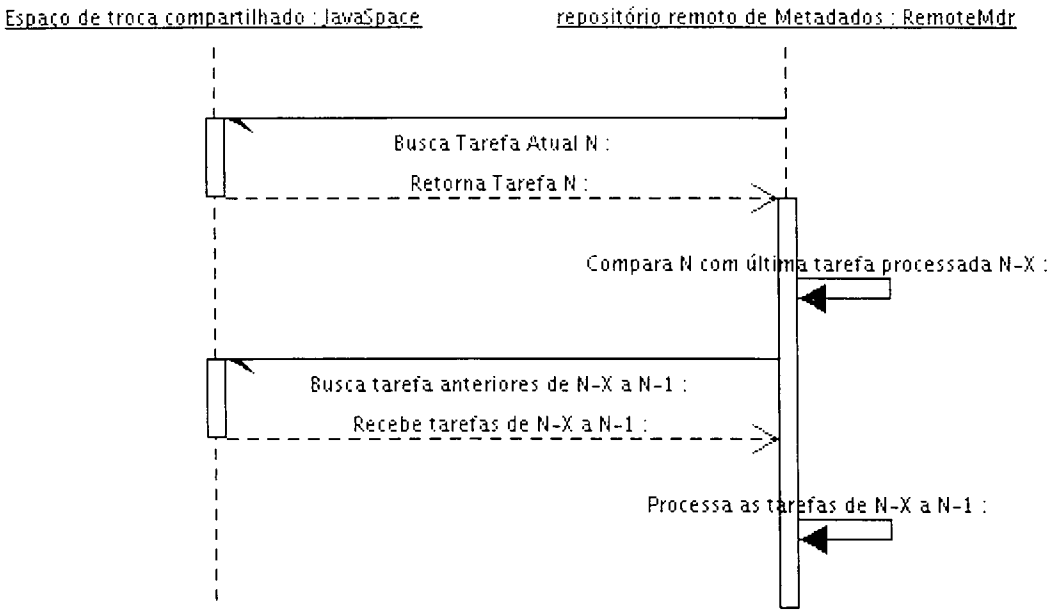

Figura 19 - Primeira abordagem de atualização

Para se atualizar, o repositório remoto A acessa o espaço compartilhado, faz a leitura de todas as X requisições de tarefas (de N-X até N) e as processa, uma a uma, seqüencialmente. Findo este passo, o repositório remoto de metadados A pode continuar processando as requisições de tarefas com numeração N em diante.

É importante observar que para se atualizar, o repositório remoto A precisa ler e processar apenas as requisições de tarefas que resultem em alteração no seu

conteúdo. Requisições de tarefas, relacionadas à consulta do conteúdo do repositório, não precisam e não devem ser processadas para que se possa diminuir o tempo gasto na atualização do repositório, melhorando, assim, o desempenho do processo.

A segunda abordagem, representada no diagrama de seqüência da figura 20, é necessária quando no espaço compartilhado não estiverem mais disponíveis as X requisições de tarefas. A pedido do administrador dos repositórios remotos, por exemplo, pode estar sendo mantido no espaço compartilhado apenas as 50 últimas requisições de tarefas e X neste caso seria maior que 50.

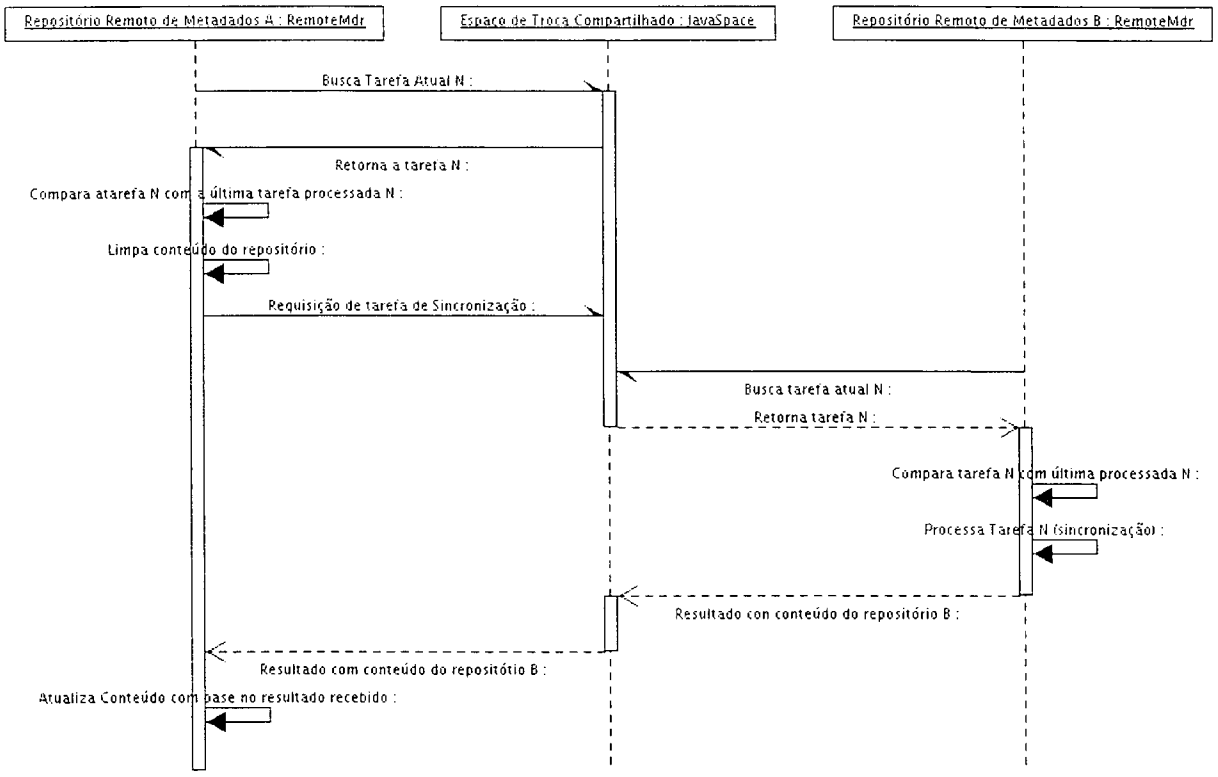

Figura 20 - Segunda abordagem de atualização

A segunda abordagem, também, é adequada, quando o número de requisições de tarefas X que precisam ser processadas pelo repositório remoto A é muito grande comparada ao conteúdo dos outros repositórios remotos. Pode ser que os repositórios remotos de metadados estejam armazenando alguns poucos

modelos e artefatos e que, por outro lado, estes artefatos e modelos estejam sofrendo diversas alterações, gerando, conseqüentemente, uma grande quantidade de requisições e tarefas no ambiente compartilhado *JavaSpaces*.

Nestes dois casos, a segunda abordagem é indicada. Nesta abordagem o repositório remoto A envia, também, uma requisição de tarefa para o espaço compartilhado, da mesma forma que os clientes dos repositórios remotos. Esta requisição de tarefa, especificamente, é um pedido para que outro repositório remoto, neste caso um que esteja atualizado, envie todo o seu conteúdo ao repositório remoto A desatualizado que enviou a requisição de tarefa.

Esta requisição de execução de tarefa será recebida por um repositório remoto atualizado B, que em resposta, irá gerar vários resultados, com o conteúdo do seu repositório de metadados. O repositório remoto de metadados B irá, então, enviar para o espaço compartilhado vários objetos com o conteúdo do seu repositório de metadados *MDR*. Todos estes resultados serão recebidos pelo repositório remoto desatualizado A que os utiliza para atualizar o conteúdo do seu repositório de metadados.

Ao fim deste processo o repositório remoto de metadados A, que estava desatualizado, estará com o conteúdo igual ao repositório remoto B no momento em que este recebeu a requisição de execução da tarefa com numeração de seqüência igual a N. A partir desse momento o repositório remoto A pode novamente tentar buscar e processar a tarefa corrente N.

Finalmente na seção 5.4.3, a seguir, é apresentado a forma como os elementos desenvolvidos no presente trabalho se encaixam na arquitetura do *DiSEN*.

5.4.3 Arquitetura

O suporte à persistência de artefatos, para o ambiente distribuído *DiSEN*, proposto e desenvolvido no presente trabalho está de acordo com a arquitetura do *DiSEN* proposta em (PASCUTTI, 2002).

A figura 21 apresenta a camada de "Gerenciamento de Aplicação (distribuição/coordenação)" do *DiSEN* com a inclusão do *DART* no lugar do

elemento que, na figura 6 da seção 4.7, aparecia simplesmente como "repositório". Na figura 21 está representado o *DART*, que por sua vez é composto pelo repositório de metadados, bem como, pelo *RemoteMDR* e pelo *RemoteMDRClient.*

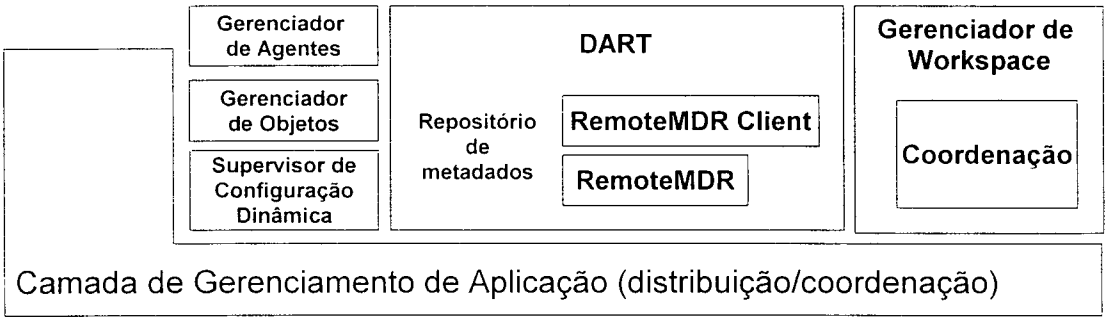

Figura 21 - Arquitetura do *DiSEN* com o *DART*

Na implementação do *DART*, para o repositório de metadados, foi utilizado o repositório *MDR,* conforme descrito na seção 5.2, enquanto que o *RemoteMDR* e o *RemoteMDRCliente* fazem parte das classes implementadas para possibilitar a comunicação de clientes do *DiSEN* com um ou mais repositórios de metadados distribuídos na rede.

O *RemoteMDRClient* deverá ser utilizado por qualquer aplicação que queira enviar tarefas de persistência para os repositórios remotos de metadados distribuídos no ambiente *DiSEN*. É ele que permite que uma aplicação inclua tarefas na estrutura de dados distribuída no espaço compartilhado *JavaSpaces* e busque ali, também, os resultados destas tarefas. Por exemplo, o *MDRExplorer* faz uso do *RemoteMDRClient* para poder se comunicar com o *DART.*

Já o *RemoteMDR* faz a ligação de um repositório de metadados *MDR* e o ambiente *DiSEN*. É ele que busca tarefas para serem processadas no espaço compartilhado *JavaSpaces,* as processa utilizando o repositório de metadados *MDR* e envia para o *JavaSpaces* o resultado do processamento.

A estrutura distribuída de dados está representada em "Coordenação" no "Gerenciador de *Workspace*", enquanto que o protocolo distribuído está implementado tanto em *RemoteMDR* como no *RemoteMDRClient.*

É importante ressaltar que a estrutura de dados distribuída, desenvolvida neste trabalho e utilizada na coordenação da comunicação entre o *RemoteMDRClient* e o *RemoteMDR*, pode, futuramente, ser utilizada, também, na coordenação da comunicação entre outros elementos da arquitetura *DiSEN*. Assim, enquanto que o protocolo distribuído desenvolvido no presente trabalho, está mais diretamente ligado a implementação do *RemoteMDRClient* e *RemoteMDR*, a estrutura de dados distribuída, por sua vez, está ligada à coordenação da comunicação entre elementos da arquitetura *DiSEN* e usuários do ambiente.

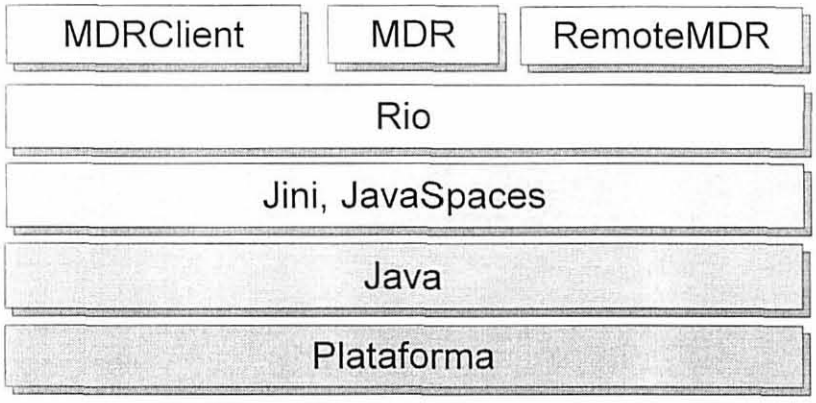

Figura 22 - Camada da arquitetura do *DART*

A figura 22 apresenta a arquitetura do *DART* dividida em camadas, enquanto que a figura 23 apresenta o diagrama das classes criadas no processo de desenvolvimento do protótipo do *DART*. As classes estão organizadas em pacotes e camadas de acordo com a arquitetura do *DART* e do *DiSEN*. Estas classes contém a implementação do suporte a persistência para o ambiente *DiSEN*.

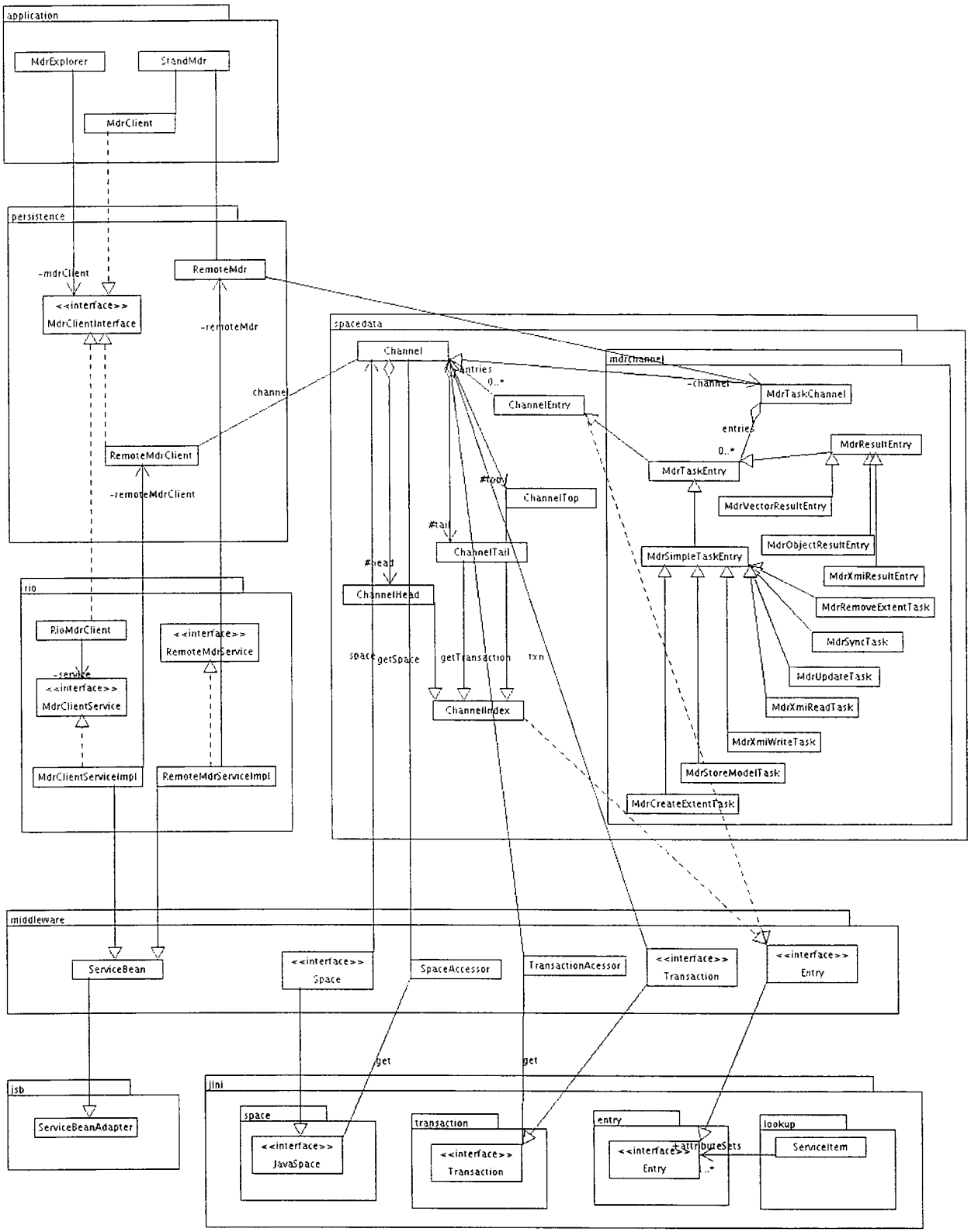

Figura 23 - Classes e pacotes usados na implementação do *DART*

5.5 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentada a criação do *DART* para o ambiente *DiSEN.* Foram apresentadas as abordagens adotadas para a implementação das características desejáveis apresentadas na seção 4.8.

No próximo capítulo, são apresentados os testes realizados com o suporte à persistência de artefatos do ambiente distribuído *DiSEN e* os resultados alcançados.

6 RESULTADOS OBTIDOS

6.1 INTRODUÇÃO

Para a validação do *DART* foi criada uma pequena aplicação cliente chamada *MDRExplorer*. Basicamente o *MDRExplorer* pode explorar o conteúdo dos repositórios de metadados remotos que estão no ambiente *DiSEN*.

O *MDRExplorer* é um exemplo de uma aplicação cliente, ligada ao *DiSEN* que faz uso das funcionalidades do repositório de metadados *MDR*, através da rede.

Utilizando o *MDRExplorer* é possível utilizar as seguintes funcionalidades de repositórios de metadados *MDR* que estejam presentes de forma distribuída no ambiente *DiSEN:*

- criar um sub-repositório a partir de um modelo de metadados existente no repositório;
- remover um sub-repositório;
- carregar elementos de um modelo de metadados em um sub-repositório lendo o conteúdo de um arquivo *XMI;*
- gravar um arquivo *XMI* com o conteúdo de um sub-repositório;
- gerar as interfaces *Java*, segundo a especificação *JMI*, que correspondem aos elementos de um modelo de metadados armazenado em um sub-repositório;
- armazenar dentro de um sub-repositório um artefato descrito em determinado modelo de metadados.

A seguir é descrito um exemplo de utilização do *MDRExplorer* juntamente com o ambiente *DiSEN* para armazenar e recuperar um artefato *UML*. Logo após, na seção 6.3, são descritos os testes realizados com o protótipo implementado.

6.2 ESTUDO DE CASO

O exemplo aqui apresentado, mostra como utilizar a aplicação cliente *MDRExplorer* para armazenar no ambiente *DiSEN* um artefato *UML*, gerado a partir de uma ferramenta de modelagem e como recuperá-lo mais tarde.

O primeiro passo do exemplo é, justamente, a criação de um arquivo *XMI* que contenha um artefato *UML*. Para tanto, pode ser utilizada uma das muitas ferramentas de modelagem *UML* disponíveis no mercado. No caso deste exemplo em particular, foi utilizada a ferramenta *Poseidon for UML* (GENTLEWARE, 2003a).

Com relação à infra-estrutura necessária para a realização do experimento, é necessário que os serviços de infra-estrutura do ambiente *DiSEN* estejam em execução. No caso do suporte à persistência de artefatos, são necessários os serviços de:

- espaço compartilhado *JavaSpaces;*
- páginas amarelas *Lookup;*
- *gerenciador de Transações distribuídas;*
- *Provision Monitor;*
- e um ou mais *cybernodes;*

Destes serviços de infra-estrutura os três primeiros são serviços do *Jini*, enquanto que o *Provision Monitor* e os *cybernodes* fazem parte do projeto *Rio*. A figura 24 apresenta a janela da ferramenta *Rio Viewer* que mostra os serviços relacionados acima em execução em uma rede local.

Uma vez que a estrutura do ambiente distribuído está em execução, é possível iniciar um ou mais repositórios de metadados remotos no ambiente. Para tanto, são utilizados os recursos do projeto *Rio.*

Os repositórios de metadados remotos são incluídos no ambiente distribuído pelo *Provision Monitor.* Para que isso seja possível, é necessário que seja passado para ele uma *Operational String* com a descrição do serviço de repositórios de metadados remoto.

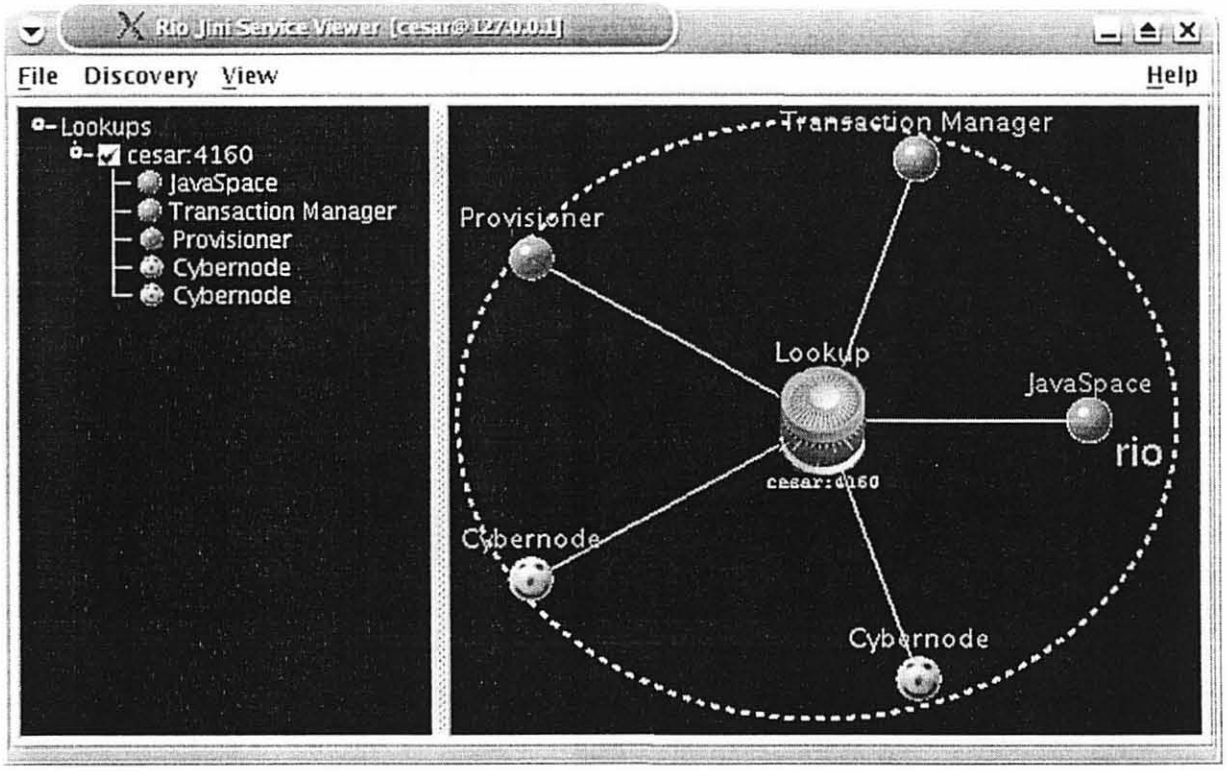

Figura 24 - Serviços necessários ao suporte a persistência de artefatos

Uma *Operational String* pode ser representada em um arquivo *XML*. Em anexos são apresentadas as *Operational Strings* do repositório de metadados remoto e da aplicação cliente *MDRExplorer*.

A figura 25 apresenta a janela da ferramenta *Operational String Viewer* que mostra uma representação gráfica das várias *Operational Strings* que descrevem os serviços em execução no ambiente distribuído.

Como foi explicado na seção 3.6 um *Operational String* pode ser passado para o *Provision Monitor* que se encarregará de inicializar os serviços necessários em nós (*cybernodes*) adequados.

A figura 26 apresenta mais uma vez a janela do *Rio Viewer* mostrando agora a execução de dois repositórios de metadados remotos (*RemoteMDR*) e uma aplicação cliente (*MDRExplorer*) no ambiente distribuído.

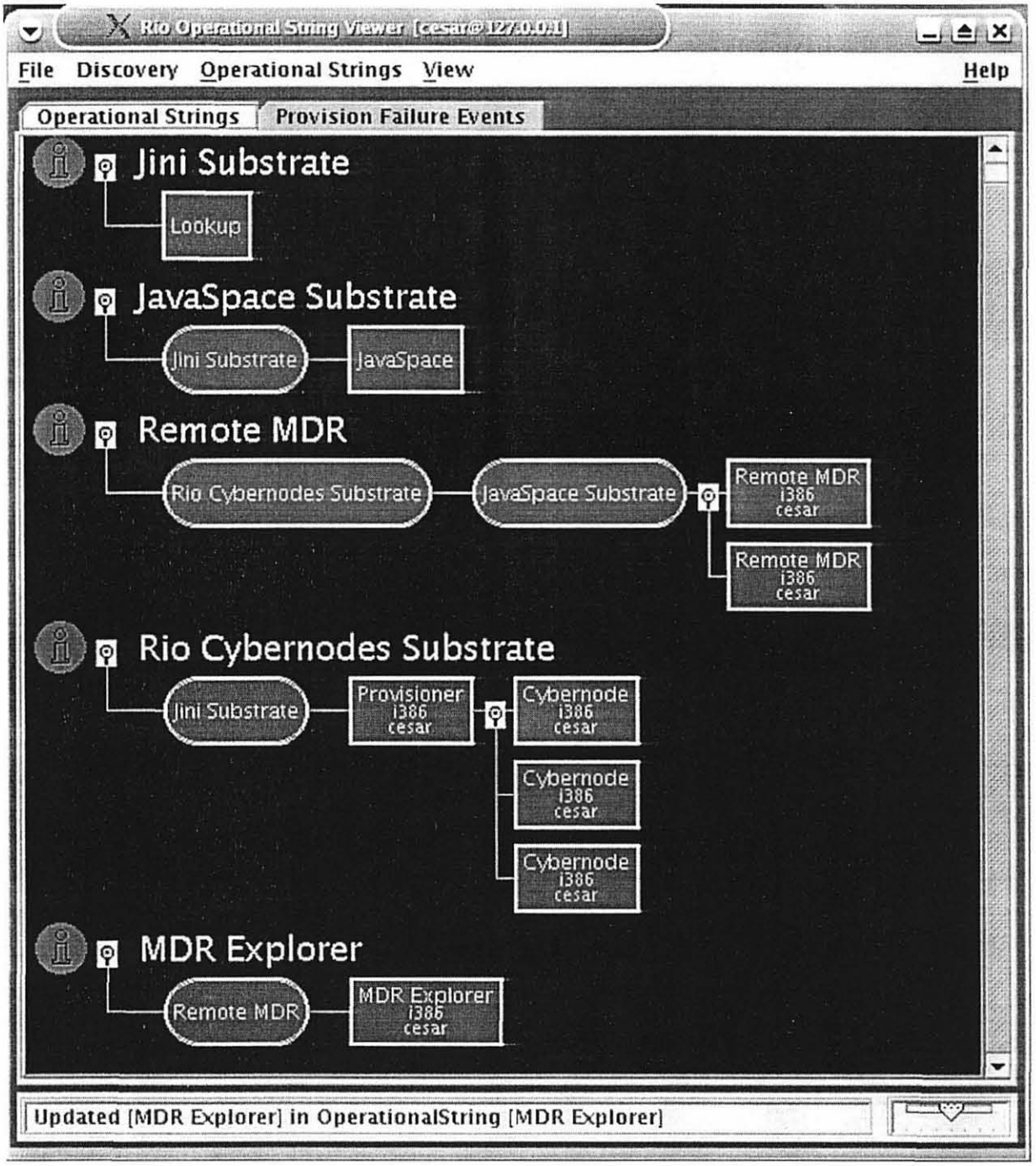

Figura 25 - *Operational Strings* dos serviços em execução na rede

Com a infra-estrutura do *DiSEN* em execução na rede local, inclusive com o suporte à persistência de artefatos, é possível agora começar a utilização da aplicação cliente *MDRExplorer.*

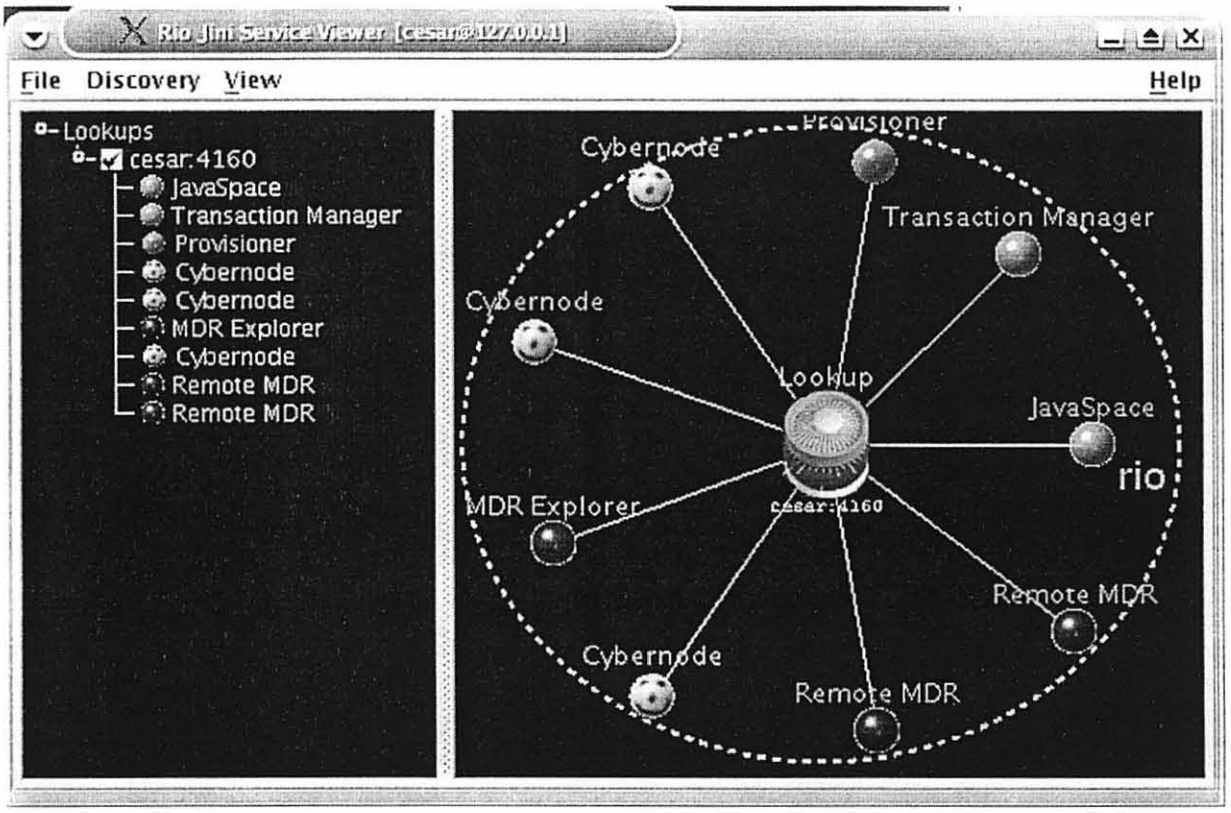

Figura 26 - Dois repositórios de metadados remotos e um cliente em execução

Para armazenar no ambiente *DiSEN* um artefato *UML*, basta selecionar o botão "*Store UML Model*" no menu lateral do lado direito da janela do *MDRExplorer.* Uma vez selecionado este botão é aberta uma janela para que se escolha o arquivo *XMI* que contém o artefato *UML* que se quer armazenar no *DiSEN* figura 27.

Após escolher o arquivo *XMI* o *MDRExplorer* envia uma tarefa requisitando a criação de um sub-repositório para artefatos *UML e* o armazenamento do conteúdo do arquivo *XMI* escolhido dentro deste sub-repositório. Os repositórios de metadados remotos do ambiente *DiSEN* processam as tarefas *e* retornam os resultados para *MDRExplorer.*

A figura 28 apresenta a janela do *MDRExplorer*, mostrando o conteúdo dos repositórios de metadados remotos do ambiente *DiSEN* após o armazenamento do artefato *UML.*

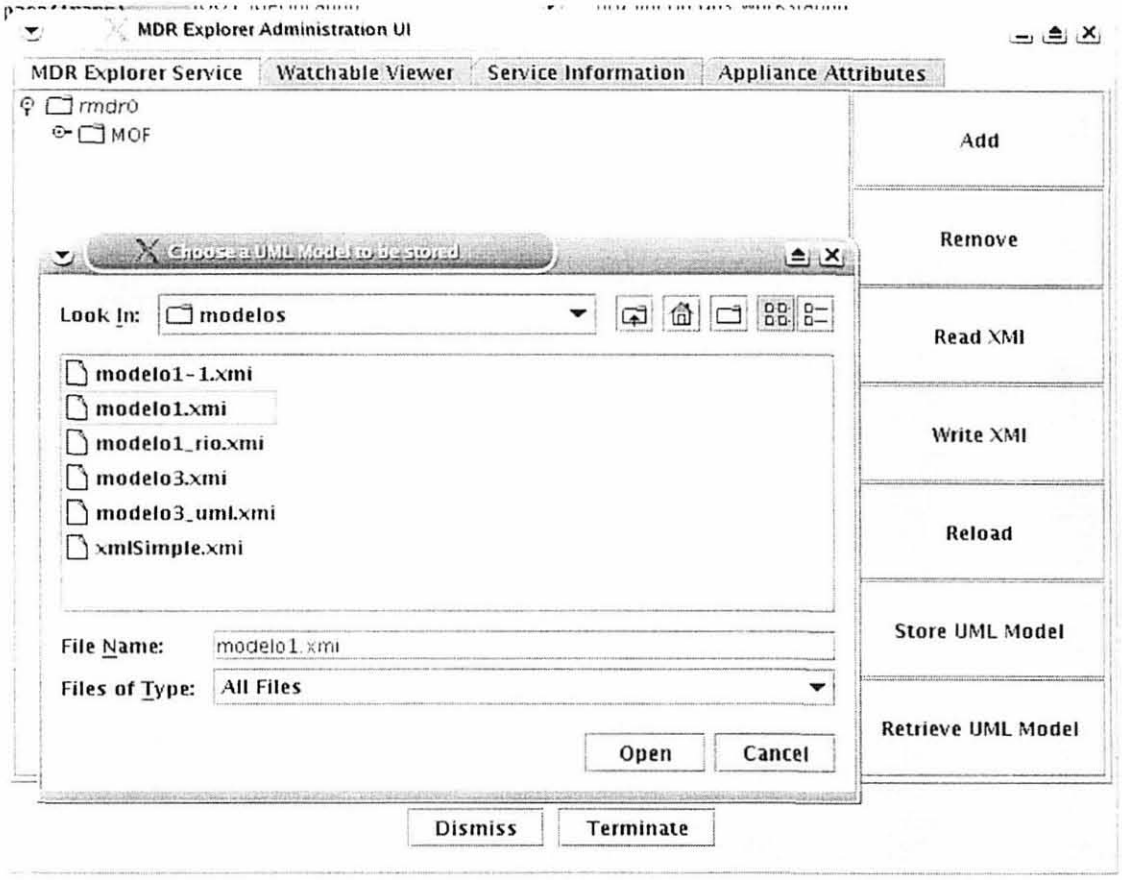

Figura 27 - Escolhendo o arquivo *XMI* para ser armazenado no *DiSEN*

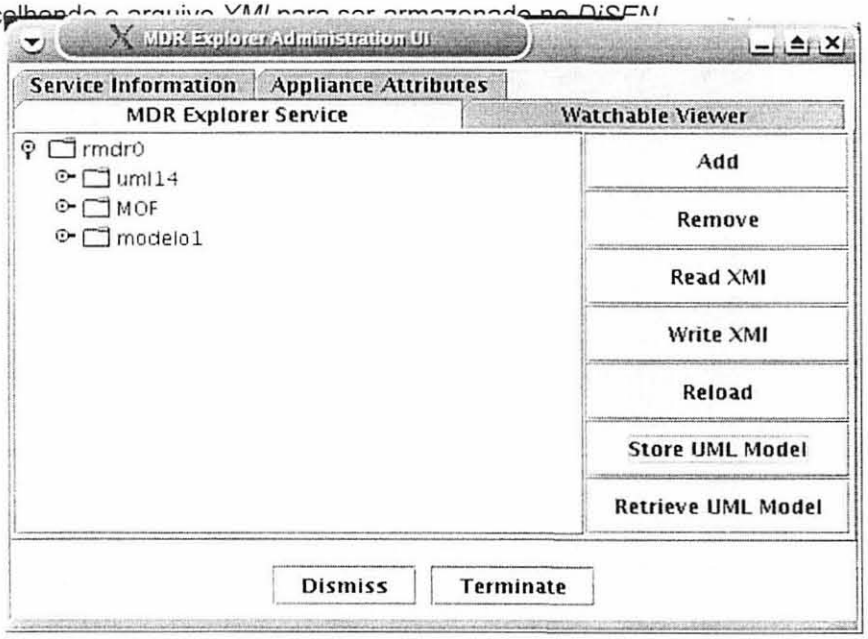

Figura 28 - Conteúdo dos repositórios de metadados

Um fato interessante, a ser notado, é que antes de enviar o artefato *UML*

para ser armazenado no *DiSEN*, o *MDRExplorer* mostra apenas o modelo *MOF* no conteúdo dos repositórios de metadados remotos. Após o armazenamento do artefato *UML*, o *MDRExplorer* mostra não só o modelo *MOF*, mas também o modelo *UML* e finalmente o artefato armazenado (MOF, UML e modelo1 respectivamente) .

Para recuperar um artefato *UML*, basta selecionar o botão *Retrieve UML Model* no menu lateral do *MDRExplorer* e escolher o sub-repositório que contém o artefato que se quer recuperar. Logo após será aberta uma outra janela para que seja informado o diretório local onde o artefato *UML* deve ser gravado figuras 29 e 30.

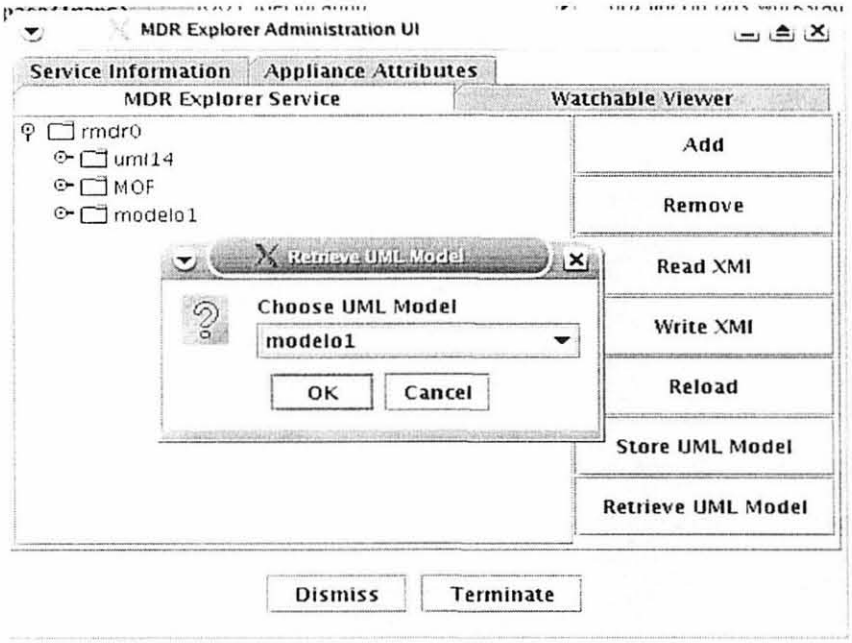

Figura 29 - Escolhendo o artefato *UML* a ser recuperado

Finalmente, o arquivo *XMI* que é gravado localmente pode então ser carregado novamente em uma ferramenta de modelagem *UML*. No exemplo aqui apresentado, foi utilizada a ferramenta *Poseidon for UML*, mas a princípio poderia ser utilizada qualquer outra ferramenta de modelagem *UML*.

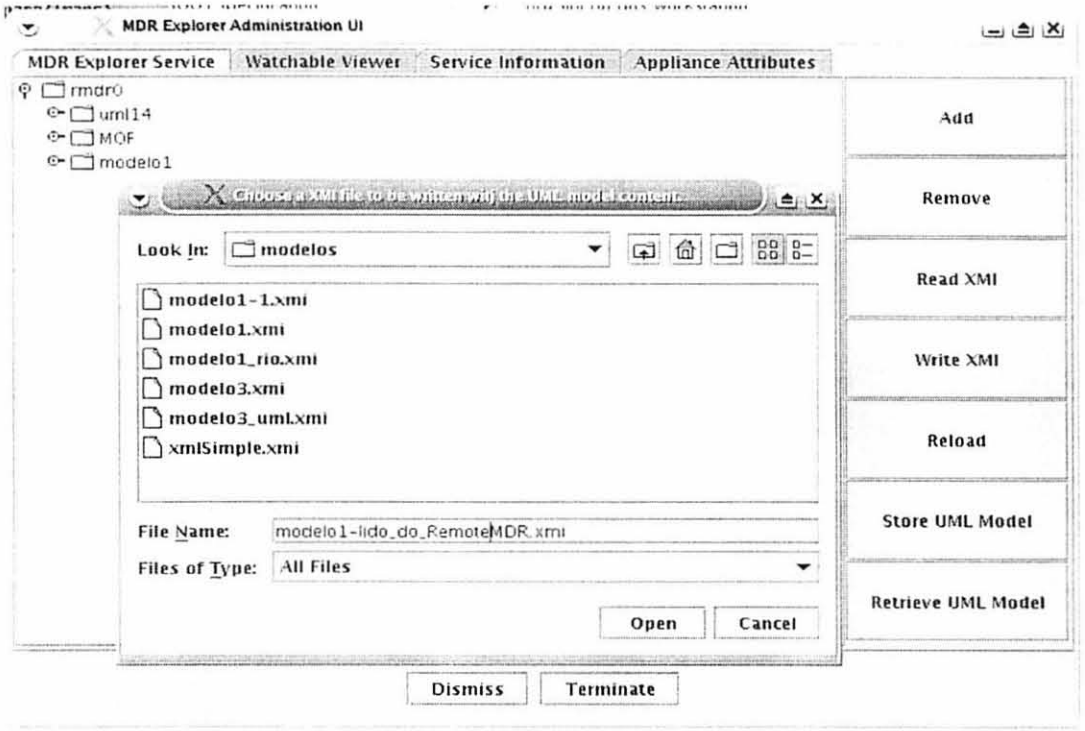

Figura 30 - Gravando a descrição *XMI* do artefato *UML*.

A seção seguinte descreve outros testes que foram realizados com relação a utilização do suporte à persistência de artefatos do ambiente *DiSEN* utilizando a aplicação cliente *MDRExplorer*.

6.3 TESTES DO SUPORTE À PERSISTÊNCIA

Para a validação do presente trabalho foram realizados vários testes de utilização do *DART* no suporte à persistência de artefatos no ambiente distribuído *DiSEN*. Nesta seção serão descritos os testes realizados e os resultados obtidos.

A tabela 6 apresenta um resumo dos testes de utilização das funcionalidades dos repositórios remotos de metadados através do *DiSEN* usando a aplicação cliente *MDRExplorer*.

Os testes apresentados na tabela 6 foram realizados em diferentes cenários de execução do ambiente *DiSEN*. Foram considerados os cenários em que poderiam haver:

- apenas um cliente e apenas uma instância do *DART*;

- apenas um cliente e duas ou mais instâncias do *DART*;
- dois ou mais clientes e apenas uma instância do *DART* e;
- dois ou mais clientes e duas ou mais instâncias do *DART*.

Tabela 6 - Testes de utilização das funcionalidades do suporte a persistência

Teste Realizado	Um Cliente X Um Repositório	Um Cliente X dois ou mais Repositórios	Dois ou mais clientes X Um Repositório	Dois ou mais clientes X Dois ou mais Repositórios
Criação de sub-repositório	Ok	Ok	Ok	Ok
Remoção de sub-repositório	Ok	Ok	Ok	Ok
Leitura de arquivo *XMI*	Ok	Ok	Ok	Ok
Escrita de arquivo *XMI*	Ok	Ok	Ok	Ok
Geração de interface *JMI*	Ok	Ok	Ok	Ok
Listagem do conteúdo	Ok	Ok	Ok	Ok
Leitura de arquivo *XMI*	Ok	Ok	Ok	Ok

A tabela 7 a seguir mostra um resumo dos testes realizados com o suporte à persistência de artefatos no *DiSEN*, com relação a falhas parciais e escalabilidade dos elementos componentes do mesmo.

Tabela 7 - Testes de recuperação de falhas parciais

Teste Realizado	Um Cliente X Um Repositório	Um Cliente X dois ou mais Repositórios	Dois ou mais clientes X Um Repositório	Dois ou mais clientes X Dois ou mais Repositórios
Atualização do repositório com base no conteúdo de outro repositório	Não se aplica	Ok	Não se aplica	Ok
Atualização do repositório com base no conteúdo do espaço compartilhado	Ok	Ok	Ok	Ok
Atualização do espaço com base no conteúdo de um repositório	Ok	Ok	Ok	Ok
Aumento e diminuição de clientes e repositórios (escalabilidade)	Ok	Ok	Ok	Ok
Queda e reinicialização de um repositórios durante a execução (falha parcial)	Ok	Ok	Ok	Ok

Teste Realizado	Um Cliente X Um Repositório	Um Cliente X dois ou mais Repositórios	Dois ou mais clientes X Um Repositório	Dois ou mais clientes X Dois ou mais Repositórios
Queda e reinicialização de todos os repositórios durante a execução (falha parcial)	Ok	Ok	Ok	Ok
Queda e reinicialização do espaço compartilhado durante a execução (falha parcial)	Ok	Ok	Ok	Ok

Para os testes de escalabilidade foi considerado um ambiente distribuído de desenvolvimento de software em uma rede local. Assim, o objetivo do teste não foi dimensionar, ou verificar o impacto no desempenho da adição de vários usuários ao suporte à persistência. O objetivo do teste de escalabilidade foi identificar possíveis erros no protocolo distribuído, desenvolvido no presente trabalho, que pudessem levar a *locks* indesejados no funcionamento do suporte à persistência de artefatos.

Já com relação às falhas, basicamente, foi testada a capacidade do suporte à persistência de artefatos do ambiente *DiSEN*, com relação à detecção e recuperação de falhas parciais no funcionamento dos repositórios de metadados remotos e do espaço compartilhado. Para tanto, foram simuladas quedas tanto dos repositórios de metadados, quanto do espaço compartilhado. Mais uma vez, o principal objetivo destes testes foi identificar possíveis erros e/ou problemas no protocolo distribuído deste trabalho.

Nos testes de falha parcial apresentados na tabela 7, quando um dos serviços necessários à infra-estrutura do suporte à persistência de artefatos do ambiente *DiSEN* torna-se indisponível, o *Provision Monitor* do projeto *Rio* se encarrega de iniciá-lo novamente no ambiente distribuído.

Por outro lado, é o protocolo distribuído apresentado na seção 5.4.2 que se encarrega de atualizar o conteúdo dos repositórios de metadados remotos, ou mesmo, do espaço compartilhado, se alguns destes for iniciado com o seu conteúdo desatualizado.

6.4 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados, um breve estudo de caso descrevendo a utilização do *DART* e, também, alguns testes realizados com relação à sua utilização.

Finalmente o próximo capítulo apresenta as conclusões do presente trabalho e as propostas de trabalhos futuros.

7 CONCLUSÃO

O presente trabalho apresentou a implementação, para o ambiente distribuído *DiSEN,* do suporte à persistência de artefatos baseado em repositórios de metadados, segundo a especificação *MOF*.

Para o suporte à persistência de artefatos no *DiSEN,* foi construído um protótipo de um repositório distribuído de artefatos chamado *DART.*

Uma das principais contribuições do presente trabalho é a criação dos elementos da camada de infra-estrutura de distribuição da arquitetura *DiSEN,* que possibilitam a coordenação da comunicação entre aplicações clientes e o *DART*.

A abordagem adotada, com a utilização de um serviço *JavaSpaces* juntamente com a elaboração de uma estrutura de dados distribuída e um protocolo distribuído utilizando os *patterns Master-Worker* e *Command,* adicionaram ao *DART*, alta coesão e fraco acoplamento, além de escalabilidade e balanceamento de carga.

Somado a isso, a utilização de *Jini, JavaSpaces* e *Rio,* foi possível atender às características desejáveis apontadas, na seção 4.8, como relacionado na tabela 3 da seção 5.3.

O presente trabalho, com a adoção de um repositório de artefatos em acordo com a especificação *MOF*, coloca o ambiente distribuído *DiSEN* no contexto da *MDA*, aumentando a motivação de continuar as pesquisas a respeito da *MDA,* utilizando o *DART*.

Apesar dos resultados obtidos até o momento, confirmarem a viabilidade da abordagem adotada, é necessário que se apontem algumas limitações encontradas durante a execução do presente trabalho.

Na realização deste trabalho foi utilizada uma implementação do serviço de espaço compartilhado *JavaSpaces,* fornecido pela própria *SUN,* juntamente com *Jini*. Devido ao fato desta implementação ser apenas uma implementação de referência da especificação do serviço *JavaSpaces*, ela é adequada para a utilização no protótipo apresentado no presente trabalho. Entretanto, em um ambiente de produção, para aumentar a robustez e disponibilidade da solução como um todo, seria necessário utilizar uma outra implementação de *JavaSpaces* que possibilitasse, por exemplo, que ele fosse executado, de forma espelhada, em dois

ou mais nós do ambiente distribuído. Isso possibilitaria um aumento na disponibilidade do sistema, de forma que se um deles falhasse, o outro poderia continuar operando.

Em um primeiro momento, existe a expectativa de que o *DART* possa ser utilizado por diferentes ferramentas de desenvolvimento de software. Entretanto, tem-se a consciência de que o presente trabalho limita-se a adotar a utilização de um repositório de artefatos, baseado em especificações de padrões abertos (*MOF*, *XMI*, e *JMI*), e que é, ainda, necessário que as ferramentas de desenvolvimento de software que forem utilizar o suporte à persistência de artefatos do ambiente *DiSEN,* também adotem estes mesmos padrões.

Neste sentido, já é possível verificar que diversas ferramentas de desenvolvimento de software, presentes no mercado, têm incorporado recursos para a gravação de seus artefatos no formato *XMI*. Entre elas pode-se citar *Rational Rose, Borland Together Control Center, Poseidon for UML*, etc (ver seção 4.3.1). Apesar disso, ainda existem incompatibilidades entre os arquivos *XMI,* gerados por diferentes ferramentas.

Antecipa-se que com o passar do tempo, as especificações dos padrões adotados no presente trabalho (*MOF, XMI* e *JMI*) amadureçam e que sejam publicadas novas versões das mesmas. Igualmente, espera-se que no decorrer deste processo de amadurecimento, surjam implementações atualizadas e mais robustas destas especificações, solucionando, por exemplo, as incompatibilidades existentes ainda hoje.

Entretanto, até que este amadurecimento ocorra, será possível a criação de filtros, usando arquivos *XSLT*, que possam ser usados na adequação de possíveis incompatibilidades entre arquivos *XMI,* gerados por diferentes ferramentas de desenvolvimento de software e o repositório de metadados, utilizado pelo *DART*.

7.1 TRABALHOS FUTUROS

Uma primeira proposta de trabalhos futuros é, justamente, a continuação da realização de testes de leitura e gravação, utilizando o *DART*, dos arquivos *XMI,*

gerados pelas diferentes ferramentas de modelagem presentes no mercado e a criação dos arquivos *XSLT*, necessários à sua execução sem erros.

Assim, em um ambiente distribuído, diferentes ferramentas de desenvolvimento de software já poderiam começar a utilizar o suporte à persistência de artefatos oferecido pelo *DART*.

A desvantagem de se utilizar arquivos *XSLT* para fazer a filtragem dos arquivos XMI, a serem armazenados no repositório de artefatos, é a exclusão das informações colocadas nas tags proprietárias pelo fabricantes das ferramentas. .

Outra abordagem para o mesmo problema, que poderia ser investigada, é a criação de arquivos *XMI* que contenham a descrição de modelos *UML*, incluindo as extensões usadas por alguns fabricantes. Dessa forma, seria possível armazenar os artefatos, gerados por estas ferramentas no repositório de artefatos, sem a exclusão das *tags* proprietárias, presentes nos arquivos *XMI*.

Ainda assim, seria necessário a filtragem das *tags* proprietárias, no caso de uma ferramenta de um fabricante querer ler um artefato gerado por outra ferramenta. Entretanto, o compartilhamento de artefatos entre ferramentas do mesmo fabricante seria facilitada.

Outra sugestão é, nos trabalhos dos pesquisadores do *GESSD*, incorporar a utilização do *DART* às ferramentas de desenvolvimento de software que estão sendo criadas, como por exemplo na *Requisite* (BATISTA, 2002) e no *DiManager* (PEDRAS, 2002).

Dado o fato de que o protótipo implementado não tinha como prioridade abordar os aspectos relacionados especificamente ao desempenho, outra sugestão de trabalhos futuros é, o estudo e aplicação de técnicas e estratégias para aumentar o desempenho do suporte à persistência de artefatos do ambiente *DiSEN*.

Finalmente, não deve-se esquecer a necessidade de abordar os aspectos relacionados ao controle de versões e ao controle de acesso, conforme indicados na seção 4.8.

Assim, apesar de, no atual momento, o ambiente *DiSEN* contar com um protótipo que adiciona ao ambiente o suporte à persistência de artefatos que pode ser utilizado pelas aplicações clientes conectadas à ele, verifica-se que ainda se faz necessário a realização de outros trabalhos para que seja possível atender a todas

as características desejáveis apresentados na seção 4.8.

REFERÊNCIAS

ABOUT the Jini Community, Disponível em: <http://www.jini.org/about/community.html>, Acesso em: 18 mai. 2003.

ADAPTIVE Ltd., **Adaptive Repository**, Disponível em: <http://www.adaptive.com/assets/downloads/adaptive_repository.pdf>, Acesso em: 20 jun. 2003

BATISTA, S. M. **Proposta de uma Ferramenta de Groupware para Apoio à Definição de Requisitos para a MDSODI**. 2002. Exame de Qualificação (Mestrado) - DIN-UEM/UFPR, Maringá.

BOHLEN, M.; ANDROMDA TEAM **AndroMDA from UML to deployable components**, Disponível em: <http://www.andromda.org/>, Acesso em 18 mai. 2003.

BOLDYREFF, C.; SMITH, M.; WEISS, D.; **Environment to Support Collaborative Software Enginering,** 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes, University of Sannio, Benevento, Italy, 25 March 2003.

BORLAND SOFTWARE CORPORATION, **Borland® Together® ControlCenter**, Borland Software Corporation. Disponível em: <http://www.togethersoft.com/products/controlcenter/features/features_brochure.pdf>, Acesso em: 19 mai. 2003.

BUSHMAN, F. ; MEUNIER, R. ; ROHNERT, H. ; SOMMERLAD, P. ; STAL, M. **A System of Patterns**, John Wiley & Sons Ltd, England, 1996.

CONCURRENT Versions System, CVS, Disponível em: <http://www.cvshome.org/>, Acesso em: 01 jun. 2003.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG T. **Distributed Systems - Concepts and Design**, Pearson Education Edinburgh Gate Ltd, , Harlow, England, 2001.

DEIFEL, B.; SALZMANN C. **Requirements and Conditions for Dynamics in Evolutionary Software Systems**, Institute for Informatics, Munich University of Technology, 80290 Munich, Germany, 12/05/1999.

DEMICHIEL, L. G.; YAÇINALP, L. Ü.; KRISNAN S. Enterprise JavaBeans TM Specification, Version 2.0, Sun Microsystems, Inc, August 14, 2001, Disponível em: <http:///java.sun.com/products/ejb/docs.html>, Acesso em: 15 mai. 2003.

DIRCKZE, R. **Java Metadata Interface(JMI) Specification**, JSR 040 Java Community Process, Version 1.0, Final Specification, 07-June-2002, Disponível em: <http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html>, Acesso em 23 mai. 2003.

DISTRIBUTED SYSTEMS TECHNOLOGY CENTRE, **dMOF User Guide**, Cooperative Research Centre for Distributed Systems Technology, Pty Ltd, University of Queensland, 2002.

FREEMAN, E.; HUPFER, S.; ARNOLD, K. **JavaSpaces Principles, Patterns, and Practice**, SUN Microsystems, Inc. Palo Alto, USA, November 1999.

GAMMA, E. ; HELM, R. ; JOHNSON, J. ; VLISSIDES, J. **Design Patterns – Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995.

GATES, L. **The changing face of repositories**, Application Development Trends, December 2001, Disponível em: <http://www.adtmag.com/article.asp?id=5748>, Acesso em: 30 jun. 2003.

GENTLEWARE, **Poseidon for UML**, Gentleware AG, Disponível em: <http:///www.gentleware.com/products/index.php3>, Acesso em: 18 mai. 2003a.

GENTLEWARE, **UML Diagram Interchange**, Gentleware AG, Disponível em: <http://www.gentleware.com/projects/diagraminterchange/>, Acesso em: 6 jun. 2003b.

GRAVENA, J. P. **Aspectos importantes de uma metodologia para desenvolvimento de software com objetos distribuídos**, Maringá, Paraná, Brasil, 2000. Trabalho de Graduação – Curso de Ciência da Computação, Centro de Tecnologia, Departamento de Informática, Universidade Estadual de Maringá.

HUZITA, E. H. M. **MOOPP - Uma Metodologia para auxiliar o desenvolvimento de aplicações para processamento paralelo**, São Paulo, 19995. tese (Doutorado em Engenharia Elétrica), Escola Politécnica – USP.

HUZITA, E. H. M. **Uma Metodologia para o Desenvolvimento Baseado em Objetos Distribuídos Inteligentes**. Projeto de pesquisa em andamento, Universidade Estadual de Maringá. Departamento de Informática, 1999.

JECKLE, M. **UML Tools (CASE & Drawing)**, May 30, 2003, Disponível em: <http://www.jeckle.de/umltools.html>, Acesso em: 06 jun. 2003.

JÉZÉQUEL, H.; HUSSMAN, S. **Cook (Eds.): UML 2002**, LNCS 2460, pp. 398 411, 2002. Verlag Berlin Heidelberg 2002, Disponível em: <http://link.springer-ny.com/link/service/series/0558/papers/2460/24600398.pdf>, Acesso em: 06 jun. 2003.

LANGIANO, B. Do C. **Um Mecanismo para Adaptação de Documentos XMI no Processo de Importação ao MDR,** Maringá, Paraná, Brasil, 2003. Trabalho de Graduação – Curso de Ciência da Computação, Centro de Tecnologia, Departamento de Informática, Universidade Estadual de Maringá.

MATULA, M. **UML2MOF Tool**, Netbeans/Sun Microsystems, Disponível em: <http://mdr.netbeans.org/uml2mof/>, Acesso em: 18 mai. 2003

META Integration Technology, Inc., **Meta Integration Repository Server (MIR)**, Disponível em: <http://www.metaintegration.net/Products/MIR/>, Acesso em: 20 jun. 2003

METAMATRIX, **metamatrix server** , Disponível em: <http://www.metamatrix.com/l3_metabase.html>, Acesso em 20 jun. 2003

NENTWICH, C.; EMMERICH, W.; FINKELSTEIN, A.; ZISMAN, A., **BOX: Browsing objects in XML**, Software-Practice and Experience, 2000.

NO MAGIG Inc. , **Magicdraw – UML Made Simple**, No Magic Inc., 27 May 2003, Disponível em: <http://www.magicdraw.com/files/MD_60_DataSheetA4.pdf?NMSESSID=3c58ee76b233e7b3b99cd1ba8b9cd2b4>, Acesso em: 06 jun. 2003.

NUTTER D.; BOLDYREFF C.;RANK S. **Active Artefact Management for Distributed Software Engineering**, Research Institute for Software Evolution, Department of Computer Science, University Of Durham, UK, 2002b.

NUTTER D.; BOLDYREFF C.;RANK S. **An Artefact Repository to Support Distributed Software Engineering**, Research Institute for Software Evolution, Department of Computer Science, University Of Durham, UK, 2002a.

OMG, Architecture Board ORMSC, **Model driven Architecture (MDA)**, Document number ormsc/2001-07-01, Object Management Group , July, 2001c.

OMG, **CommonObject Request Broker Architecture: CoreSpecification**, Object Management Group, Version 3.0.2, December 2002a, Disponível em: <http://www.omg.org/cgi-bin/apps/doc?formal/02-12-02.pdf>, Acesso em: 25 mai. 2003.

OMG, **OMG Meta Object Facility (MOF) Specification**, Object Management Group, Version 1.4, April 2002b, Disponível em: <http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf>, Acesso em: 25 mai. 2003.

OMG, **OMG Unified Modeling Language Specification,** Object Management Group, Version 1.5, March 2003a , Disponível em: http://www.omg.org/cgi-bin/doc?formal/03-03-01>, Acesso em: 25 mai. 2003.

OMG, **What Is OMG-UML and Why Is It Important?,** Object Management Group, Inc, 2003b, Disponível em: <http://www.omg.org/news/pr97/umlprimer.html>, Acesso em: 23 mai. 2003.

OMG, **XML Metadata Interchange (XMI) Specification,** Object Management Group, Version 2.0, May 2003c, Disponível em: <http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf>, Acesso em: 25 mai. 2003.

ORFALI; HARKEY; EDWARDS; **The Essential Distributed Objects Survival Guide**, John Wiley & Sons, Inc., New York, USA, 1996

PASCUTTI, M, **Proposta de Arquitetura de um Ambiente de Desenvolvimento de Software Distribuído Baseada em Agentes**, Porto Alegre, 2002. Dissertação (Mestrado em Ciência da Computação), Universidade Federal do Rio Grande do Sul.

PEDRAS, M. E. V. **Uma Ferramenta de Apoio ao Gerenciamento de Desenvolvimento de Software Distribuído**. 2002. Exame de Qualificação (Mestrado) - DIN-UEM/UFPR, Maringá.

POSTGRESQL, Disponível em: <http://www.postgresql.org>, Acesso em: 01 jun. 2003.

RATIONAL, **Rational Rose Model-Driven Development with UML**, Rational Software Corporation, 2002, Disponível em: <http://www.rational.com/media/products/rose/D185H_Rose.pdf>, Acesso em: 18 mai. 2003.

RIO PROJECT, **Rio Architecture Overview**, 15/02/2001, 2001b, Disponível em: <http://rio.jini.org/docs/rio_architecture_overview.pdf>, Acesso em: 10 jul. 2002

RIO PROJECT, **Rio Operational String**, 10/12/2001, 2001a, Disponível em: <http://rio.jini.org/docs/rio_opstring-v8.pdf>, Acesso em: 18 mai. 2003.

RIO PROJECT, **Rio Tools Overview**, 28/01/2002, Disponível em: <http://rio.jini.org/docs/rio_tools-v8.pdf>, Acesso em: 18 mai. 2003.

SALZMANN, C., **Design Principles for Dynamic Object Systems**, Institute for Informatics, Munich University of Technology, 80290 Munich, Germany, March 31, 2000.

SCHLAPBACH, A. **Generic XMI Support for the Moose Reengineering Environment**, June 17, 2001, Disponível em: <http://www.iam.unibe.ch/~scg/Archive/Projects/schlapbach.pdf>, Acesso em: 25 mai. 2003.

SCIENTIFIC COMPUTING ASSOCIATES **Maximum Performance Through Parallel Execution**, 2003, Disponível em: <http://lindaspaces.com/products/linda_overview.html>, Acesso em: 03 jul. 2003.

SUN, **JavaSpaces - Service Specification, Version 1.2.1**, SUN Microsystems, Inc. Palo Alto, USA, April 2002a.

SUN, **Jini - Architecture Specification, Version 1.2**, SUN Microsystems, Inc, Palo Alto, USA, December 2001.

SUN, **JINI NETWORK TECHNOLOGY Datasheet**, Sun Microsystems, Inc, Palo Alto, USA, 2001. Disponível em: <http://wwws.sun.com/software/jini/whitepapers/jini-datasheet0601.pdf>, Acesso em: 17 mai. de 2003d.

SUN, **MDR Users**, SUN Microsystems, Inc, Palo Alto, USA, Disponível em: <http://mdr.netbeans.org/users.html >, Acesso em: 7 jun. 2003b.

SUN, **METADATA Repository (MDR) home**, SUN Microsystems, Inc. Palo Alto, USA, Disponível em: <http://mdr.netbeans.org>, Acesso em: 10 jul. 2002b.

SUN, **METAMODELS Catalog**, SUN Microsystems, Inc, Palo Alto, USA, Disponível em : <http://mdr.netbeans.org/metamodels.html>, Acesso em: 10 mar. 2003c.

SUN, **NETBEANS Feature List**, SUN Microsystems, Inc, Palo Alto, USA, Disponível em: <http://www.netbeans.org/products/ide/features.html>, Acesso em: 18 mai. 2003a.

VENNERS, B., **The ServiceUI API Specification,** Artima Software, Inc., Version 1.1, 2003, Disponível em: <http://www.jini.org/standards/ServiceUI/ServiceUISpec.html>, Acesso em: 23 mai. 2003.

VISUAL Paradigm, **Visual Paradigm for UML**, Disponível em: <http://www.visual-paradigm.com/html/pdf/vpuml_brochure_fine.pdf>, Acesso em: 24 mai. 2003.

TUPLESPACE, Portland Pattern Repository, Disponível em: <http://c2.com/cgi/wiki?TupleSpace>, Acesso em: 03 jul. 2003.

W3C, **Extensible Markup Language (XML) 1.0 Second Edition**, World Wide Web Consortium, october 6, 2000, Disponível em: <http://www.w3.org/TR/2000/REC-xml-20001006.html>, Acesso em: 18 mai. 2003.

ANEXOS

OPERATIONAL STRING DO Remote MDR

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE opstring SYSTEM "java://org/jini/rio/dtd/rio_opstring.dtd" [
        <!ENTITY Local.IP SYSTEM "java://java.net.InetAddress.getLocalHost().
getHostAddress()" >
        <!ENTITY Local.Port "9000" >
        <!ENTITY CodeServerURL "http://&Local.IP;:&Local.Port;/" >
]>
<opstring>
  <OperationalString Name="Remote MDR">
    <ServiceBean Name="Remote MDR" MatchOnName="yes">
      <Interfaces>
        <Interface>br.uem.din.gessd.rmdr.rio.RemoteMdrService</Interface>
      </Interfaces>
      <ExportURLs>
        <ExportURL>&CodeServerURL;rmdr.jar</ExportURL>
        <ExportURL>&CodeServerURL;rio-dl.jar</ExportURL>
      </ExportURLs>
      <Component>br.uem.din.gessd.rmdr.rio.RemoteMdrServiceImpl</Component>
      <ComponentURLs>
        <ComponentURL>&CodeServerURL;rmdr.jar</ComponentURL>
        <ComponentURL>&CodeServerURL;rio-core.jar</ComponentURL>
      </ComponentURLs>
      <Qos>
       <QosSpecification>&DefaultQosSpec;</QosSpecification>
      </Qos>
      <Groups>
        <Group>all</Group>
```

```
      </Groups>
      <Maintain>1</Maintain>
      <Comment>Remote MDR</Comment>
    </ServiceBean>
    <Include>&CodeServerURL;rioCybernodes.xml</Include>
    <Include>&CodeServerURL;javaspace.xml</Include>
  </OperationalString>
</opstring>
```

OPERATIONAL STRING DO MDRExplorer

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE opstring SYSTEM "java://org/jini/rio/dtd/rio_opstring.dtd" [
        <!ENTITY Local.IP SYSTEM "java://java.net.InetAddress.getLocalHost().
getHostAddress()" >
        <!ENTITY Local.Port "9000" >
        <!ENTITY CodeServerURL "http://&Local.IP;:&Local.Port;/" >
]>
<opstring>
  <OperationalString Name="MDR Explorer">

    <ServiceBean Name="MDR Explorer" MatchOnName="yes">
      <Interfaces>
        <Interface>br.uem.din.gessd.rmdr.rio.MdrClientService</Interface>
      </Interfaces>
      <ExportURLs>
        <ExportURL>&CodeServerURL;rmdr.jar</ExportURL>
        <ExportURL>&CodeServerURL;rio-dl.jar</ExportURL>
      </ExportURLs>
      <Component>br.uem.din.gessd.rmdr.rio.MdrClientServiceImpl</Component>
      <ComponentURLs>
```

```
        <ComponentURL>&CodeServerURL;rmdr.jar</ComponentURL>
        <ComponentURL>&CodeServerURL;rio-core.jar</ComponentURL>
      </ComponentURLs>
      <Qos>
       <QosSpecification>&DefaultQosSpec;</QosSpecification>
      </Qos>
      <Groups>
        <Group>all</Group>
      </Groups>
      <Maintain>1</Maintain>
      <Comment>MDR Explorer</Comment>
    </ServiceBean>
    <Include>&CodeServerURL;remoteMdr.xml</Include>
  </OperationalString>
</opstring>
```