FÁBIO SILVA CARVALHO

Integração entre Sistema Multi-Agentes e Sistemas de Banco de Dados Distribuídos

São Paulo
2008

FÁBIO SILVA CARVALHO

Integração entre Sistema Multi-Agentes e Sistemas de Banco de Dados Distribuídos

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia Elétrica.

São Paulo
2008

FÁBIO SILVA CARVALHO

Integração entre Sistema Multi-Agentes e Sistemas de Banco de Dados Distribuídos

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia Elétrica.

Área de Concentração:
Sistemas Digitais

Orientadora:
Profª. Dr.ª Liria Matsumoto Sato

São Paulo
2008

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 25 de julho de 2008.

Assinatura do autor ____________________________

Assinatura do orientador ________________________

FICHA CATALOGRÁFICA

Carvalho, Fábio Silva

Integração entre sistema multi-agentes e sistemas de banco

de dados distribuídos / F.S.Carvalho. -- ed.rev. -- São Paulo, 2008.

p. 142

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Sistemas multiagentes 2.Banco de dados distribuídos 3.Controle de concorrência 4. Sistemas distribuídos I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

DEDICATÓRIA

Dedico este trabalho a mim mesmo, pois só eu sei
o quanto foi difícil criar a oportunidade de tê-lo,
conduzi-lo e finalmente concluí-lo.
Eu valorizo muito todo o meu esforço.

AGRADECIMENTOS

Agradeço primeiramente a Deus, pois devo minha vida a Ele e o faço com muita felicidade. Agradeço a minha família pelo apoio, confiança, incentivo e principalmente pela educação e o amor que têm me dado até hoje. Agradeço a mim mesmo, pois valorizo sempre o meu trabalho e mereço esse agradecimento. Agradeço a cada amigo que de alguma forma me incentivou em qualquer momento desta jornada. Agradeço a minha orientadora professora Liria que sempre demonstra muito amor e dedicação em seu trabalho. Agradeço finalmente a todos os amigos do laboratório LAHPC da Escola Politécnica da USP pela sua contribuição indireta durante a elaboração deste trabalho.

RESUMO

Sistemas multi-agentes devem oferecer recursos suficientes para que seus agentes possam interagir de maneira satisfatória e atingir seus objetivos. Um exemplo de recurso é um conjunto de dados armazenados em algum tipo de mecanismo de persistência, como um sistema gerenciador de banco de dados. O acesso a dados deve ser possível mesmo que eles estejam distribuídos, fato inclusive que também caracteriza os sistemas multi-agentes. Assim, este trabalho apresenta um sistema chamado DASE cujo objetivo é prover a agentes o acesso a dados distribuídos de forma simples e transparente, ou seja, independentemente da complexidade que o ambiente dos agentes possui e das peculiaridades do Sistema de Banco de Dados Distribuído. O DASE suporta qualquer Sistema Gerenciador de Banco de Dados, seja ele centralizado ou distribuído, desde que o mesmo esteja em conformidade com o JDBC. Além disso, oferece recursos importantes como controle de concorrência, suporte a ambientes de dados simultâneos e uso de sentenças de acesso a dados pré-definidas e parametrizadas. Todos os aspectos mais importantes analisados durante o projeto deste sistema estão descritos neste trabalho, evidenciando e justificando o porquê de cada decisão que certamente refletiram no funcionamento e comportamento do DASE. O sistema foi implementado de acordo com o seu projeto, resultando em uma versão funcional e estável, o que foi comprovado através de seu uso em um projeto que envolvia sistemas multi-agentes e controle de tráfego aéreo. Além disso, alguns testes de análise de desempenho considerando cenários variados foram realizados.

Palavras-chave: Sistemas multi-agentes. Banco de dados distribuído. Controle de concorrência. Sistemas distribuídos.

ABSTRACT

Multi-agent systems must offer the needed resources to allow their agents to interact and to reach their goals. An example of resource is a set of data stored in any kind of resource manager, such as a database management system. Data access must be possible even if the data is distributed, characteristic that is also present in multi-agent systems. Thus, this work describes a system whose objective is to provide to agents distributed data access in a simple and transparent way, in other words, hiding the agent environment and complexities related to distributed database systems. DASE supports any database management system, centralized or distributed, in compliance with JDBC (Java Database Connectivity). In addition it offers important features, such as concurrency control, simultaneous data environments and stored SQL sentences. All challenges and important aspects overcome in order to design and implement DASE are described, explaining and justifying every decision that in some way had a participation to form DASE set of functions and behavior. The system was implemented following its design, resulting in a functional and stable version, what could be verified through its adoption in a project based on multi-agent systems and air traffic control systems. In addition, a plenty of performance tests were done regarding different scenarios.

Keywords: Multi-agent systems. Distributed data base. Concurrency control. Distributed systems.

LISTA DE ILUSTRAÇÕES

Figura 1 – Organização das especificações FIPA 20

Figura 2 – Modelo de referência de gerência de agentes proposto pela FIPA 23

Figura 3 – Ciclo de vida de um agente FIPA 26

Figura 4 – Protocolo de interação entre agentes FIPA Request 30

Figura 5 – Protocolo de interação entre agentes FIPA Contract Net.......................32

Figura 6 – Protocolo de interação entre agentes FIPA Propose 33

Figura 7 – Protocolo de interação entre agentes FIPA Brokering 35

Figura 8 – Protocolo de interação entre agentes FIPA Recruiting 36

Figura 9 – Uma transação de transferência de *v* de *x* para *y* e sua abstração 38

Figura 10 – Estados de uma transação 38

Figura 11 – Arquitetura genérica de um SGBD 42

Figura 12 – Arquitetura de um SGBD distribuído 43

Figura 13 – Número de bloqueios de uma transação ao longo do tempo 47

Figura 14 – Arquitetura do JADE 56

Figura 15 – Arquitetura do SisBDPar 59

Figura 16 – Interação entre o DASE e os três sistemas 64

Figura 17 – Relacionamento entre agentes DASE 71

Figura 18 – Serviços prestados entre agentes DASE
e entre agentes DASE e agentes clientes 79

Figura 19 – Arquitetura DASE com dois nós 81

Figura 20 – Exemplo de uma requisição de acesso a dados...................................85

Figura 21 – Arquitetura do controle de concorrência do DASE 91

Figura 22 – Início transparente do sistema 106

Figura 23 – Criação de ambiente de dados a partir de arquivo XML......................108

Figura 24 – Exemplo de um arquivo XML
que representa um descritor de ambiente de dados 109

Figura 25 – Exemplo de agente cliente obtendo o AID do agente
DASEServiceLocator..113

Figura 26 – Exemplo de objeto *ServiceDescriptor*...114

Figura 27 – Envio de mensagem ACL com *ServiceDescriptor*
ao agente *DASEServiceLocator* e recebimento de objetos *MessageTemplate*.....115

Figura 28 – Recebimento do resultado de uma requisição de acesso a dados.....116

Figura 29 – Sentença pré-definida *tx1* 121

Figura 30 – Escalonamento das transações A e B 122

Figura 31 – Diagrama de contexto 131

Figura 32 – Diagrama de casos de uso 134

LISTA DE TABELAS

Tabela 1 – Nomenclatura e multiplicidade dos agentes DASE................................72

Tabela 2 – Serviços prestados pelos agentes DASE..75

Tabela 3 – Detalhes sobre os serviços prestados pelos agentes DASE................76

Tabela 4 – Descrição dos serviços prestados pelos agentes DASE.......................77

Tabela 5 – Explicação das setas da figura 21..82

Tabela 6 – Compatibilidade entre modos de controle de concorrência
e modos de controle de recuperação...96

Tabela 7 – *Tags* de um arquivo XML descritor de ambiente de dados...................111

Tabela 8 – Testes elaborados para o DASE...117

Tabela 9 – Tempos referentes ao teste 1...119

Tabela 10 – Tempos referentes ao teste 2...119

Tabela 11 – Lista de eventos..131

Tabela 12 – Caso de uso
"Criação e configuração de um novo ambiente de dados"..135

Tabela 13 – Caso de uso "Inicio de um ou mais ambientes de dados"...................136

Tabela 14 – Caso de uso "Salvar um ou mais ambientes de dados".....................137

Tabela 15 – Caso de uso “Requisição de dados” 138

Tabela 16 – Caso de uso “Interrupção de um ou mais ambiente de dados” 139

Tabela 17 – Caso de uso “Selecionar ambiente de dados”......................................140

Tabela 18 – Caso de uso “Publicar serviços DASE” 141

Tabela 19 – Caso de uso “Encontrar serviços DASE” 141

Tabela 20 – Caso de uso “Cancelar publicação de serviços DASE”......................142

LISTA DE ABREVIATURAS E SIGLAS

2PL	Two-Phase Locking
ACC	Agent Communication Channel
ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management System
API	Application Programming Interface
BDI	Belief-Desire-Intention
CC	Concurrency Controller
CFP	Call for Proposal
CORBA	Common Object Request Broker Architecture
CP	Connection Provider
DASE	Distributed data Agent Service Environment
DB	Data Bridge
DC	DASE Configurator
DE	DASE Environment

DF	Directory Facilitator
DM	DASE Manager
DP	DASE Data Provider
DX	DASE Proxy
DSM	Distributed Shared Memory
FIPA	Foundation for Intelligent Physical Agents
HAP	Home Agent Platform
IIOP	Internet Inter-ORB Protocol
JADE	Java Agent Development Environment
JDBC	Java Database Connectivity
JNI	Java Native Interface
JVM	Java Virtual Machine
LGPL	GNU Lesser General Public License
MTS	Message Transport Service
Nd	DASE Node
NPFS	Network Parallel File System
ORB	Object Request Broker
RMI	Remote Method Invocation

SBD	Sistema de Banco de Dados
SGBD	Sistema Gerenciador de Banco de Dados
SL	Service Locator
SQL	Structured Query Language
UML	Unified Modeling Language
XML	eXtended Markup Language

Sumário

1 INTRODUÇÃO 17

1.1. Motivação 17

1.2. Objetivo 18

1.3. Organização do Texto 19

2 SISTEMAS MULTI-AGENTES E O PADRÃO FIPA 20

2.1. Modelo de Referência de Gerência de Agentes 21

2.2. DF (*Directory Facilitator*) 23

2.3. AMS (*Agent Management System*) 24

2.4. Ciclo de Vida de um Agente 25

2.5. Protocolos de Interação entre Agentes 28

3 CONTROLE DE CONCORRÊNCIA 37

3.1. Transações 37

3.2. Escalonamento de Transações 39

3.3. Propriedades Básicas das Transações 39

3.4. Arquitetura Típica de um SGBD 41

3.5. Controle de Concorrência Distribuído 43

3.6. Método de Controle de Concorrência 2PL 44

3.7. Método de Controle de Concorrência *Timestamps* 49

4 SISTEMAS UTILIZADOS 53

4.1. JADE 53

4.2. SisBDPar 57

5 DASE: Proposta de um ambiente de serviços de acesso a dados distribuídos a agentes 60

5.1. Plataforma de Agentes Adotada 60

5.2. Principais Características 62

6 ARQUITETURA DO DASE 64

6.1. Agentes DASE 65

6.2. Serviços Prestados pelos Agentes DASE 74

6.3. Balanceamento de Carga 79

6.4. Controle de Concorrência 83

6.5. Controle de Recuperação 93

7 ASPECTOS DE PROJETO E IMPLEMENTAÇÃO 97

7.1. Critério Utilizado para Definir Agentes 97

7.2. Agentes *DASEProxy* 99

7.3. Interação entre o *DASEManager* e outros agentes 100

7.4. Resultado de Requisição 102

8 ASPECTOS FUNCIONAIS 105

8.1. Início e Encerramento do Sistema 105

8.2. Criação e Configuração de um Ambiente de Dados 107

8.3. Requisição de Dados 112

9 RESULTADOS 117

9.1. Resultado do Teste 1 118

9.2. Resultado do Teste 2 119

9.3. Resultado do Teste 3 119

9.4. Projeto GIGA-CDM 123

10 CONCLUSÃO 124

10.1. Conclusões 124

10.2. Trabalhos Futuros 125

REFERÊNCIAS BIBLIOGRÁFICAS 127

APÊNDICE A – DIAGRAMA DE CONTEXTO E LISTA DE EVENTOS 131

APÊNDICE B – CASOS DE USO 133

1 INTRODUÇÃO

Este trabalho apresenta e justifica a necessidade de integração entre sistemas multi-agentes e sistemas de banco de dados distribuídos, além de propor um sistema que viabilize e facilite tal integração. Tal sistema apresenta como principal característica um mecanismo de controle de concorrência, garantindo que os dados armazenados no sistema de banco de dados distribuídos estejam sempre consistentes.

1.1 Motivação

Agentes são entidades de software bem definidas que possuem variado nível de autonomia e pró-atividade. Além dos próprios agentes, outros elementos compõem os sistemas multi-agentes: o ambiente que os mantém, uma coleção de objetos que interagem com eles e um conjunto de regras que regulamenta todas as suas atividades. Os agentes, apesar de possuírem uma percepção limitada do ambiente que os cerca, devem ter acesso aos recursos necessários para atingirem seus objetivos. Tais recursos são oferecidos por outros agentes ou pelo próprio ambiente (Aldo e Giovanni, 2003).

Sistemas multi-agentes são utilizados em diversos domínios de aplicações diferentes, principalmente os relacionados à solução de problemas complexos e de natureza distribuída, envolvendo dados, controle ou conhecimento distribuídos (Oliveira, Fischer *et al.*, 1999). Alguns exemplos de aplicações são sistemas de controle de tráfego aéreo (Almeida Junior, Sato *et al.*, 2005), simulações de sistemas financeiros, gerenciamento de comunicações de redes (Kinny e George, 1996), comércio eletrônico (Oliveira, Fischer *et al.*, 1999), sistemas hospitalares (Moreno, Aïdavalls *et al.*, 2001) e simulações de sistemas naturais e sociais complexos (Conte, Gilbert *et al.*, 1998).

Em todos os exemplos citados, os agentes precisam resolver problemas, interagir com o ambiente e comunicar-se com outros agentes e/ou usuários, atividades que envolvem a necessidade de representar o conhecimento sobre o ambiente externo. Além disso, o conjunto de dados que compõem a base deste

conhecimento é descentralizado (Sycara, 1998), o que traz à integração entre sistemas multi-agentes e sistemas de banco de dados distribuídos uma complexidade adicional, o controle de concorrência durante o acesso aos dados distribuídos.

Assim, o acesso a dados armazenados em um ambiente distribuído é um requisito importante para qualquer sistema multi-agente. Entretanto, é fundamental que tal recurso possa ser utilizado de maneira padronizada e eficiente, assim como todo serviço oferecido dentro de uma comunidade de agentes.

1.2 Objetivo

O objetivo deste trabalho é propor um sistema, chamado DASE, *Distributed data Agent Service Environment*, que seja capaz de promover a integração entre um sistema multi-agente e um ou mais sistemas de banco de dados (SBD) distribuídos.

Para que tal integração seja flexível, manutenível e para que confira qualidade, é fundamental que seja capaz de garantir simplicidade e transparência ao acesso aos dados distribuídos. O benefício direto desta integração é a redução considerável da complexidade durante requisições de acesso dos agentes aos dados armazenados em de sistemas de banco de dados distribuído.

O DASE foi projetado de acordo com o paradigma de agentes, o que significa que os componentes que formam sua arquitetura e garantem sua funcionalidade são dispostos através de serviços prestados por agentes. Esta característica é fundamental para facilitar a integração proposta neste trabalho.

O DASE oferece recursos adicionais relacionados ao acesso aos dados, como garantia de consistência de dados através de controle de concorrência, balanceamento de carga de requisições de acesso aos dados distribuídos, e a possibilidade de uso simultâneo de mais de um sistema de banco de dados distribuído.

Portanto, os dois principais objetivos do sistema proposto neste trabalho são:

1. Prover a agentes de um sistema multi-agentes acesso a dados distribuídos de forma simples e transparente.

2. Oferecer recursos adicionais relacionados ao acesso aos dados, como, por exemplo, a garantia de consistência de dados através de controle de concorrência.

1.3 Organização do Texto

Esta dissertação está organizada da seguinte maneira: o segundo e o terceiro capítulo trazem uma revisão acerca dos principais conceitos utilizados durante o desenvolvimento deste trabalho, sistemas multi-agentes e controle de concorrência. O quarto capítulo descreve cada um dos sistemas ou padrões que se relacionam com o DASE, além de explicar o porquê de sua adoção e qual a sua função. No quinto capítulo o DASE é especificado detalhadamente e suas funcionalidades são descritas.

O sexto capítulo traz a arquitetura do sistema, representada por um conjunto de agentes, além de descrever de forma detalhada os principais recursos do DASE. No sétimo capítulo alguns aspectos de implementação são discutidos, ilustrando decisões de projeto e como algumas dificuldades foram superadas. O oitavo capítulo descreve aspectos funcionais do sistema através de exemplos de configuração e uso. O nono capítulo traz o resultado de um conjunto de testes realizados sobre o sistema desenvolvido, possibilitando a avaliação tanto de sua funcionalidade quanto de seu desempenho. As considerações finais incluem as conclusões deste trabalho e uma relação de itens importantes para o aperfeiçoamento futuro do sistema aqui proposto.

2 SISTEMAS MUTI-AGENTES E O PADRÃO FIPA

Um sistema multi-agente pode ser definido como uma rede fracamente acoplada de solucionadores de problemas autônomos, que interagem para solucionarem problemas que estão além da capacidade individual ou conhecimento de cada um. Tais solucionadores de problemas são chamados agentes (Sycara, 1998). Os sistemas multi-agentes são responsáveis por garantir que os agentes, executados em um mesmo ambiente, interajam e tenham todas as condições necessárias para atingirem seus objetivos (Weiss, 2000).

FIPA (*Foundation for Intelligent Physical Agents*) é uma organização filiada ao *IEEE Computer Society* cujo objetivo é promover e padronizar a tecnologia orientada a agentes, além de possibilitar a interoperabilidade entre este paradigma e outras tecnologias (Aparicio, Chiariglione *et al.*, 1999).

O padrão FIPA é formado por uma coleção de especificações que promove a interoperabilidade entre agentes heterogêneos e suas atividades. O conjunto completo de especificações é dividido nas seguintes categorias: aplicações, arquitetura abstrata, comunicação entre agentes, gerência de agentes e transporte de mensagens. A comunicação entre agentes é a categoria mais importante deste modelo de arquitetura de sistemas multi-agentes. A figura 1 ilustra a organização das especificações FIPA.

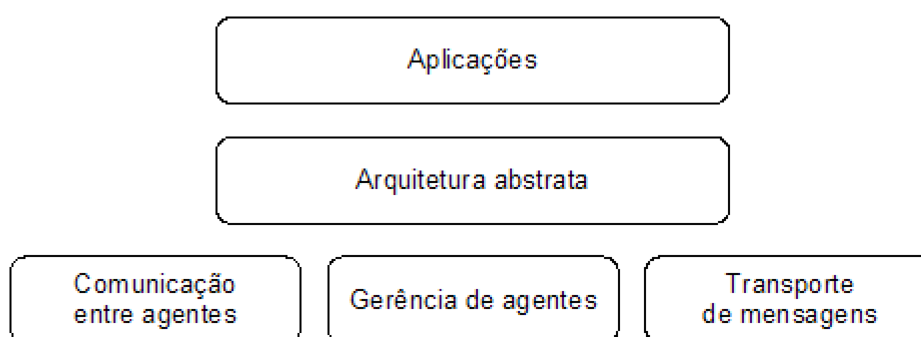

Figura 1 - Organização das especificações FIPA.

As especificações de gerência de agentes lidam com o controle e gerência de agentes em uma plataforma ou entre diferentes plataformas de agentes. Já as especificações de comunicação entre agentes lidam com a estrutura das mensagens utilizadas, protocolos de interação entre agentes e representação do conteúdo das mensagens. As especificações de transporte de mensagens lidam com o transporte e representação de mensagens através de protocolos de rede diferentes, incluindo ambientes com ou sem fio.

Segundo a FIPA, agente é um processo computacional que implementa a funcionalidade comunicativa autônoma de uma aplicação. Um agente deve possuir no mínimo um proprietário, como por exemplo, uma organização ou um usuário humano, e deve possuir um conceito de identidade chamado AID (*Agent Identifier*) que o rotula, permitindo que o mesmo possa distinguir-se dos demais agentes, além de permitir a definição de remetente e destinatários em mensagens trocadas entre eles. Os agentes comunicam-se fazendo uso de uma linguagem chamada ACL (*Agent Communication Language*).

2.1 Modelo de Referência de Gerência de Agentes

A FIPA define um modelo de referência lógico para a criação, registro, localização, comunicação, migração e desativação de agentes, representando um ambiente em que eles possam existir e atuar (FIPA Agent Management Specification, 2004). Este modelo consiste dos seguintes componentes lógicos, cada um representando um conjunto de capacidades:

- **Software**: representa todas as coleções de instruções executáveis acessíveis através de um ou mais agentes.
- **Agente**: elemento fundamental em uma plataforma de agentes que combina um ou mais serviços, publicados em um repositório, e um modelo de execução unificado e integrado.
- **AMS** (*Agent Management System*): componente obrigatório em uma plataforma de agentes responsável pelo controle e supervisão do uso e acesso a plataforma de agentes. Existe somente um AMS em uma plataforma de agentes. O AMS mantém um diretório de AIDs que

contém, entre outras informações, endereços de transporte dos agentes registrados na plataforma. O AMS oferece o serviço[1] de "páginas brancas" aos agentes, o que significa que cada agente deve registrar-se através do AMS para obter um AID válido e passar a atuar no sistema.

- **DF** (*Directory Facilitator*): componente opcional da plataforma de agentes cuja função é prover o serviço de "páginas amarelas[2]", permitindo aos agentes registrarem seus serviços, ou encontrar serviços disponibilizados por outros agentes. Podem existir múltiplos DFs em uma plataforma de agentes.
- **MTS** (*Message Transport Service*): elemento que permite a comunicação entre agentes de plataformas diferentes. O MTS também é denominado como ACC (*Agent Communication Channel*).
- **Plataforma de agentes**: provê a infra-estrutura física em que os agentes são executados, representando os computadores, sistemas operacionais, software que oferece recursos aos agentes, componentes de controle de agentes FIPA (DF, AMS e MTS), além dos próprios agentes.

A figura 2 ilustra o modelo de referência de gerência de agentes, proposto pela FIPA, por meio da interação entre duas plataformas diferentes. É importante destacar que uma plataforma de agentes não é restrita a um único *host*[3], podendo ser distribuída ao longo de diversos computadores.

[1] O termo "serviço" se refere exclusivamente a um conjunto de tarefas de um agente cuja correta execução, e eventuais resultados gerados, interessam a outro(s) agente(s).
[2] Os termos "páginas brancas" e "páginas amarelas" fazem alusão ao modo em que é organizada uma lista telefônica, em que as páginas brancas contêm informações de pessoas, enquanto que as páginas amarelas contêm informações sobre serviços prestados pelos assinantes.
[3] O termo *host* refere-se a um dispositivo computacional capaz de responder ou agir de forma participativa em um ambiente de rede de computadores. Esta palavra não foi traduzida em razão de não haver um termo correspondente em português que confira o mesmo significado. Este termo sempre aparecerá neste trabalho em itálico.

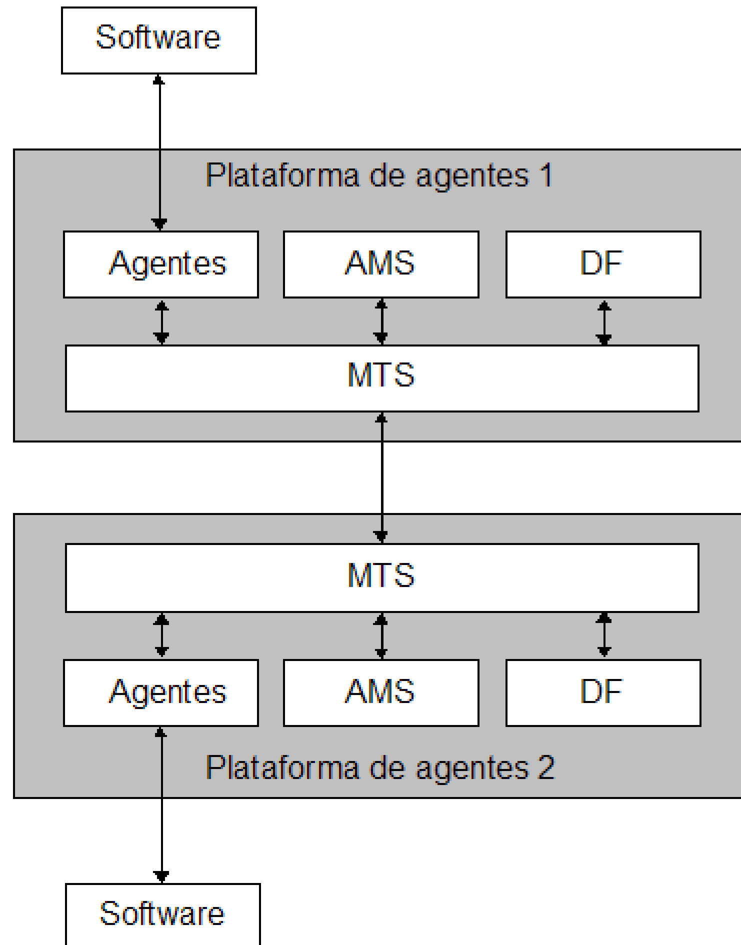

Figura 2 - Modelo de referência de gerência de agentes proposto pela FIPA.

2.2 DF (Directory Facilitator)

Embora seja opcional em uma plataforma de agentes, o DF é o componente responsável por prover aos agentes o importante serviço de diretório de “páginas amarelas”, o que significa a publicação e a localização de serviços prestados por agentes (FIPA Agent Management Specification, 2004). Cada DF é capaz de executar as seguintes funções:

- Registro de informações sobre um agente e os serviços que deseja publicar.
- O cancelamento total ou parcial do registro de um agente e seus serviços.
- A modificação do registro de um agente e seus serviços.

- A busca de um ou mais agentes que prestam um serviço específico.

Cada agente que desejar publicar seus serviços a outros agentes deve solicitar a um DF que registre a sua descrição e a de seus serviços. Entretanto, não há qualquer garantia por parte do DF de que o serviço registrado existirá de fato, ou então que será prestado exatamente de acordo com o que foi publicado. Isto se deve à autonomia e pró-atividade que cada agente possui, permitindo inclusive que um agente recuse uma solicitação de prestação de um serviço previamente publicado através de um DF. Portanto, além de não garantir a prestação dos serviços registrados, todo DF também não pode assegurar que a informação utilizada em cada registro é verdadeira e atual.

A qualquer momento, e por qualquer motivo, um agente pode solicitar ao DF que modifique ou exclua a publicação de seu serviço. Um agente pode realizar uma busca em um DF para encontrar um ou mais agentes que possam prestar os serviços que lhe interessam.

Dependendo da implementação do padrão FIPA, um DF ainda pode oferecer um mecanismo de inscrição de acordo com o protocolo de interação entre agentes *FIPA Subscribe* (FIPA Subscribe Interaction Protocol Specification, 2002). Tal funcionalidade permite aos agentes interessados em um determinado serviço informarem ao DF que desejam ser notificados a respeito de qualquer operação que ocorra no registro deste serviço, ou do agente que o publicou. Essas operações incluem registro, cancelamento de registro, ou modificação da descrição de um agente e seus serviços. Assim, sempre que o registro de um determinado agente ou de um de seus serviços for alterado, todos os agentes interessados e inscritos serão avisados pelo DF.

2.3 AMS (Agent Management System)

Este componente é responsável por gerenciar a plataforma de agentes controlando a criação e a exclusão de agentes, além da migração, caso a plataforma ofereça mobilidade de agentes. O AMS ainda gerencia os recursos da plataforma de agentes e disponibiliza informações sobre eles, além de manter o ciclo de vida de cada agente da plataforma (FIPA Agent Management Specification, 2004).

Existe sempre exatamente um AMS em cada plataforma de agentes, mesmo que a plataforma esteja distribuída ao longo de diversos *hosts*. O AMS tem o poder de solicitar que um determinado agente execute uma operação de controle específica, como o encerramento de sua execução na plataforma, e inclusive forçar tal operação, caso o agente em questão se recuse a atender a solicitação.

Além do registro de um agente, o AMS é capaz de executar as funções de busca, cancelamento, ou modificação de registro de um agente. Assim, a descrição de um agente pode ser obtida ou alterada por meio do AMS.

O AMS mantém um índice de todos os agentes presentes na plataforma. Tal índice contém o AID de cada agente. A descrição de um agente pode ser alterada a qualquer momento e por qualquer motivo no AMS, desde que o AMS autorize tal operação. Entretanto, o nome de um agente, que é apenas uma parte de seu AID, nunca pode ser alterado. O nome de um agente é um identificador único e global formado por um apelido seguido do símbolo “@” e o endereço da plataforma onde ele foi criado, ou HAP (*Home Agent Platform Address*). Além do nome do agente, o AID ainda inclui informações como uma lista de endereços para os quais mensagens destinadas ao agente podem ser endereçadas, e uma lista de endereços de serviços de resolução de nomes, prestados pelo AMS através da função *search*.

A vida de um agente em uma plataforma se encerra com o cancelamento do seu registro no AMS. Depois disso, seu AID é removido do diretório.

2.4 Ciclo de Vida de um Agente

Agentes FIPA existem fisicamente em uma plataforma de agentes e utilizam todas as facilidades oferecidas pela plataforma para atingirem seus objetivos. A partir deste contexto e visto como um processo, um agente possui um ciclo de vida, controlado pelo AMS. O ciclo de vida de um agente FIPA possui as seguintes características (FIPA Agent Management Specification, 2004):

- **Limitado a uma plataforma de agentes.** Um agente é fisicamente gerenciado dentro de uma plataforma e, portanto, seu ciclo de vida é sempre limitado a uma plataforma de agentes específica.

- **Independente de aplicação.** O modelo de ciclo de vida define somente os estados e as transições de estados ao longo da vida de um agente, independentemente de qualquer aplicação.
- **Orientado a instância.** O agente descrito no modelo de ciclo de vida é uma instância de um agente, o que significa que seu estado é independente do estado de qualquer outro agente.
- **Único.** Todo agente possui somente um estado de ciclo de vida em qualquer momento, e dentro de somente uma plataforma de agentes.

Abaixo a figura 3 ilustra o ciclo de vida de um agente.

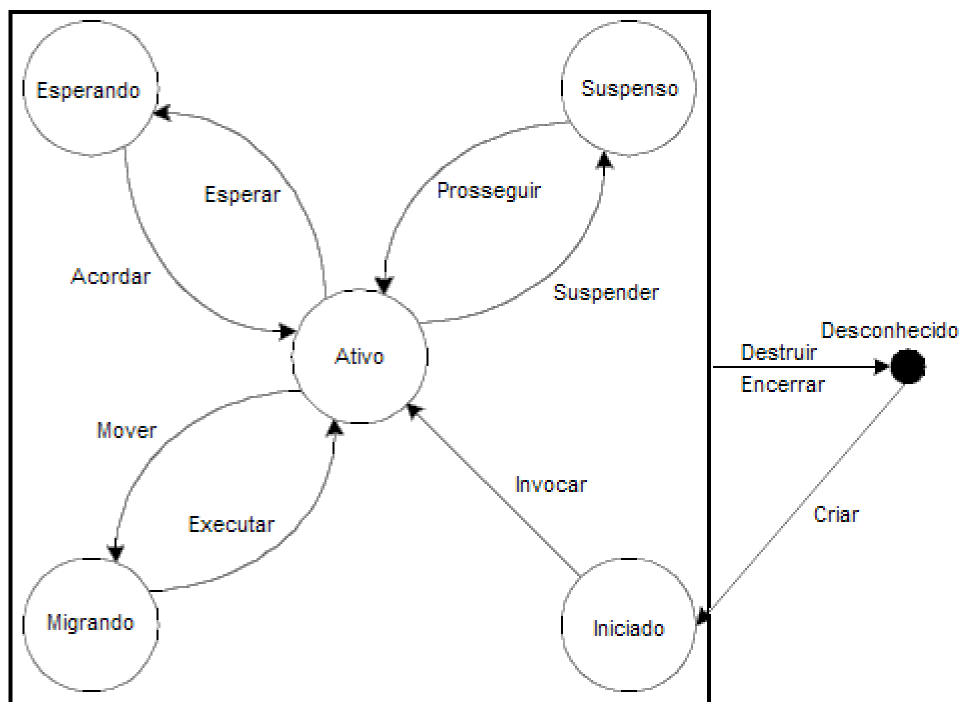

Figura 3 - Ciclo de vida de um agente FIPA.

A seguir são descritos os possíveis estados de um agente:

- **Iniciado**: o agente já foi criado, no entanto ainda não é capaz de atuar, ou seja, interagir com os demais agentes.
- **Ativo**: o agente registra-se junto ao AMS, recebe um AID válido, e torna-se apto a interagir com os demais agentes. Neste estado o agente é capaz de receber e enviar mensagens a qualquer outro agente normalmente.

- **Suspenso**: o agente fica incapaz de executar suas atividades temporariamente. Esta transição pode ser causada pelo próprio agente ou pelo AMS.
- **Esperando**: o próprio agente decide ficar inativo temporariamente enquanto aguarda que um evento externo ocorra.
- **Migrando**: somente agentes móveis podem estar neste estado. Um agente estacionário executa todas as suas instruções no *host* em que foi invocado. Já um agente móvel pode deslocar-se para outro *host* e continuar sua execução em uma máquina diferente.
- **Desconhecido**: este estado identifica um agente inoperante e incapaz de integrar novamente o ambiente multi-agente.
  - A seguir são descritas as transições de estado possíveis de um agente:
- **Criar**: criação ou instalação de um novo agente na plataforma.
- **Invocar**: invocação de um novo agente fazendo com que ele se torne ativo e comece de fato a executar suas tarefas.
- **Destruir**: encerramento forçado de um agente realizado pelo AMS. O agente é incapaz de impedir seu encerramento nesta situação.
- **Encerrar**: encerramento não forçado de um agente. Tal operação pode ser ignorada pelo agente.
- **Suspender**: o agente passa do estado ativo para o estado suspenso.
- **Prosseguir**: retorna o agente do estado suspenso tornando-o ativo novamente. Esta transição também pode ser causada pelo próprio agente ou pelo AMS.
- **Esperar**: faz com que o agente entre em um estado de espera. Esta transição pode ser causada somente pelo próprio agente.
- **Acordar**: retorna o agente do estado esperando tornando-o ativo novamente. Esta transição pode ser causada somente pelo próprio agente.
- **Mover**: faz com que o agente entre em um estado transitório enquanto move-se de uma plataforma a outra. Esta transição pode ser causada somente pelo próprio agente.

- **Executar**: retorna o agente do estado migrando tornando-o ativo novamente. Esta transição pode ser causada somente pelo AMS.

2.5 Protocolos de Interação entre Agentes

O padrão FIPA possui um conjunto de nove protocolos que definem diversos modos possíveis de interação entre agentes através de troca de mensagens ACL. A seguir apenas os protocolos estudados para o desenvolvimento do DASE serão descritos sucintamente. Durante a descrição de cada protocolo, o agente denominado *initiator* será o responsável pelo início da interação, ou comunicação. O agente a quem se dirige o *initiator* é chamado de *participant*. Em muitas situações o *initiator* é um agente que deseja um serviço, enquanto que o *participant* é um agente candidato a oferecer este serviço. Em alguns casos pode existir mais de um *participant*.

Cada tipo de protocolo é identificado através do parâmetro *protocol* de uma mensagem ACL. Cada interação utilizando um protocolo é identificada através de um parâmetro de mensagem ACL único, não nulo e global, chamado *conversation-id*. Este parâmetro é definido pelo *initiator* e é utilizado em todas as mensagens trocadas durante a comunicação.

Os protocolos são ilustrados através de um tipo de diagrama, semelhante a um diagrama de seqüência UML, proposto em (Odell, Parunak *et al.*, 2001):

2.5.1 FIPA Request

Neste protocolo de interação, o *initiator* necessita de um serviço e sabe a qual agente solicitar. Tanto o *initiator* quanto o *participant* conhecem todas as pré-condições para a prestação do serviço, por isso nesta situação não há negociação, há simplesmente a solicitação (*request*) seguida da prestação do serviço.

Entretanto este protocolo está longe de ser semelhante a um modelo cliente-servidor, pois a prestação do serviço pode não ocorrer devido ao *participant*:

- Simplesmente ignorar a solicitação, já que é autônomo como todo agente.

- Não entender o que foi solicitado, podendo pedir que o *initiator* repita a solicitação.
- Recusar-se (*refuse*) a prestar o serviço, informando o motivo da recusa.

Ainda assim, o *participant* pode ter a intenção de prestar o serviço, mas não obter êxito devido a uma falha (*failure*). Neste caso ele informa isto ao *initiator*. Caso o *participant* aceite prestar o serviço, ele envia uma mensagem indicando a sua concordância (agree), no entanto esta mensagem é opcional (FIPA Request Interaction Protocol Specification, 2002).

Quando tudo ocorre normalmente e o serviço é prestado, duas respostas são possíveis por parte do *participant*: uma simples mensagem de confirmação, dizendo que o serviço foi realizado como esperado (*inform-done*), ou uma mensagem semelhante à anterior seguida de algum resultado (*inform-result*). A figura 4 ilustra este protocolo.

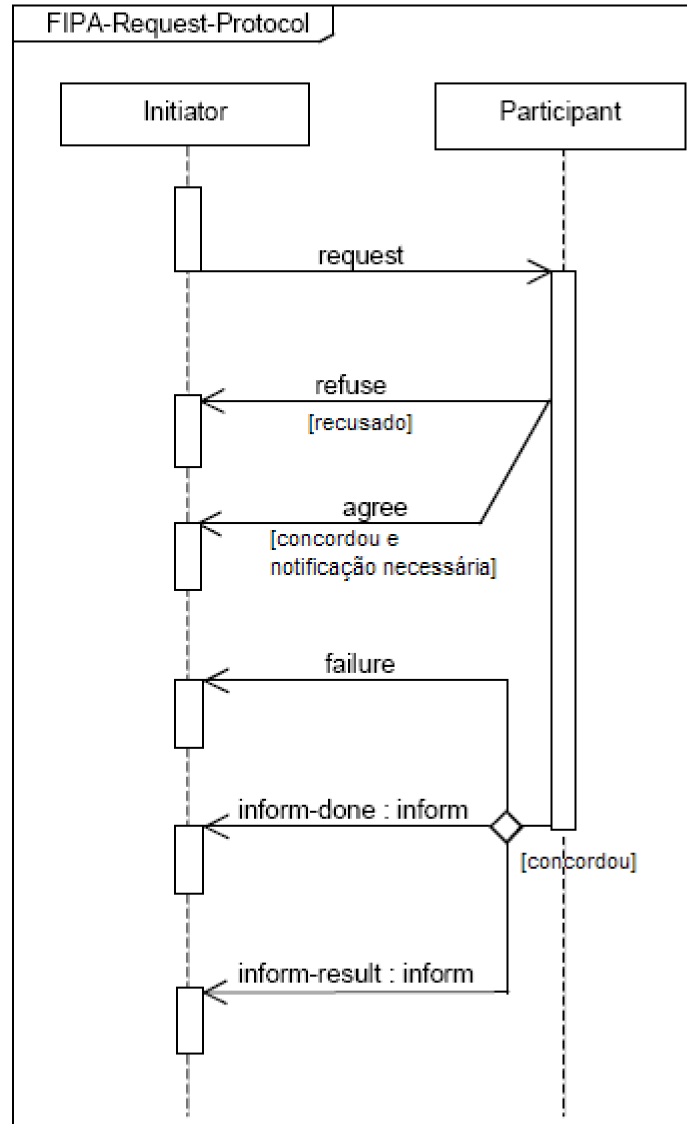

Figura 4 - Protocolo de interação entre agentes FIPA Request.

2.5.2 FIPA Contract Net

Neste protocolo de interação, o *initiator* necessita de um serviço, sabe o que deseja, mas não sabe exatamente quem lhe prestará o serviço. Além disso, ele tem consciência de que pode obter o melhor serviço, se pesquisar e negociar. A figura 5 ilustra este protocolo (FIPA Contract Net Interaction Protocol Specification, 2002).

O *initiator* envia uma mensagem a *m* agentes *participant* solicitando que enviem suas propostas de prestação de serviço. Esta mensagem inicial, chamada de CFP (*call for proposal*), contém todas as especificações e restrições do serviço que o agente cliente deseja obter.

Eventualmente um ou mais agentes *participant* que receberam a mensagem CFP não a respondem. Isto ocorre porque tal agente simplesmente ignorou a mensagem, ou então porque não houve tempo para que a respondesse antes do tempo hábil definido pelo agente *initiator* para o envio das respostas (*deadline*).

Uma quantidade de *j* agentes *participant*, que receberam a mensagem CFP e se "candidataram" a prover o serviço, enviam suas propostas (*propose*) ao *initiator*. Este valor é igual ao total de agentes que responderam à mensagem CFP, *n*, subtraído pelo total de agentes que enviou como resposta *refuse*, recusando-se a enviar uma proposta. Esta quantidade é representada na figura 5 pela letra *i*.

O *initiator*, após receber propostas, escolhe a melhor e informa aos agentes candidatos a sua decisão enviando ao agente cuja proposta foi aceita uma mensagem *accept-proposal*, ou *reject-proposal* aos agentes cujas propostas não foram aceitas. É possível também que o *initiator* deseje utilizar mais de um serviço, assim ele envia mais de uma mensagem *accept-proposal*.

A negociação se encerra quando o *participant* que proverá o serviço recebe uma mensagem *accept-proposal*. Tal agente, então, realiza o que foi acordado e envia uma mensagem de confirmação, dizendo que o serviço foi realizado como esperado (*inform-done*), ou uma mensagem semelhante à anterior seguida de algum resultado (*inform-result*). Caso o agente provedor do serviço não consiga realizar sua tarefa, então ele envia uma mensagem *failure* contendo o motivo de não ter obtido êxito.

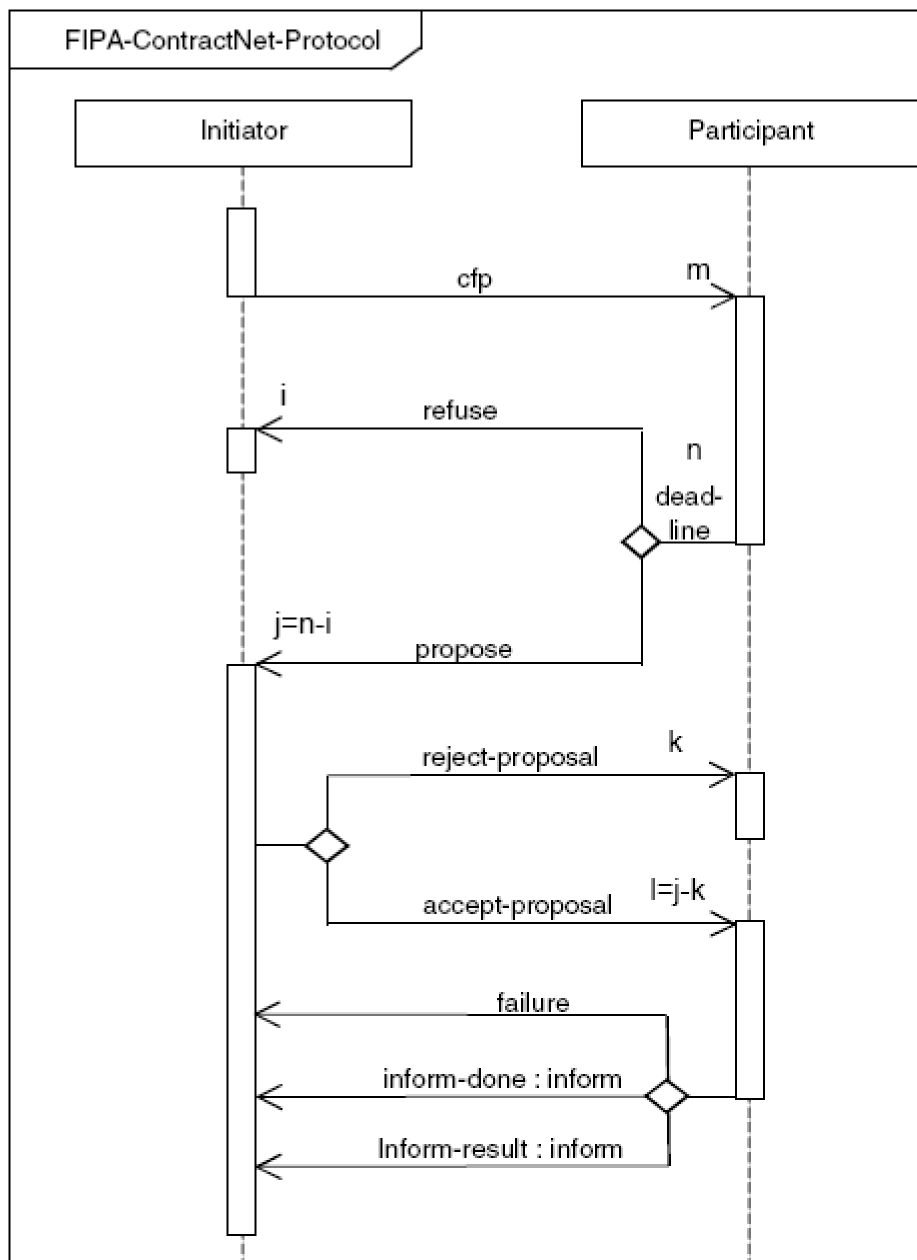

Figura 5 - Protocolo de interação entre agentes FIPA Contract Net.

2.5.3 FIPA Propose

Através deste protocolo de interação, o agente *initiator* envia uma mensagem contendo uma proposta a outro agente, o *participant*. Esta proposta normalmente solicita o consentimento do agente *participant* para que uma determinada tarefa seja executada. Se este concordar, através de uma mensagem *accept-proposal*, o que foi proposto será feito. A proposta pode ser rejeitada através de uma mensagem *reject-proposal*.

Este protocolo é semelhante ao *Contract Net*, no entanto, o serviço é realizado sem que haja a solicitação de um agente através de um CFP. Além disso, o serviço executado não precisa ter efeito direto nos desejos ou necessidades do agente "cliente", caracterizando, portanto um pedido de permissão para a execução de serviço, e não uma oferta de serviço. Um efeito disso é que, diferentemente dos outros protocolos, o agente *initiator* é justamente o que executa o serviço, e não o *participant*. . A figura 6 ilustra este protocolo (FIPA Propose Interaction Protocol Specification, 2002).

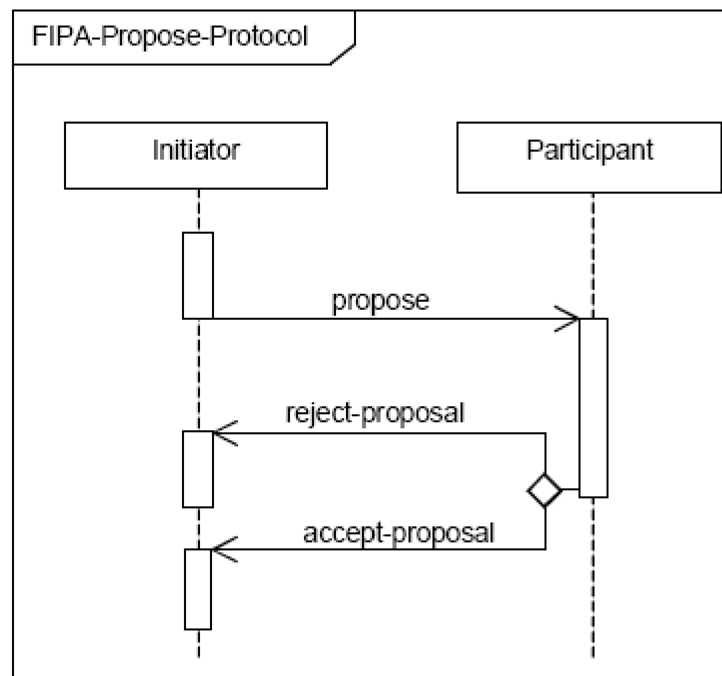

Figura 6 - Protocolo de interação entre agentes FIPA Propose.

2.5.4 FIPA Brokering

Um agente *broker*[4] oferece um conjunto de serviços de facilidade de comunicação a outros agentes através de seu conhecimento sobre requisitos comuns em um problema específico, e sobre a capacidade de alguns agentes em atender a estes requisitos, oferecendo uma solução ao problema. Quando um agente *initiator* solicita

[4] O termo *broker* refere-se a um componente de software responsável por mediar o relacionamento entre um cliente e um servidor, ou entre um repositório e um elemento que solicita acesso a tal repositório. Esta palavra não foi traduzida em razão de não haver um termo correspondente em português que confira o mesmo significado. Este termo sempre aparecerá neste trabalho em itálico.

a um *broker* para localizar um ou mais agentes que possam prestar um determinado serviço, o *broker* determina o conjunto de agentes apropriados para prestar o serviço, envia a solicitação a tais agentes, e em seguida encaminha a resposta ao agente que fez a solicitação.

Tal esquema confere eficiência e transparência de localização durante a prestação do serviço, já que o *initiator* não sabe e nem precisa saber onde estão os agentes que poderiam lhe prestar o serviço. O uso de agentes *broker* pode ainda simplificar significativamente a tarefa de interação entre agentes. Adicionalmente, agentes *broker* também tornam o sistema mais adaptável e robusto em situações dinâmicas, oferecendo inclusive escalabilidade e segurança, assim como qualquer sistema que possua alta coesão e baixo acoplamento oferece.

Este protocolo é chamado de macro protocolo, pois utiliza outro protocolo, chamado, sub-protocolo, para interagir com os agentes que de fato realizaram os serviços solicitados. A figura 7 ilustra este protocolo (FIPA Brokering Interaction Protocol Specification, 2002).

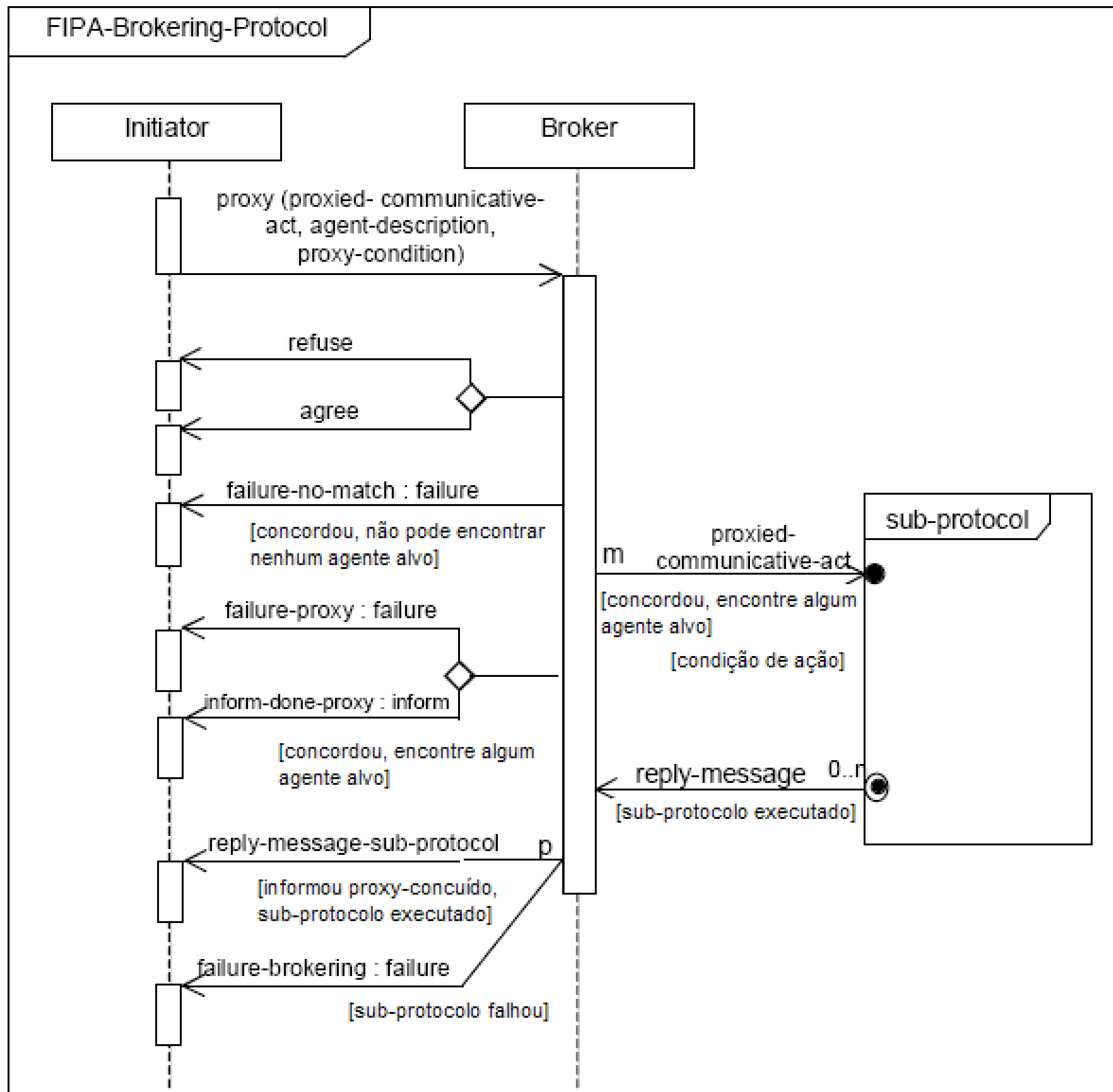

Figura 7 - Protocolo de interação entre agentes FIPA Brokering.

2.5.5 FIPA Recruiting

Este protocolo de interação entre agentes é uma variação do protocolo *Brokering*. Neste caso, ao invés de intermediar a prestação do serviço, o agente *broker*, agora chamado de *recruiter*, irá localizar os agentes que prestarão o serviço, mas não irá encaminhar as respostas dos serviços executados pelos agentes localizados. Tais respostas serão entregues pelo agente que prestou o serviço diretamente ao agente *initiator* ou qualquer outro indicado por ele. A figura 8 ilustra este protocolo (FIPA Recruiting Interaction Protocol Specification, 2002).

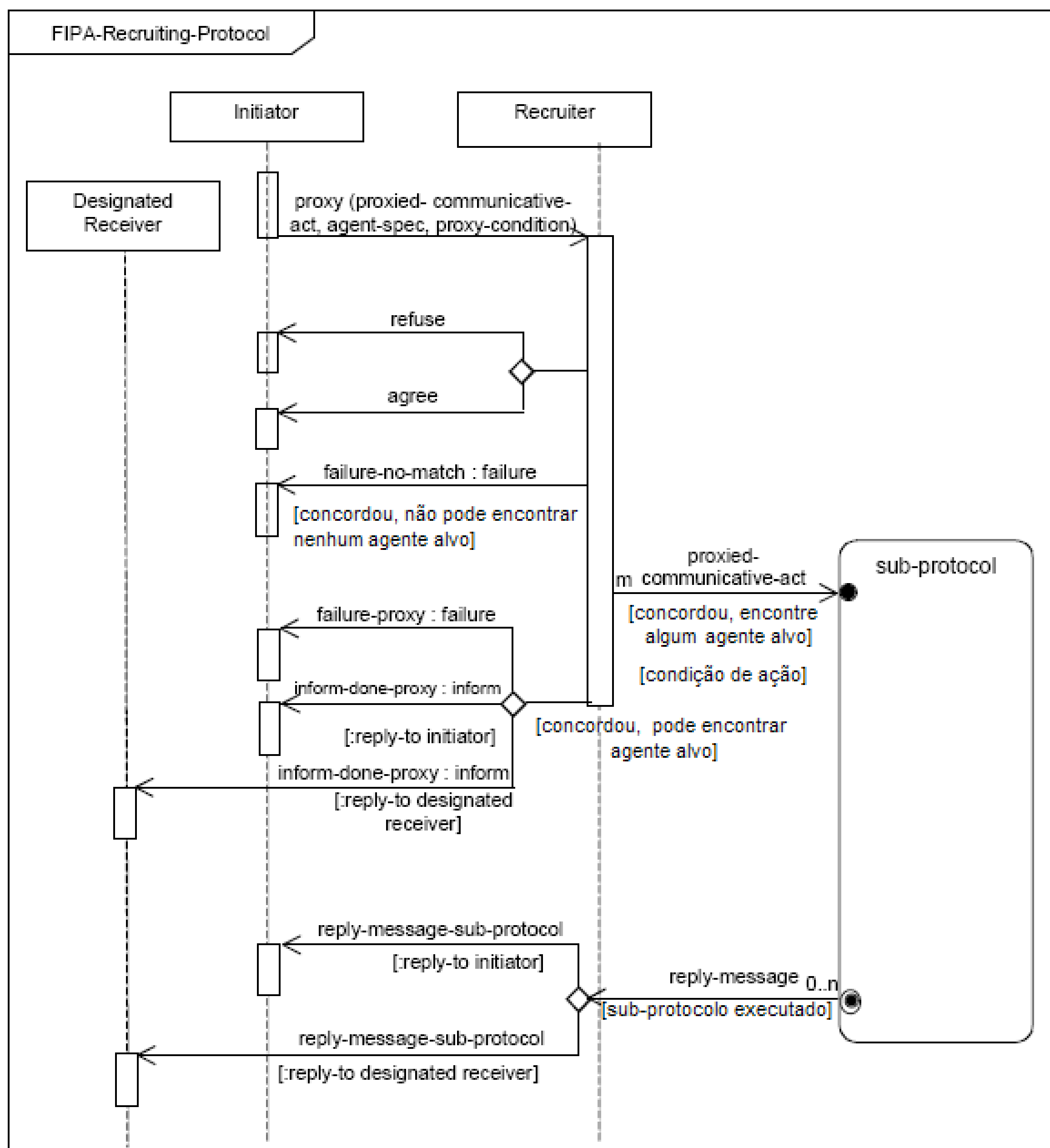

Figura 8 - Protocolo de interação entre agentes FIPA Recruiting.

3 CONTROLE DE CONCORRÊNCIA

O controle de concorrência é um mecanismo que permite a diversos programas acessarem um conjunto de recursos compartilhados de forma transparente, ou seja, sem que um programa precise saber da existência dos outros. Cabe ao controlador de concorrência garantir que cada programa possa ser executado de forma independente de todos os outros. Para que isso seja feito certas restrições precisam ser impostas formando um modelo de comportamento, a ser satisfeito pelos programas, cujo elemento principal é a transação (Ferreira e Finger, 2000). O principal motivo para a necessidade de execução concorrente de várias transações é o ganho em eficiência.

3.1 Transações

Transações são programas, ou conjuntos de instruções de programas, que acessam recursos compartilhados. Uma aplicação pode ser composta de uma ou mais transações. As transações possuem duas características que permitem e facilitam sua funcionalidade, a atomicidade e a abstração. O objetivo do controle de concorrência é garantir que as transações sejam executadas atomicamente, o que quer dizer que uma transação é um programa que ou sucede na íntegra, ou é totalmente desfeito.

O modelo de transações tem a característica de ser abstrato com relação às operações realizadas pelas transações, já que o modelo de transações se preocupa apenas com as operações realmente executadas de leitura e escrita em itens de dados compartilhados, e com os dados envolvidos nas operações, pois o valor lido na operação de leitura de um dado é ignorado. A figura 9 mostra um exemplo de uma transação de transferência de um valor *v* entre as contas *x* e *y*, e sua correspondente abstração contendo as operações de leitura, representada pela letra “r”, escrita, representada pela letra “w” e confirmação, representada pela letra “c”.

Transação	*Abstração*
BEGIN TRANSACTION	
/* Decrementa x de v */	
X := READ(x);	r[x]
X := X – v;	
WRITE(x, X);	w[x]
/* Incrementa y de v */	
Y := READ(y);	r[y]
Y := Y + v;	
WRITE(y, Y);	w[y]
COMMIT	c

Figura 9 - Uma transação de transferência de *v* de *x* para *y* e sua abstração.

Uma transação pode estar confirmada, abortada, ou ativa. Uma transação ativa é uma transação que já foi iniciada, mas que ainda não se confirmou nem abortou. Uma transação confirmada é aquela que obteve êxito durante a execução de todas as suas operações. Transação abortada é toda transação interrompida antes que se confirmasse. Todas as modificações nos dados geradas por operações de uma transação abortada são desfeitas. Os três possíveis estados para uma transação estão ilustrados na figura 10 (Ferreira e Finger, 2000).

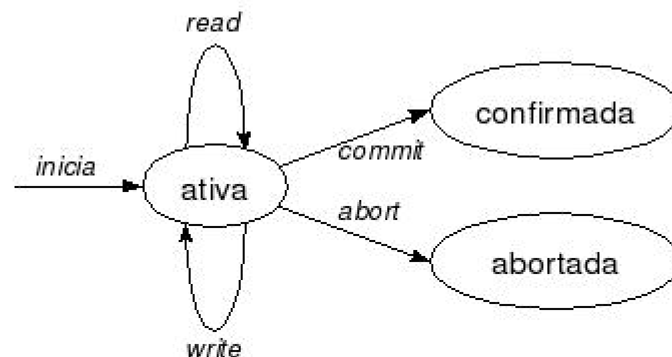

Figura 10 - Estados de uma transação.

3.2 Escalonamento de Transações

Como é interessante que haja execução concorrente de transações, mais de uma transação poderá ser executada ao mesmo tempo. Conseqüentemente, diversas operações de diversas transações também serão executadas ao mesmo tempo. A concorrência das operações de transações concorrentes é modelada através de uma ordem parcial de operações. Portanto, uma ordem (parcial) de operações das diversas transações concorrentes é chamada de escalonamento.

É importante que um escalonamento não permita que operações conflitantes entre as transações sejam concorrentes. Duas operações de transações distintas são chamadas conflitantes se operam sobre o mesmo item de dado e pelo menos uma delas for uma operação de escrita. Um escalonamento sobre o conjunto de transações preserva a ordem interna de cada transação e ordena mais algumas operações entre as transações, incluindo todas as operações conflitantes.

O conceito de escalonamento é independente do método usado para gerá-los. Na prática muitos escalonamentos possíveis são indesejáveis, e o controle de concorrência tem como função garantir que só os escalonamentos desejáveis sejam gerados pelo sistema. Escalonamentos indesejáveis são aqueles que (Ferreira e Finger, 2000):

- Não permitem que transações abortadas sejam desfeitas consistentemente.
- São capazes de forçar que transações confirmadas sejam desfeitas.
- Não permitem que transações sejam executadas concorrentemente de forma independente, causando interferência entre transações.

3.3 Propriedades Básicas das Transações

As propriedades básicas das transações estão definidas em três grupos: as propriedades fundamentais, que independem do método de controle de concorrência, as propriedades dependentes do controle de concorrência de transações, e o critério de correção de escalonamentos, ou seja, a condição para que um escalonamento de transações com operações concorrentes seja aceito como correto.

3.3.1 Propriedades ACID

Essas propriedades, cujas iniciais formam o nome dado a elas, são as propriedades básicas que consagram o modelo de transações:

- **Atomicidade**. Uma transação é uma unidade atômica de processamento que deve ser executada integralmente, ou totalmente desfeita.
- **Consistência**. A execução de uma transação deve levar o banco de dados de um estado consistente a outro estado consistente.
- **Isolamento ou independência**. A execução de uma transação não pode ser afetada por outras transações sendo executadas concorrentemente.
- **Durabilidade ou persistência**. Os efeitos de uma transação confirmada não podem ser desfeitos.

3.3.2 Propriedades dependentes do método de controle de concorrência

As diversas transações sendo processadas em um sistema enviam suas operações (leitura, escrita, confirmação e aborto) em uma ordem qualquer, e o escalonador deve decidir em que ordem estas operações devem ser executadas para garantir as propriedades ACID, além de garantir o critério de correção de escalonamentos.

As propriedades do controle de concorrência envolvem a maneira como o escalonador permite que as transações, e suas operações, sejam organizadas e interajam entre si. Através da atuação do escalonador, o relacionamento entre as transações é estabelecido e a prevenção de eventos indesejáveis que violam as propriedades ACID é garantida.

3.3.3 Critério de Correção de Escalonamentos

O critério de correção de escalonamentos concorrentes deve garantir que a execução concorrente de diversas transações produza um efeito no banco de dados

equivalente ao de alguma execução em série das transações presentes no escalonamento.

A execução serial é aquela em que uma, e só uma transação, é executada de cada vez. Desta forma, considerando três transações *T1*, *T2* e *T3* sendo executadas serialmente, uma execução concorrente correta deve ter os mesmos efeitos que uma das seguintes execuções seriais:

T1; T2; T3	*T2; T1; T3*	*T3; T1; T2*
T1; T3; T2	*T2; T3; T1*	*T3; T2; T1*

Um escalonamento que obedece a este critério de correção é dito seriável.

3.4 Arquitetura Típica de um SGBD

O escalonador é um elemento fundamental na arquitetura da maioria dos sistemas gerenciadores de banco de dados (SGBD). A figura 11 ilustra a arquitetura de um SGBD genérico. Tal arquitetura assume que o sistema é executado em um ambiente centralizado, mas pode ser estendida para um ambiente distribuído. O escalonador é o elemento responsável por evitar a ocorrência de escalonamentos indesejáveis e de garantir as propriedades básicas dos escalonamentos, descritas anteriormente.

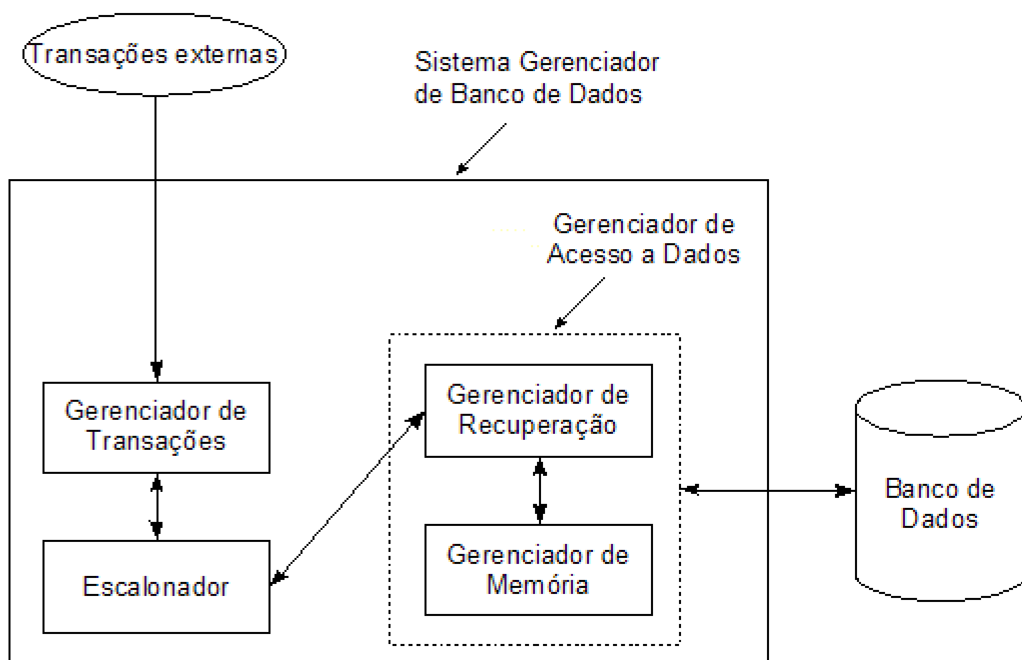

Figura 11 - Arquitetura genérica de um SGBD.

O gerenciador de transações é o responsável pela interface do gerenciador de banco de dados com as aplicações externas ao banco de dados. O gerenciador de transações realiza um pré-processamento das informações recebidas das transações e as envia ao escalonador.

O gerenciador de acesso a dados possui duas partes, o gerenciador de memória e o gerenciador de recuperação. O gerenciador de memória é o responsável pelo controle das estruturas de dados que guardam um espelho em memória principal da parte do conteúdo do banco de dados que está em disco. Já o gerenciador de recuperação é o responsável pelo controle de recuperação do sistema em caso abortos ou falhas. Ele realiza a manutenção do registro de transações, anotando todas as suas operações básicas que forem executadas. Este registro é usado durante o processo de desfazer transações abortadas e durante o procedimento de reinício do sistema após falhas, restaurando-o a um estado consistente.

3.5 Controle de Concorrência Distribuído

O controle de concorrência garante a consistência dos dados mesmo que diversas transações os acessem. Quando considerado um sistema de banco de dados (SBD) distribuído, o controle de concorrência passa a possuir complexidade adicional, pois os usuários podem solicitar o acesso a dados armazenados em diferentes nós do sistema distribuído, e porque um mecanismo de controle de concorrência em um único computador não pode instantaneamente obter informações sobre as operações que ocorrem em outros computadores (Bernstein, Hadzilacos *et al.*, 1987).

Em um sistema de controle de concorrência centralizado, os dados são controlados por um único escalonador. Em um sistema de controle de concorrência distribuído cada nó contém o seu próprio sistema de gerenciamento de dados, além de seu próprio escalonador. O conjunto dos escalonadores dos nós do sistema constitui um escalonador distribuído. A figura 12 ilustra a arquitetura de um SGBD

distribuído. Nesta figura GT significa gerenciador de transações, enquanto que GD significa gerenciador de acesso a dados.

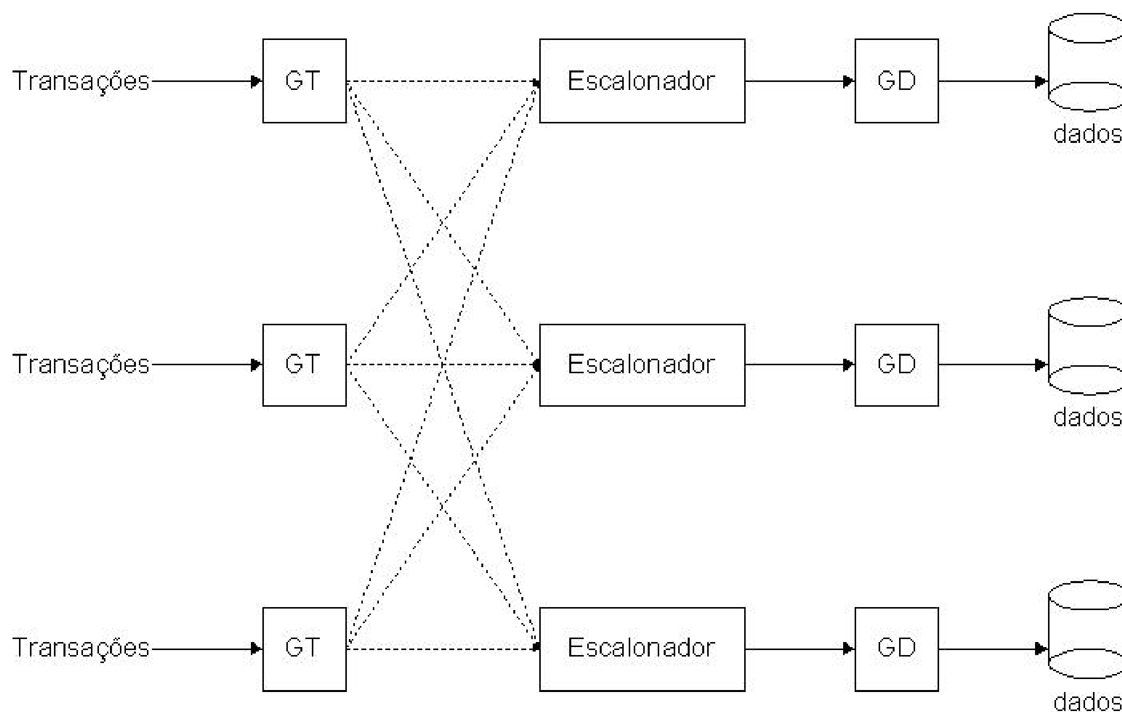

Figura 12 - Arquitetura de um SGBD distribuído.

Do ponto de vista de uma aplicação, uma transação distribuída é executada da mesma forma que uma transação local, ou seja, submetendo as operações a um gerenciador de transações de um servidor de banco de dados.

A cada operação de escrita ou leitura submetida, o gerenciador de transações deve decidir se o dado é local ou se o dado é remoto. Caso o dado seja local, o gerenciador de transações deve enviar a operação ao escalonador local. Caso seja distribuído, ele envia a operação ao banco de dados remoto que, para executá-la, submete-a ao seu escalonador. Em cada nó que uma transação global visitar, ela será executada como uma transação local composta das operações globais que lhe forem enviadas. A transação distribuída é, portanto, a união de diversas transações locais e, desta forma, há mais de um nó participando de uma transação distribuída.

O controle de concorrência de transações distribuídas deve garantir também que qualquer transação sendo executada em diversos nós do sistema deve ter uma terminação atômica, ou seja, garantir que a transação ou se confirma em todos os

nós ou aborta em todos eles. Para isso, são utilizados protocolos de confirmação atômica.

3.6 Método de Controle de Concorrência 2PL

Para um escalonamento ser aceitável ele deve ser seriável. Um escalonamento é dito seriável se garante que a execução concorrente de suas transações produz um efeito equivalente ao de alguma execução das mesmas transações, na qual apenas uma é executada de cada vez. Portanto, todo método de controle de concorrência deve sempre garantir que todos os escalonamentos gerados serão seriáveis (Ferreira e Finger, 2000). As abordagens mais utilizadas para lidar com controle de concorrência são *Two-Phase Locking* (2PL) e *Timestamps* (Bernstein e Goodman, 1981), embora existam outras (Bernstein e Goodman, 1983) (Papadimitriou e Kanellakis, 1984).

3.6.1 2PL

Este método de controle de concorrência é baseado em bloqueios de acesso a recursos compartilhados, que são definidos para cada operação realizada sobre os dados acessados pelas transações. Assim, $rl_i[x]$ é o bloqueio, representado pela letra "l", que deve ser obtido antes de cada operação de leitura sobre o dado x realizada pela transação j, $r_i[x]$. Da mesma forma, $wl_i[x]$ é o bloqueio de escrita que deve ser obtido antes de cada operação de escrita sobre o dado x realizada pela transação j. Analogamente, há operações de desbloqueio $ru_i[x]$ e $wu_i[x]$. Neste caso a letra "u" representa o desbloqueio. Para uma dada transação T_i e um item de dado x, tal seqüência de operações deve ocorrer sempre:

$$rl_i[x] <_j r_i[x] <_j ru_i[x] \qquad wl_i[x] <_j w_i[x] <_j wu_i[x]$$

Dois bloqueios de transações distintas conflitam se suas operações correspondentes conflitam. Ou seja, se p_i e q_j são operações conflitantes, então os bloqueios pl_i e ql_j são bloqueios conflitantes. Duas transações não podem obter bloqueios conflitantes para um mesmo dado, e desta forma:

1. Dois bloqueios de leitura no mesmo dado, $rl_i[x]$ e $rl_j[x]$, não são conflitantes. Assim, um dado pode ser lido por mais de uma transação concorrentemente, por isso o bloqueio de leitura é chamado de bloqueio compartilhado.
2. Se uma transação T_i obtém um bloqueio de escrita, $wl_i[x]$, então nenhuma outra transação T_j poderá obter um bloqueio de escrita ou leitura para x até que a transação T_i seja desbloqueada. Por isso o bloqueio de escrita é chamado de bloqueio exclusivo.

Num dado escalonamento E, se p_i e q_j são operações conflitantes, então elas estão ordenadas: ou $p_i <E\ q_j$ ou $q_j <E\ p_i$. No entanto, para que a segunda operação possa ser executada, é preciso que a primeira tenha sido desbloqueada e a segunda bloqueada nesta ordem. Ou seja, se p_i e q_j são operações conflitantes, então:

$$Se\ p_i <E\ q_j \text{ então } p_i <E\ pu_i <E\ ql_j <E\ q_j$$

Bloqueios de leitura podem ser promovidos a bloqueios de escrita da seguinte maneira: se uma transação T_i possui um bloqueio de leitura no dado x, e T_i solicita um bloqueio de escrita para x, e se nenhuma outra transação possuir outro bloqueio de leitura para x, então este bloqueio poderá ser promovido a um bloqueio exclusivo de escrita.

Entretanto, se alguma outra transação estiver simultaneamente compartilhando este bloqueio de leitura com T_i, o bloqueio de T_i não poderá ser promovido. Assim, T_i será posta em espera e aguardará a liberação de todos os bloqueios de leitura de outras transações sobre x. Somente então o bloqueio de leitura de T_i poderá ser promovido a um bloqueio de escrita.

Na arquitetura de um SGBD genérica, ilustrada na figura 11, o escalonador recebe operações de leitura, escrita, confirmação e aborto do gerenciador de transações e as repassa, possivelmente em ordem diferente, ao gerenciador de dados. Um escalonador que opera pelo método 2PL tem como tarefas:

1. Ao receber uma operação $p_i[x]$, o escalonador deve verificar se o bloqueio $pl_i[x]$ não está em conflito com algum $ql_j[x]$ em atividade. Se houver conflito, o escalonador colocará a transação T_i em uma fila de espera até que o bloqueio necessário seja liberado. Caso contrário, o

escalonador anota que $pl_i[x]$ está ativo escrevendo a tripla (x, p, T_i) numa tabela de bloqueios.

2. Uma vez que o bloqueio $pl_i[x]$ está ativo, ele só poderá ser desbloqueado após o gerenciador de dados informar que a operação $p_i[x]$ foi completada.
3. Existe ainda uma terceira condição que deve ser mantida pelo escalonador, conhecida como condição de duas fases: uma vez que o escalonador desbloqueou um dado qualquer de uma transação, esta transação não poderá mais obter nenhum novo bloqueio, para nenhum item de dado.

Logo, uma transação divide-se em duas fases: a fase de obtenção de bloqueios e a fase de liberação dos mesmos. Ou seja, para quaisquer itens de dados x e y, e quaisquer operações p e q:

$$pu_i[x] \nless_i ql_i[y]$$

É esta terceira condição que dá nome ao método. Ela permite a liberação de qualquer bloqueio apenas quando nenhuma nova solicitação de bloqueio for ocorrer (Ferreira e Finger, 2000).

Apesar de funcional, este modelo de controle de concorrência apresenta uma deficiência. Uma transação só deve desbloquear um dado quando for absolutamente garantido que nenhum novo bloqueio será necessário. O problema é identificar o momento exato em que esta situação ocorre. O modelo apresentado a seguir, uma variação do método 2PL, é capaz de evitar tal problema.

3.6.2 2PL estrito

O término de uma transação é o único ponto de seu processamento em que há certeza de que nenhum novo bloqueio será solicitado por essa transação. Este término pode ser bem sucedido (confirmação) ou mal sucedido (aborto). Um escalonamento 2PL é chamado de estrito quando todos os bloqueios obtidos pela transação são liberados apenas após a sua confirmação ou aborto.

A figura 13 mostra o número de bloqueios obtidos por uma transação a partir do método 2PL genérico e do método 2PL estrito. A linha vertical tracejada indica a separação (arbitrária) entre a fase de aquisição de bloqueios e a fase de liberação dos mesmos, enquanto que a linha vertical contínua indica o momento em que a transação se encerra.

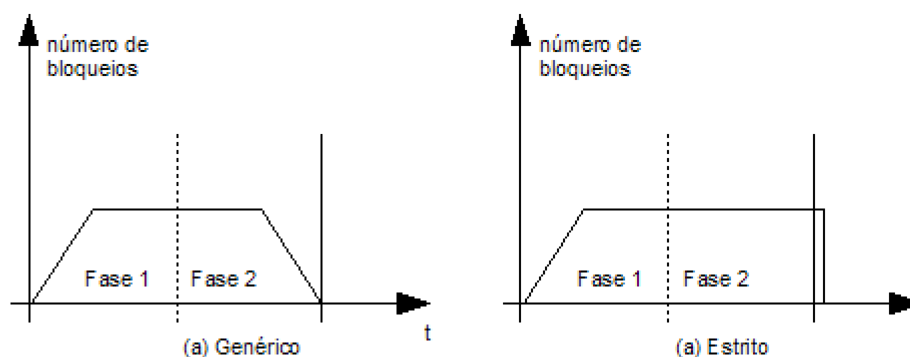

Figura 13 - Número de bloqueios de uma transação ao longo do tempo.

Analisando-se o comportamento do método 2PL genérico, nota-se que o número de bloqueios cresce monotonicamente com o tempo, até estabilizar-se e em seguida decresce monotonicamente. Já no caso do método 2PL estrito, o número de bloqueios cresce a partir do início da transação e também atinge um patamar. A diferença é que no caso estrito todos os bloqueios são liberados de uma única vez somente após o término da transação. O método estrito é, portanto, um caso particular do método genérico (Ferreira e Finger, 2000).

Se o escalonador obedece ao método 2PL estrito, a última transação, T_j, que escreveu sobre um dado qualquer necessariamente se confirmou antes de liberar seus bloqueios. Desta forma, T_i somente irá ler dados de transações confirmadas. Pelo mesmo raciocínio, nota-se que T_i somente irá escrever sobre dados escritos por transações confirmadas.

Portanto, o método 2PL estrito gera escalonamentos estritos, o que significa que uma transação só lê ou escreve sobre dados escritos por transações confirmadas. Tal regra traz o benefício de que os abortos podem ser processados restabelecendo-se as imagens prévias dos dados escritos pela transação abortada.

É importante destacar que tanto o 2PL quanto o seu caso estrito estão suscetíveis à situação de travamento (*deadlock*), ou seja, a uma situação em que um conjunto de transações não pode prosseguir o processamento e ficam indefinidamente num estado de espera. Um sistema travado, sem interferência externa, continuará travado para sempre e é tarefa do escalonador 2PL detectar a ocorrência de travamentos e abortar uma das transações envolvidas, a chamada vítima, liberando seus bloqueios e, consequentemente, destravando todas as transações envolvidas. Posteriormente, a transação vítima deve ser reiniciada.

3.6.3 2PL distribuído

Em um sistema de banco de dados distribuído em que cada escalonador obedece ao protocolo 2PL, cada escalonador mantém sua própria tabela de bloqueios, logo, uma transação distribuída pode ter bloqueios em diversos nós do sistema, cada um dos quais localmente obedecendo ao 2PL.

A observância do 2PL local, no entanto, não garante a observância do 2PL globalmente. Pode existir uma transação liberando bloqueios em um nó N1, enquanto em um nó N2 outros bloqueios ainda estão sendo obtidos. Tal situação pode levar a escalonamentos não seriáveis de transações distribuídas, pelo mesmo motivo que uma violação da condição de duas fases localmente pode levar a escalonamentos não seriáveis.

É necessário, portanto, uma condição distribuída de duas fases que garanta que uma vez que o escalonador desbloqueou um dado qualquer de uma transação em algum nó da rede, essa transação distribuída não poderá mais obter nenhum novo bloqueio em qualquer nó da rede.

Para que o 2PL distribuído não cause um grande aumento de mensagens na rede, de um nó para todos os outros participantes em uma transação distribuída, avisando quando não há mais bloqueios a serem solicitados, todos os escalonadores locais devem adotar o 2PL estrito. E para respeitar a observância da condição distribuída de duas fases, é necessário garantir que os nós participantes da transação só receberão a solicitação de confirmação após todos os nós terem executados todas as suas operações. Tal comportamento só é possível através de um protocolo de confirmação atômica.

Um escalonamento produzido por um sistema que respeita 2PL localmente e a condição distribuída de duas fases é indistinguível de um escalonamento 2PL local e, portanto, só pode gerar escalonamentos seriáveis. Além disso, se cada 2PL local é estrito, então o 2PL distribuído também possui esta característica.

3.7 Método de Controle de Concorrência Timestamps

O controle de concorrência por marcas de tempo (*timestamps*) atribui uma marca de tempo a cada transação T_i, $mt(T_i)$. As marcas de tempo são valores totalmente ordenados atribuídos em ordem crescente às transações, ou seja, se T_i é diferente de T_j, então ou $mt(T_i) < mt(T_j)$ ou $mt(T_j) < mt(T_i)$. As transações são escalonadas de acordo com a ordem de suas marcas de tempo, o que é garantido de acordo com a seguinte regra, chamada de regra MT: se $p_i[x]$ e $q_j[x]$ são duas operações conflitantes, então $p_i[x]$ será executada antes de $q_j[x]$ se e somente se $mt(T_i) < mt(T_j)$.

A regra MT garante que as transações sejam seriáveis, pois as marcas de tempo são totalmente ordenadas, conseqüentemente o escalonamento será sempre seriável. Normalmente é o gerenciador de transações que atribui as marcas de tempo. Em um sistema centralizado, isto pode ser facilmente gerado usando-se um contador, o qual pode ser o próprio relógio do sistema, o que garante a ordenação das transações.

Em sistemas distribuídos, a ordem total pode ser obtida da seguinte maneira, cada gerenciador de transações recebe um número único e mantém um contador seguido pelo número do gerenciador de transações. Assim, duas marcas de tempo geradas por gerenciadores diferentes sempre serão diferentes e totalmente ordenadas.

3.7.1 Funcionamento

Considera-se que uma operação $p_i[x]$ chegou tarde demais se o escalonador já executou uma operação conflitante $q_j[x]$ com $mt(T_i) < mt(T_j)$. Neste caso, se $p_i[x]$ for executada, a regra MT será violada. Só resta como opção abortar T_i e reiniciá-la com uma nova marca de tempo $mt'(T_i) > mt(T_j)$.

Para que este esquema funcione corretamente, cada item de dados x no banco de dados recebe dois valores, que são alterados com o tempo:

- *r-mt*(*x*): valor da marca de tempo da última transação a ler *x*.
- *w-mt*(*x*): valor da marca de tempo da última transação a escrever sobre *x*.

O funcionamento básico do protocolo por marcas de tempo é o seguinte (Ferreira e Finger, 2000):

1. Para executar uma operação de leitura $r_i[x]$, o valor de *w-mt*(*x*) deve ser verificado.
   - Se *w-mt*(*x*) < $mt(T_i)$, então *r-mt*(*x*) := $mt(T_i)$, e a leitura deverá ser executada.
   - Caso contrário, *w-mt*(*x*) > $mt(T_i)$, T_i deve ser abortada e reiniciada com uma nova marca de tempo $mt'(T_i)$ > *w-mt*(*x*).
2. Para executar uma operação de escrita $w_i[x]$, o valor de *r-mt*(*x*) e *w-mt*(*x*) devem ser verificados.
   - Se $mt(T_i)$ é maior que ambos, então *w-mt*(*x*) := $mt(T_i)$, e a operação de escrita deverá ser executada.
   - Caso contrário, a operação de escrita deve ser rejeitada, T_i deve ser abortada e reiniciada com uma nova marca de tempo maior que *r-mt*(*x*) e *w-mt*(*x*).

Este método nunca gera travamentos, mas não está livre de reinícios cíclicos (*livelock*). Além disso, embora os escalonamentos gerados sejam sempre seriáveis, eles nem sempre são estritos ou recuperáveis.

3.7.2 Timestamps estrito

Para garantir que escalonamentos por marcas de tempo serão sempre estritos, leituras e escritas a dados escritos por transações não confirmadas devem ser proibidas. O método *timestamps* estrito possui duas filas de operações, *op-em-transito* e *op-em-espera*. Uma vez que uma operação $p_i[x]$ foi autorizada para execução, ela não é executada imediatamente e deve seguir às seguintes regras:

1. Se não há uma operação conflitante $q_j[x]$ em *op-em-transito*, então a operação $p_i[x]$ é executada e inserida em *op-em-transito*.
2. Se, por outro lado, há uma operação conflitante $q_j[x]$ em *op-em-transito*, então $p_i[x]$ não é executada imediatamente e a operação $p_i[x]$ é inserida em *op-em-espera*.

Quando a transação T_j termina (com ou sem sucesso), todas as suas operações são removidas da fila *op-em-transito*. Se T_j foi abortada, então suas operações também são removidas de *op-em-espera*. Em seguida, por ordem de marca de tempo, as operações de cada transação em *op-em-espera* são examinadas e, se não houver mais conflito, elas serão executadas e inseridas em *op-em-transito*.

Nota-se que as filas *op-em-transito* e *op-em-espera* funcionam como bloqueios, mas não há perigo de travamentos uma vez que a regra MT é obedecida. Tais filas garantem que os escalonamentos serão estritos. Para melhorar a eficiência, as filas *op-em-transito* e *op-em-espera* podem ser parametrizadas por item de banco de dados e tipo de operação (leitura ou escrita). Por fim, é importante notar que o este método ainda pode sofrer reinícios cíclicos.

3.7.3 Timestamps distribuído

Para que o método de controle de concorrência por marcas de tempo seja distribuído, cada nó do sistema deve possuir seu próprio escalonador por marcas de tempo e um identificador único. Assim, o esquema de produzir uma marca de tempo com dois componentes pode ser implementado.

A decisão sobre permitir, atrasar ou rejeitar uma operação sobre um item de dados *x* é tomada localmente, e cada escalonador mantém apenas as informações sobre seus dados locais, independentemente de onde foi iniciada a transação que realiza as operações. O funcionamento do método permanece basicamente o mesmo. Além disso, o método por marcas de tempo distribuído possui a vantagem de não necessitar a comunicação entre os nós para coordenar a detecção de travamentos distribuídos. Ou seja, não há nenhuma necessidade de comunicação entre os escalonadores.

Por fim, um escalonador que utiliza o método por marcas de tempo pode participar de um banco de dados distribuído com outros escalonadores 2PL, bastando que os escalonadores 2PL também atribuam marcas de tempo a suas transações. Quanto ao procedimento para a confirmação distribuída, ela permanece a mesma independentemente do escalonador, pois este é um problema de recuperação.

4 SISTEMAS UTILIZADOS

Para o desenvolvimento do sistema proposto neste trabalho, foi adotada uma plataforma de agentes, denominada JADE, descrita a seguir. Tal plataforma é um *middleware*[5] (Bernstein, 1996) de agentes e por este motivo o sistema desenvolvido se integra a ela estendendo-a, ou seja, adicionando um conjunto de novas funcionalidades.

Durante todos os testes com bancos de dados distribuídos foi utilizado o sistema gerenciador de banco de dados paralelo distribuído SisBDPar (Lubacheski, 2005), descrito sucintamente na seção 4.2. Para o desenvolvimento de um mecanismo de controle de concorrência, incluído no sistema aqui proposto, os padrões descritos na seção 4.3 foram utilizados.

4.1 JADE

O JADE (*Java Agent Development Framework*) é um *middleware* voltado para o desenvolvimento e execução de aplicações distribuídas desenvolvidas em linguagem Java e baseadas no paradigma de agentes em conformidade com o padrão FIPA. O objetivo do JADE é simplificar o desenvolvimento de sistemas multi-agentes, enquanto garante a conformidade com o padrão FIPA através de um conjunto de sistemas de serviços e agentes prontos e disponíveis.

Esta plataforma oferece um sistema de execução que facilita o desenvolvimento de sistemas multi-agentes através do uso de um modelo de agentes programável pré-definido, além de um conjunto de ferramentas de gerenciamento e de testes. Assim, além de um *middleware*, o JADE é também um *framework*.

O JADE é um projeto *Open Source* sobre licença LGPL (GNU Lesser General Public License, 1999) que possui uma comunidade crescente de colaboradores e usuários, além de ser controlado por uma organização oficial sem fins lucrativos

[5] Camada de software que faz a mediação entre componentes de software ou aplicações.

chamada JADE Governing Board. O JADE já foi utilizado com sucesso por diversas instituições e empresas (Bellifemine, Caire *et al.*, 2003).

4.1.1 Arquitetura do JADE

O JADE é um sistema distribuído e sua arquitetura baseia-se na coexistência de várias JVMs (*Java Virtual Machine*) que podem estar espalhadas por diversos *hosts*. Cada JVM corresponde a um contêiner de agentes que provê um ambiente de execução completo, além de permitir que vários agentes sejam executados concorrentemente no mesmo *host*. O JADE ainda permite a integração de diversas plataformas, que correspondem a um grupo de contêineres.

Devido à conformidade com o padrão FIPA, o JADE inclui o AMS, o DF e o ACC, que são iniciados automaticamente. No JADE o AMS e o DF são implementados como agentes. Apenas um contêiner atua como *front-end* (o contêiner principal) em uma plataforma, no qual os agentes AMS e DF estão localizados, além do *RMI Register*. Já o ACC deve estar presente em todos os contêineres para prover, de forma transparente, serviços de troca de mensagens eficientes e flexíveis às aplicações dos usuários.

No JADE existem três tipos possíveis de troca de mensagens entre agentes:

1. **Intra-contêiner**: os dois agentes que interagem estão no mesmo contêiner. As mensagens trocadas dentro de um mesmo contêiner são transportadas por meio de eventos Java.
2. **Intra-plataforma**: os agentes estão em contêineres diferentes, mas na mesma plataforma. Neste caso o JADE faz uso de Java RMI (*Remote Method Invocation*) (Downing, 1998).
3. **Inter-plataforma**: os agentes remetente e destinatário localizam-se em plataformas diferentes. A troca de mensagens entre plataformas é realizada utilizando o ACC em conjunto com um tradutor e o protocolo IIOP (*Internet Inter-ORB Protocol*) (CORBA™/IIOP™ Specification, 2002), e está em conformidade com as especificações FIPA.

O JADE permite ainda mobilidade de agentes intra-plataforma, incluindo a transferência tanto do código, quando necessário, quanto do estado do agente.

Quando uma mensagem ultrapassa a fronteira de uma plataforma, ela é automaticamente convertida de/para a sintaxe, codificação e protocolo de transporte, todos em conformidade com o padrão FIPA. Tal conversão é transparente para quem implementa os agentes, assim é necessário apenas lidar com os objetos Java.

O contêiner principal é responsável por manter internamente os seguintes itens:

- Os registros de RMI usados por outros contêineres ao se registrarem na plataforma.
- Uma tabela de todos os contêineres registrados e de suas referências de objeto RMI (*Agent Contêiner Table*).
- Uma tabela global de descrição de agentes (*Agent Global Descriptor Table*) que relaciona cada nome de agente com seus dados AMS e suas referências de objeto RMI.

Cada contêiner arquiva as referências de objetos dos outros contêineres em uma memória cachê sempre que mensagens são enviadas. Isso é feito para evitar que a *Agent Global Descriptor Table* seja consultada inúmeras vezes, melhorando assim o desempenho da plataforma e diminuindo a sobrecarga, que depende da localização do receptor e do estado da memória cachê.

O modelo de execução de agentes JADE é baseado na utilização de *behaviours*. Um *behaviour*, ou comportamento, de um agente representa uma tarefa específica que o agente deve executar. O objetivo de um agente JADE pode ser identificado como o conjunto de comportamentos que possui. O JADE oferece diversos tipos de *behaviours*, como, por exemplo, para tarefas cíclicas executadas continuamente até que um evento ocorra, ou para tarefas que são executas uma vez só e se encerram em seguida.

A partir deste modelo, o conjunto de comportamentos é escalonado e executado de forma cooperativa e não-preemptiva dentro de cada agente de maneira múltipla, paralela e concorrente. Cada comportamento é como se fosse um segmento[6] local de um software que implementa um agente (Bellifemine, Caire *et al.*,

---

[6] O termo segmento neste contexto refere-se a processos leves que possuem compartilhamento de espaço de memória.

2005). No entanto, o agente como um todo é apenas um segmento. Isso ocorre devido à tentativa do JADE de limitar o número de segmentos criados.

A arquitetura do JADE está representada na figura 14. Os elementos "A" representam agentes, os elementos "B" representam comportamentos de agentes, RMI representa o *RMI Register* e as setas representam os três diferentes tipos de troca de mensagens possíveis na arquitetura do JADE.

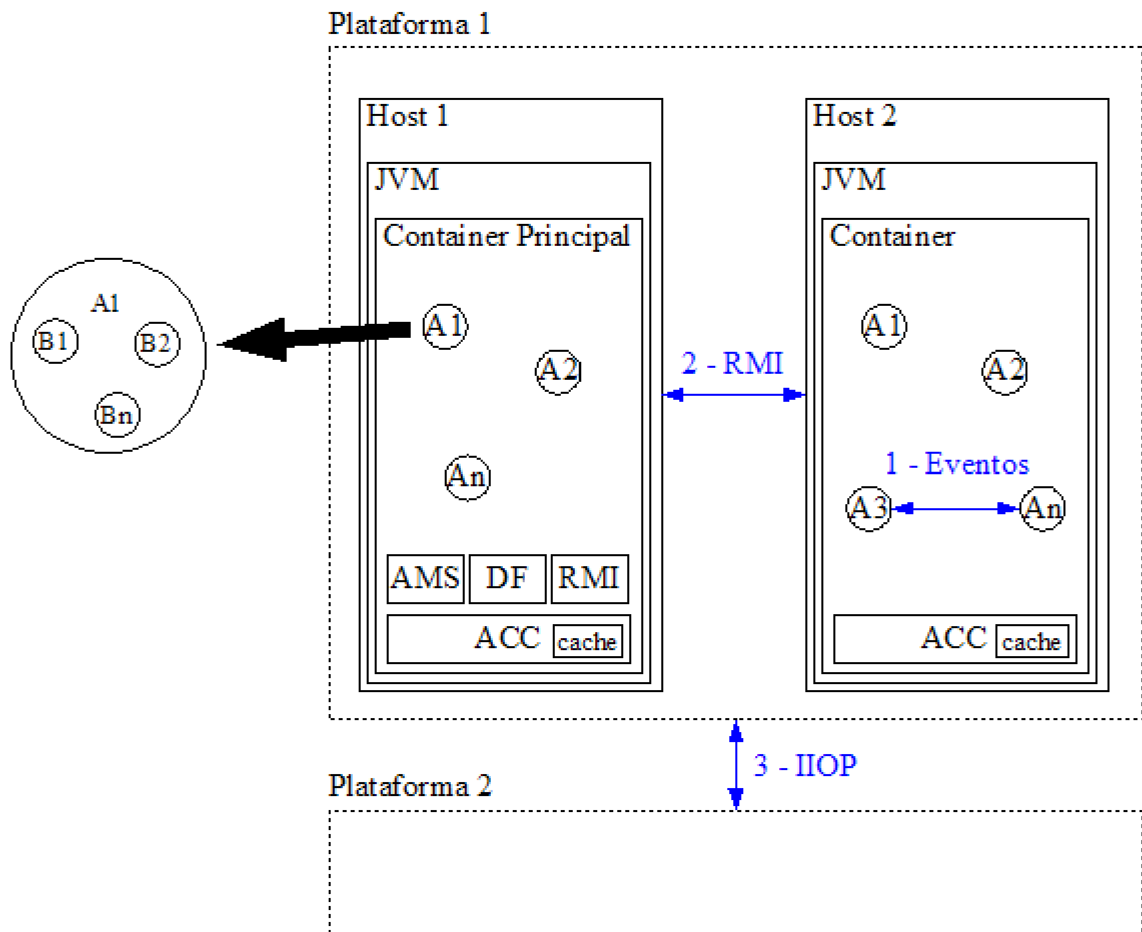

Figura 14 - Arquitetura do JADE.

O JADE permite a cada agente descobrir dinamicamente outros agentes e se comunicarem. Os agentes se comunicam trocando mensagens assíncronas individuais uns com os outros, assim não há dependências temporais entre eles. Apesar deste tipo de comunicação a segurança é preservada, já que o JADE provê mecanismos de autenticação e verificação de direitos dos agentes.

Os agentes enviam e recebem objetos Java, que representam mensagens ACL (FIPA ACL Message Structure Specification, 2002) dentro do escopo dos protocolos de interação. O JADE codifica transparentemente todas as mensagens.

Além de uma biblioteca de execução e programação de agentes, o JADE ainda oferece algumas ferramentas que permitem o gerenciamento da plataforma, a monitoração, e a depuração da comunidade dos agentes (Bellifemine, Poggi *et al.*, 2001). Todas essas ferramentas também são implementadas como agentes FIPA.

4.2 SisBDPar

O SisBDPar é um sistema gerenciador de banco de dados paralelo e distribuído projetado e implementado sobre o NPFS (*Network Parallel File System*) (Guardia, 1999), um sistema de arquivos paralelo que permite ao SisBDPar distribuir o processamento de suas consultas e sub-consultas, aumentando o desempenho da base de dados, já que as consultas são decompostas entre diversos processadores.

Uma aplicação cliente pode conectar-se ao SisBDPar e utilizá-lo de acordo com o modelo cliente-servidor. O SisBDPar oferece uma API (*Application Programming Interface*), cujas funções possibilitam ao cliente realizar consultas em uma linguagem de alto nível. Durante a execução de uma consulta, parte do processamento é realizado na máquina cliente e informações contidas no catálogo são consultadas, assim o sistema define quais servidores locais serão utilizados. Os servidores locais são considerados servidores de dados, e são responsáveis por lidar com o processamento paralelo das consultas.

Como pode ser visto na figura 15, os clientes (API/Cliente) podem acessar as informações armazenadas a partir de qualquer nó da arquitetura distribuída, e os servidores de dados locais (SGBDs locais) são iniciados em cada nó que contenha um servidor NPFS. Através da API/Cliente as consultas são analisadas antes de serem transmitidas para os SGBDs locais. Tanto a API/Cliente quanto os SGBDs locais possuem componentes que permitem estabelecerem e administrarem comunicação entre si.

Durante a requisição de uma consulta, os operadores relacionais são enviados aos SGBDs locais, que os executam e produzem como resultado uma

relação temporária. A interação entre a API/Cliente e os SGBDs locais procede da seguinte maneira durante o processamento de uma consulta:

1. A consulta solicitada é analisada e avaliada pela API/Cliente, que determina a necessidade de utilizar os SGBDs locais ou os servidores NPFS ou ambos. Além disso, é criada uma relação temporária para armazenar o resultado da consulta.
2. Os SGBDs locais processam o operador relacional e armazenam o resultado em uma relação temporária.
3. A API/Cliente combina os resultados a partir da relação resultante a e repassa ao cliente através de uma estrutura interna, permitindo a ele apresentar ou processar o resultado.

Caso tenha necessidade de utilizar os servidores NPFS, a API/Cliente faz somente solicitações de leitura e escrita de uma seqüência de bytes, e toda coordenação e processamento da consulta ficam sob sua responsabilidade.

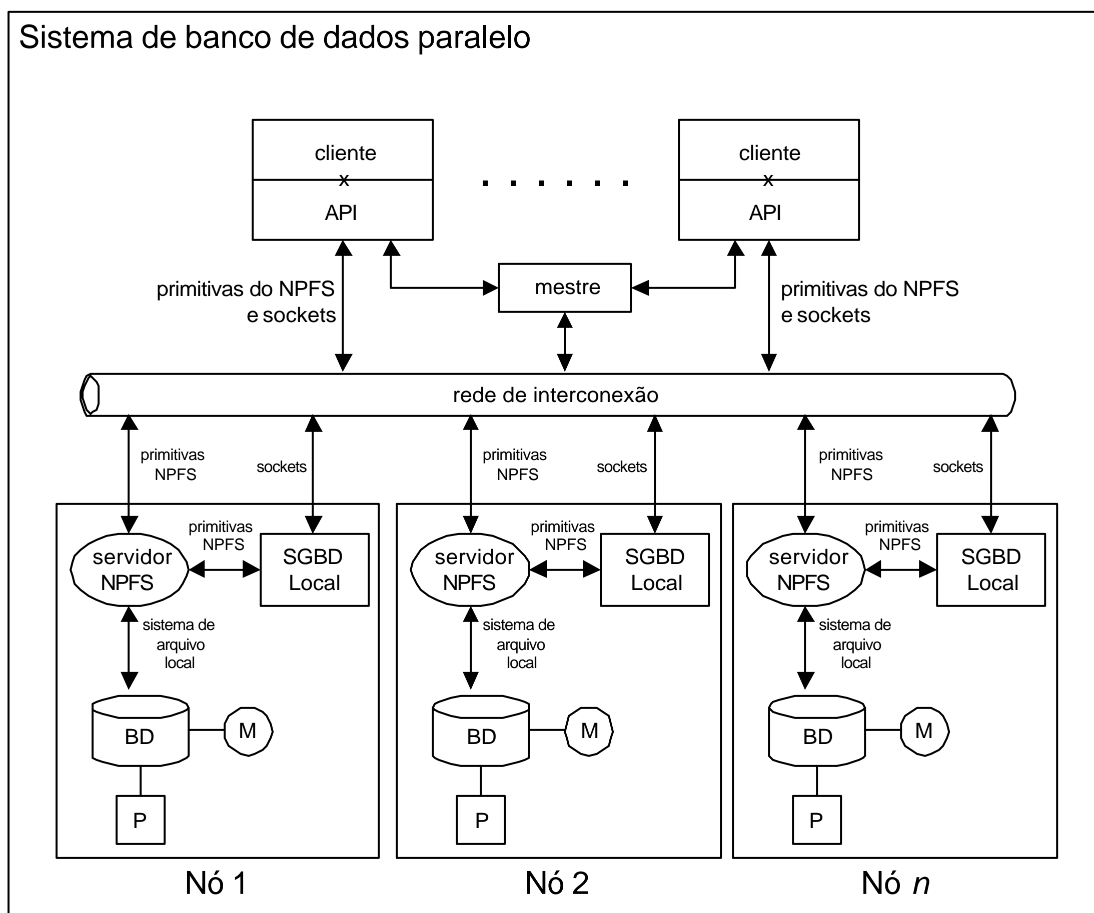

Figura 15 - Arquitetura do SisBDPar.

Especificamente para o desenvolvimento do sistema relatado neste trabalho foi desenvolvido um *driver*[7] JDBC (*Java Database Connectivity*) (Ellis, Ho *et al.*, 2001) para o SisBDPar utilizando JNI (*Java Native Interface*) (Gordon, 1998). O uso de JNI justifica-se pelo fato de o SisBDPar ser desenvolvido em linguagem C, enquanto que o DASE é desenvolvido em linguagem Java.

[7] O termo *driver* refere-se a um componente de software responsável por permitir a abstração de um sistema complexo cuja funcionalidade deve ser disponibilizada através de uma interface bem definida. Esta palavra não foi traduzida em razão de não haver um termo correspondente em português que confira o mesmo significado. Este termo sempre aparecerá neste trabalho em itálico.

5 DASE: Proposta de um ambiente de serviços de acesso a dados distribuídos a agentes

Este trabalho propõe e descreve um sistema que promove a integração entre sistemas multi-agentes e sistemas de banco de dados distribuídos, resolvendo inclusive problemas como a melhor forma de oferecer os dados considerando o paradigma de comunicação entre agentes, e a garantia de consistência dos dados distribuídos mesmo existindo o acesso concorrente. Tal sistema chama-se DASE, *Distributed data Agent Service Environment*, e seus principais objetivos são:

1. Prover a agentes de um sistema multi-agentes acesso a dados distribuídos de forma simples e transparente.
2. Oferecer recursos adicionais relacionados ao acesso aos dados, como, por exemplo, a garantia de consistência de dados através de controle de concorrência.

O DASE não pretende ser uma plataforma de execução de agentes, sua proposta é ser um ambiente que ofereça um conjunto de serviços que se integre de forma harmônica aos demais agentes de uma plataforma de agentes já existente. Portanto, o DASE é parte de um sistema multi-agente.

Os serviços de seus agentes são relacionados ao acesso a dados distribuídos de um modo transparente, bem definido, eficiente e em conformidade com qualquer padrão que a plataforma de agentes se comprometa a seguir.

O apêndice A deste trabalho traz o diagrama de contexto e a lista de eventos do DASE. O apêndice B traz os casos de uso e o diagrama de casos de uso relacionados ao DASE.

5.1 Plataforma de Agentes Adotada

Existem alguns sistemas, tanto comerciais quanto acadêmicos, que oferecem uma plataforma de execução de agentes, ou, em outras palavras, um *middleware* de

agentes. De todas as plataformas de agentes pesquisadas nenhuma apresentou qualquer serviço ou funcionalidade, nativa ou acessória, semelhante à proposta pelo DASE, ou mesmo que considerasse o simples acesso a dados centralizados.

Dessas plataformas, algumas estão em fase inicial de desenvolvimento, e outros se restringem a somente um determinado tipo de sistema multi-agente, ou então são voltados apenas a simulações (Hiebeler, 1994) (Sonnessa) (North, Collier *et al.*, 2006). Outras, como (Busetta, Rönquist et al., 1999) e (Bellifemine, Poggi et al., 2001) oferecem um sistema funcional no entanto não apresentam os recursos providos pelo DASE.

Assim, certamente estão sujeitas a fazer acesso a dados como qualquer outra aplicação faria, de forma não padronizada, pouco manutenível e flexível, e completamente dependente dos detalhes da requisição de dados, do ambiente de agentes e do SBD utilizado. Entretanto, duas plataformas destacam-se por oferecerem um ambiente de execução de agentes funcional, que trazem consigo diversas ferramentas auxiliares, e já são utilizadas tanto pela indústria, quanto pelo meio acadêmico.

JACK (Busetta, Rönquist *et al.*, 1999) é um arcabouço de agentes desenvolvido em Java e que implementa a arquitetura BDI, *Belief-Desire-Intention* (Bratman, 1987). O JACK não é restrito a nenhuma linguagem específica de comunicação entre agentes, oferecendo inclusive a KQML, *Knowedge Query Manipulation Language* (Finin e Fritzson, 1994), embora tenha sido projetado para priorizar a comunicação baseada em troca de mensagens e ORB, *Object Request Broker* (Vinoski, 1997). Esta decisão de projeto reflete a busca de maior compatibilidade com a indústria, como por exemplo, a tentativa de aproveitar a extensa adoção do paradigma orientado a objetos, e a necessidade de interoperabilidade entre sistemas.

Outra plataforma de agentes relativamente madura e robusta é o JADE, que, assim como o JACK, também é desenvolvido totalmente em Java. Conforme já revelado no quarto capítulo, a plataforma escolhida sob a qual o DASE foi projetado é o JADE. Os principais motivos da escolha desta plataforma são apresentados a seguir:

- É desenvolvida em Java, o que garante portabilidade, fácil integração com sistemas distribuídos, além de todas as vantagens que o paradigma orientado a objetos oferece.
- É um projeto *Open-Source*, o que favorece a formação de uma comunidade de usuários participativos e a evolução do projeto de um modo descentralizado e transparente, além de ser mais coerente com o contexto de um projeto de cunho acadêmico.
- Está relativamente madura, estável e possui um conjunto de funcionalidades, ferramentas e documentação razoável.
- Possui a preocupação de estar em conformidade com as especificações FIPA, o que certamente favorece a padronização e a evolução da tecnologia.

5.2 Principais Características

Antes de listar as características do DASE, é necessário definir os seguintes termos a serem utilizados ao longo do texto:

- **Agentes clientes**: agentes do sistema multi-agente que se relacionarão diretamente ou indiretamente com o DASE.
- **Ambiente de dados**: conjunto de informações referentes à identificação de um SBD específico cujos dados serão disponíveis aos agentes clientes através dos serviços prestados pelo DASE.

Assim como o JADE, o DASE também é desenvolvido usando Java. As principais características do DASE são apresentadas abaixo:

- Seus serviços são disponibilizados através de agentes em conformidade com o padrão FIPA.
- Qualquer complexidade inerente ao acesso aos dados é abstraída, o que garante a transparência durante o acesso aos dados. Assim, a localidade física dos dados acessados é transparente aos agentes clientes mesmo se os dados estiverem distribuídos.

- Acesso a dados pelos agentes clientes de maneira padronizada independentemente do tipo ou versão do gerenciador de recursos, desde que este possua *driver* JDBC (Ellis, Ho *et al*., 2001).
- Controle de concorrência e balanceamento de carga.
- Uso simultâneo de múltiplos SBDs diferentes.
- Acesso aos dados pelos agentes clientes sempre através de requisição SQL ou alguma variação adotada pelo gerenciador de recursos.

6 ARQUITETURA DO DASE

O DASE interage com três sistemas distintos, os agentes clientes, os agentes JADE e o SBD. Os agentes clientes utilizam serviços tanto de agentes DASE quanto de agentes JADE. Os agentes DASE também utilizam serviços de agentes JADE, além de utilizar os serviços do SBD. A figura 16 lustra a interação entre o DASE e estes outros sistemas.

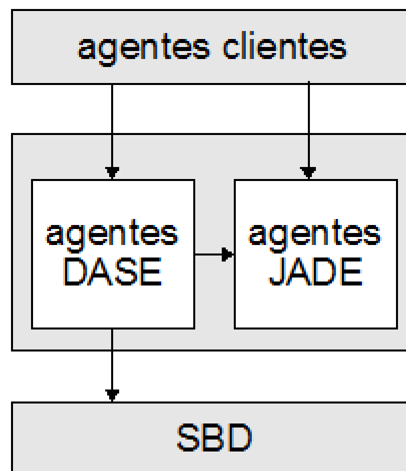

Figura 16 - Interação entre o DASE e os três sistemas.

Na figura 16 a seta indica a dependência em relação à prestação de serviços. Os serviços disponíveis pelo JADE aos agentes clientes e aos agentes DASE são basicamente: controle do ciclo de vida e identificação dos agentes, publicação e localização de seus serviços, e infra-estrutura de comunicação entre os agentes. Os serviços providos pelos agentes DASE são localizados pelos agentes clientes graças à publicação no agente JADE DF.

6.1 Agentes DASE

Os agentes DASE são classificados através dos seguintes grupos:

1. Núcleo do sistema:
   - *DASEProxy* (DX)
   - *DASEManager* (DM)
   - *DASEConfigurator* (DC)
2. Serviço de acesso a dados:
   - *DASEEnvironment* (DE)
   - *DASEServiceLocator* (SL)
   - *DASENode* (Nd)
   - *DASEDataProvider* (DP)
3. Controle de concorrência:
   - *Scheduler* (Sc)
   - *ConcurrencyController* (CC)
4. Utilitários:
   - *Configure*
   - *Request*

6.1.1 Núcleo do sistema

São os principais agentes e objetos do DASE, responsáveis por iniciar, encerrar, controlar e monitorar o sistema.

6.1.1.1. *DASEProxy* (DX)

Este tipo de agente tem papel importante na API do DASE, pois é através dele que um agente cliente solicita o início e encerramento de todo o sistema. Através do *DASEProxy* um agente cliente pode também obter o AID do *DASEManager*, tipo de agente descrito a seguir, embora isso possa ser feito

também através de uma consulta junto ao DF. Este agente também é capaz de oferecer informações sobre o sistema, como sua versão, e se ele está em execução. Além disso, agentes *DASEProxy* mantêm informações sobre todos os contêineres JADE existentes na plataforma. Tais informações são utilizadas para evitar que um agente cliente referencie um contêiner inválido durante a inclusão de um nó de um novo ambiente de dados. Este controle é realizado pelo objeto *DASEEnvironmentDescriptor*, descrito na seção 8.2. Há um ou nenhum agente *DASEProxy* em cada nó que faz parte do DASE. A subseção 6.1.5 explica a nomenclatura, a quantidade e a localização de cada agente DASE.

6.1.1.2. *DASEManager* (DM)

É o principal tipo de agente DASE, sendo responsável pela configuração inicial do sistema e seu encerramento, já que controla todos os agentes *DASEProxy*, *DASEEnvironment* e o agente *DASEConfigurator*, caso este existir. Um agente *DASEManager* permite operações como: criação, carregamento, início, configuração, interrupção e encerramento de ambientes de dados, representados por agentes *DASEEnvironment*. Assim, possui papel importante no DASE. Só há um agente deste tipo em todo o sistema.

6.1.1.3. *DASEConfigurator* (DC)

Este tipo de agente representa a única interface gráfica do DASE com o usuário, e oferece acesso a recursos de configuração global do sistema e controle de ambientes de dados, permitindo operações como criação, remoção, modificação e monitoração de ambientes de dados. Um agente *DASEConfigurator* ainda traz informações extras sobre o sistema. Só há um agente deste tipo em todo o sistema.

6.1.2 Serviço de acesso a dados

São os agentes que lidam, diretamente ou indiretamente, com os agentes clientes, por isso compõem uma parte importante da interface de todo sistema com os agentes clientes. Suas tarefas estão relacionadas à representação de ambientes de dados e serviço de requisição de dados.

6.1.2.1. *DASEEnvironment* (DE)

Este tipo de agente representa e controla um ambiente de dados. Sua principal função é registrar informações sobre quais são os nós DASE participantes, além de características sobre o SBD utilizado, sobre o balanceamento de carga e sobre o controle de concorrência. Agentes *DASEEnvironment* são responsáveis pelo controle dos agentes *DASENode*, descritos abaixo. Todo ambiente de dados deve ter pelo menos um nó, e, conseqüentemente, um agente *DASENode*. Como múltiplos ambientes de dados podem ser disponibilizados simultaneamente pelo DASE, mais de um agente deste tipo pode existir no mesmo nó, embora sempre de ambientes de dados diferentes. Um agente *DASEEnvironment* cria e controla um agente *DASEServiceLocator*.

6.1.2.2. *DASEServiceLocator* (SL)

É o tipo de agente requisitado pelo agente cliente quando este necessita dos serviços de um agente *DASEDataProvider*, que é responsável pela requisição de dados. Este agente representa o *service locator* (Endrei, Ang *et al.*, 2004) presente em qualquer arquitetura orientada a serviços. Junto dos agentes *DASENode* e *DASEEnvironment*, é um dos agentes responsáveis pelo balanceamento de carga do DASE. Existe um agente *DASEServiceLocator* para cada agente *DASEEnvironment*.

6.1.2.3. *DASENode* (Nd)

Representa um nó de um SBD em um contêiner do JADE para um determinado ambiente de dados. Um agente *DASENode* é responsável pelo controle dos objetos *ConnectionProvider* e *DataBridge*, e do agente *Scheduler* de seu nó. Todo agente *DASENode* registra informações sobre a conexão ao SBD através de seu nó, além de dados estatísticos e de controle, como controle de concorrência e balanceamento de carga, de acordo com especificações de seu ambiente de dados. Existe um agente *DASENode* para cada nó de um ambiente de dados. Um nó pode abrigar mais de um agente *DASENode*, já que mais de um ambiente de dados pode

fazer uso daquele nó. Os agentes *DASENode* de um ambiente de dados participam de forma conjunta do balanceamento de carga do sistema.

6.1.2.4. *DASEDataProvider* (DP)

Um agente *DASEDataProvider* comunica-se indiretamente com os agentes clientes recebendo suas requisições de serviço de acesso a dados através do agente *DASEServiceLocator*. O agente *DASEDataProvider* tem papel importante durante o processo de controle de concorrência, atuando como gerenciador de transações. Este agente é baseado no padrão de projeto *Façade* (Rising, 1999), embora esteja votado ao paradigma de troca de mensagens, que é coerente com sistemas multi-agentes.

6.1.2.5. *ConnectionProvider* (CP)

O *ConnectionProvider* não é um tipo de agente, mas sim um objeto. Um objeto *ConnectionProvider* guarda qualquer tipo de informação necessária para conectar-se ao nó do SBD, além de estabelecer e manter uma conexão direta com o SBD, algo que nenhum agente DASE faz. Existe um objeto *ConnectionProvider* para cada agente *DASENode*.

6.1.2.6. *DataBridge* (DB)

Assim como o *ConnectionProvider*, o *DataBridge* também é somente um objeto. Além do objeto *ConnectionProvider*, é o único que se relaciona diretamente com o SBD. Entretanto, somente este objeto é capaz de enviar solicitações e receber dados diretamente do SBD. Sua função é repassar ao SBD toda instrução SQL que receber. O objeto *DataBridge* depende da conexão mantida pelo objeto *ConnectionProvider* de seu nó. Um objeto *DataBridge* possui papel importante durante o processo de controle de concorrência e controle de recuperação, atuando como gerenciador de dados. Existe um objeto *DataBridge* para cada agente *DASENode*.

6.1.3 Controle de Concorrência

São os agentes e objetos responsáveis por garantir a consistência dos dados mesmo que haja acessos concorrentes.

6.1.3.1. *Scheduler* (Sc)

O escalonamento das operações das transações é realizado pelo agente *ConcurrencyController*, no entanto são agentes *Scheduler* que recebem a requisição de execução de transações do agente *DASEDataProvider* e controlam a correta execução de cada operação da transação. Eles realizam tarefas importantes também, como prevenção de reinícios cíclicos e solicitação de aborto de transações em caso de violação de marca de tempo ou execução de operação com sentença SQL inválida. Caso o controle de concorrência esteja ativado em um ambiente de dados, cada um de seus nós terá um agente *Scheduler*, o que significa exatamente um agente *Scheduler* para cada agente *DASENode*.

6.1.3.2. *ConcurrencyController* (CC)

É o principal tipo de agente responsável pelo controle de concorrência oferecido pelo DASE, atuando de fato como o escalonador de transações, embora isso seja possível graças à interação de agentes *Scheduler* e um agente *ConcurrencyController*. O agente *ConcurrencyController* opera um método de controle de concorrência de acordo com o que foi configurado para o ambiente de dados. Todo ambiente de dados em que o controle de concorrência estiver ativado terá um agente *ConcurrencyController*, que ficará no mesmo nó em que o agente *DASEEnvironment* estiver, enquanto que cada nó do ambiente de dados terá um agente *Scheduler*. Portanto para um ambiente de dados com *n* nós sempre há um agente *ConcurrencyController* e *n* agentes *Scheduler*, um em cada nó.

6.1.4 Utilitários

São agentes que facilitam a interação entre agentes clientes e o DASE. Existem somente dois tipos de agentes utilitários, *Request* e *Configure*. Agentes *Request*

facilitam a solicitação de acesso a dados, intermediando a interação dos agentes clientes com agentes *DASEServiceLocator* e *DASEDataProvider*. Agentes *Configure* facilitam a solicitação de configuração e início de ambientes de dados, intermediando a interação entre agentes clientes e o agente *DASEManager*. Tanto agentes *Request* quanto agentes *Configure* podem ser utilizados diretamente pelo usuário através da interface gráfica do JADE, chamada RMA, ou mesmo através do *DASEConfigurator*.

6.1.5 Relacionamento entre agentes DASE

A figura 17 traz um diagrama que representa o relacionamento entre os agentes descritos. A notação utilizada neste diagrama para o relacionamento entre agentes é semelhante à utilizada em um diagrama de classes UML. Este diagrama possui a seguinte convenção para os relacionamentos entre classes, que neste caso representam agentes:

- **Associação**: significa que os agentes relacionados não precisam estar necessariamente no mesmo *host* e que se comunicam sempre através de troca de mensagens ACL. Por este motivo cada um possui o AID do outro.
- **Agregação**: significa que os agentes relacionados estão necessariamente no mesmo nó, consequentemente na mesma máquina virtual Java, e cada um possui uma variável de referência de objeto ao outro. Eles se comunicam através de chamadas a métodos.
- **Composição**: considerando um efeito prático, possui a mesma característica atribuída à agregação, explicada no tópico anterior. Entretanto, assim como um diagrama de classes comum, uma composição é um relacionamento mais forte do que uma agregação, o que significa que um objeto faz parte da “composição” do outro, e não que simplesmente “agrega” um conjunto adicional de características.

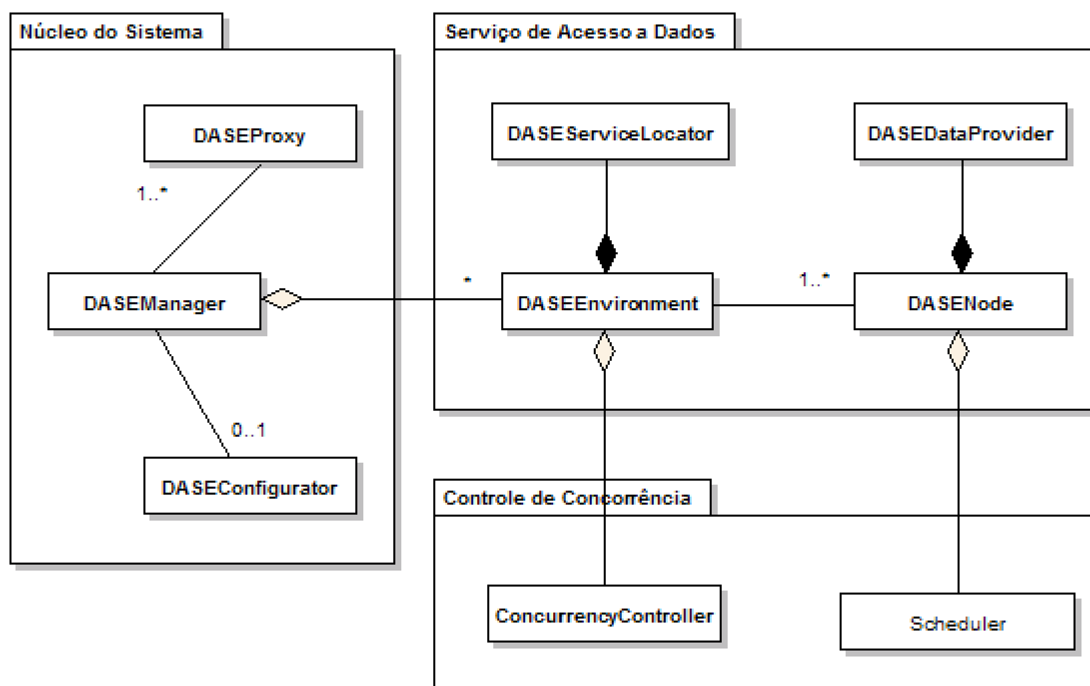

Figura 17 - Relacionamento entre agentes DASE.

A notação de multiplicidade existente no diagrama da figura 17 representa a quantidade máxima e mínima de agentes participantes em cada relacionamento, assim como em um diagrama de classes comum. A subseção 6.1.6 traz mais detalhes sobre a multiplicidade considerando a quantidade máxima e mínima de cada tipo de agente por nó.

6.1.6 Nomenclatura e multiplicidade dos agentes DASE

A tabela 1 apresenta todos os tipos de agentes DASE, além de sua nomenclatura e multiplicidade. A coluna nomenclatura descreve como cada agente será nomeado em tempo de execução.

Na maioria dos casos o nome do agente é formado a partir de seu tipo seguido de um texto curto que garante unicidade e um significado coerente com a natureza do agente. A coluna multiplicidade indica qual a quantidade máxima permitida de agentes de cada tipo no sistema. O termo nó, além de representar um *host* de todo o ambiente distribuído, representa exatamente um contêiner JADE em uma máquina virtual Java em um *host*.

Tabela 1 - Nomenclatura e multiplicidade dos agentes DASE.

Tipo	Nomenclatura	Multiplicidade
DASEProxy	DASEProxy-*hostname*	Nenhum ou 1 por nó
DASEManager	DASEManager	1 em todo o sistema
DASEConfigurator	DASEConfigurator	Nenhum ou 1 em todo o sistema
DASEEnvironment	DE-*environmentLabel*	Indefinido
DASEServiceLocator	SL-*environmentLabel*	1 para cada *DASEEnvironment*
DASENode	Nd[*n*]-*environmentLabel*	No mínimo 1 para cada *DASEEnvironment*
DASEDataProvider	DP[*n*]-*environmentLabel*	1 para cada *DASENode*
Scheduler	Sc[*n*]-*environmentLabel*	1 para cada *DASENode*
ConcurrencyController	CC-*environmentLabel*	1 para cada *DASEEnvironment*

O agente *DASEProxy* é responsável por interagir diretamente com os agentes clientes. Sua principal tarefa é permitir que o agente cliente tenha acesso a operações básicas, como o início ou encerramento de todo o sistema. Este agente comunica-se com agentes clientes através de chamadas a métodos Java, ou seja, localmente. Por este motivo, sempre que um agente cliente precisar dos serviços de um agente *DASEProxy*, tal agente deverá ser criado, caso ainda não exista, no nó em que está o agente cliente. Como não há necessidade de mais de um agente *DASEProxy* para atender solicitações de agentes clientes do mesmo nó, e como eventualmente em alguns nós pode ocorrer de nenhum agente cliente precisar de serviços de agentes *DASEProxy*, então a multiplicidade deste tipo de agente DASE é nenhum ou um por nó. O nome de um agente *DASEProxy* possui o nome de seu nó (*hostname*).

O agente *DASEManager* é o mais importante de todo o sistema. É responsável por tarefas como a validação e intermediação de criação de ambientes

de dados, configuração inicial do sistema e encerramento seguro do sistema. Ele possui contato com o agente *DASEConfigurator*, caso exista, com todos os agentes *DASEProxy* e com todos agentes *DASEEnvironment*, o que lhe garante acesso indireto a todos os agentes DASE. Assim, desconsiderando fatores relacionados a tolerância a falhas, não há necessidade de mais de um agente deste tipo em todo o sistema.

O agente *DASEConfigurator* tem como tarefa oferecer ao usuário uma interface gráfica que lhe permita configurar e monitorar todo o sistema. Por motivos de acesso concorrente, somente um agente deste tipo pode existir em todo o sistema. Mas esse tipo de restrição pode facilmente ser superada no futuro. O nó em que ele é criado é indiferente, já que tal agente comunica-se com os demais agentes DASE sempre através de mensagens ACL. A possibilidade de criação de mais de um agente deste tipo está incluída na lista de melhorias futuras do sistema.

Quando o DASE é iniciado não há nenhum ambiente de dados, mas o sistema já é capaz de atender a requisições de criação de ambiente de dados oriundas dos agentes clientes. Como um ambiente de dados é representado e controlado por um agente *DASEEnvironment*, e como o DASE não impõe quantidade máxima de ambientes de dados criados, então não há qualquer restrição quanto ao número de agentes *DASEEnvironment* existentes no sistema, podendo ser zero ou qualquer outro número, desde que haja recursos computacionais suficientes para permitir um comportamento aceitável do sistema. Agentes *DASEEnvironment* são identificados, e sua unicidade é garantida, através de um rótulo chamado *environmentLabel*.

Para que um agente cliente possa requisitar o acesso aos dados relacionados a um ambiente de dados, ele depende da intermediação deste serviço, realizada pelo agente *DASEServiceLocator*. Assim, todo ambiente de dados, ou agente *DASEEnvironment*, precisa estar atrelado a um agente *DASEServiceLocator*. Além disso, não há motivo para existir mais de um agente deste tipo para um mesmo agente *DASEEnvironment*, o que justifica sua multiplicidade descrita na tabela 1. Caso o controle de concorrência esteja ativado para cada agente *DASEEnvironment* haverá também um agente *ConcurrencyController*. Por questões de simplificação de projeto e desempenho, todo agente *DASEServiceLocator* está sempre no mesmo nó

em que está seu respectivo agente *DASEEnvironment*, conforme figura 17. O mesmo ocorre para o agente *ConcurrencyController*.

Ao ser configurado e iniciado, todo ambiente de dados precisa ter necessariamente no mínimo um nó, caso contrário ele não seria funcional. Um nó de um ambiente de dados é controlado e representado por um agente *DASENode*, o que explica a multiplicidade deste tipo de agente informada na tabela 1. Todo agente *DASENode* é responsável por controlar um agente *DASEDataProvider* e, caso o controle de concorrência esteja ativado, um agente *Scheduler*. A unicidade destes três tipos de agentes DASE é garantida através de suas nomenclaturas, formada pelo rótulo do ambiente de dados a que pertencem e um número seqüencial que identifica o nó do ambiente de dados. Cada agente *DASEDataProvider* e agente *Scheduler* estão sempre no mesmo nó em que está o agente *DASENode* relacionado.

6.2 Serviços Prestados pelos Agentes DASE

A arquitetura do DASE é composta por diversos agentes. Cada um desses agentes possui um objetivo específico, e para tal, realiza um conjunto de tarefas. No entanto, nenhum desses agentes é capaz de operar de modo isolado, tendo que interagir com outros agentes DASE. Esta interação ocorre normalmente através da prestação de serviços entre os agentes.

A tabela 2 relaciona todos os serviços prestados pelos agentes DASE. A coluna “agente solicitante” traz o agente que solicita o serviço. Há um único serviço em que o protocolo *FIPA Request* não é utilizado, dando lugar ao protocolo *FIPA Recruiting*. Isto ocorre no serviço *Data Request*. Neste caso o agente solicitante, um agente cliente, não possui contato direto com quem presta o serviço, um agente *DASEDataProvider*. Um agente intermediário, *DASEServiceLocator*, é responsável por selecionar ou encontrar quem prestará o serviço, agente *DASEDataProvider*, e encaminhar a solicitação do solicitante. E para que esse encaminhamento ocorra, um serviço intermediário é utilizado, chamado *Service Dispatch*. O resultado do serviço é entregue pelo agente servidor, *DASEDataProvider*, diretamente ao agente solicitante. Os protocolos FIPA de interação entre agentes utilizados durante o

projeto do DASE foram descritos com mais detalhes na seção 2.5. Os serviços *Service Dispatch* e *Data Request* estão fortemente relacionados e serão descritos detalhadamente nas seções 6.3 e 8.3.

Tabela 2 - Serviços prestados pelos agentes DASE.

Agente solicitante	**Agente intermediário**	**Serviço**	**Agente servidor**	**Protocolo**
DM	-	Kill itself	DX	FIPA Request
DX	-	DASEProxy AID registry	DM	FIPA Request
DX	-	System Shutdown	DM	FIPA Request
CA	-	Environment Creation	DM	FIPA Request
Nd	-	Node Abortion	DE	FIPA Request
Nd	-	Node Ready	SL	FIPA Request
CA	-	Service Dispatch	SL	FIPA Recruiting
DE	-	Kill itself	Nd	FIPA Request
CA	SL	Data Request	DP	FIPA Recruiting
Sc	-	Transaction Observer	DP	FIPA Request
Sc	-	Request Transaction Operation	CC	FIPA Request
Sc	-	Confirm Transaction Operation	CC	FIPA Request
Sc	-	Transaction Finished	CC	FIPA Request

A tabela 3 relaciona a cada serviço os parâmetros que especificam sua solicitação, o resultado esperado e uma constante de identificação, utilizada para identificá-lo. *EnvironmentDescriptor* e *ServiceDescriptor* são objetos DASE que representam respectivamente todas as características de um ambiente de dados e todos os detalhes referentes a uma requisição de acesso a dados. Já *SQL Sentence* é a sentença SQL utilizada em uma requisição de acesso a dados. *TransactionId* é

um identificador que garante unicidade entre as transações do sistema de controle de concorrência do DASE, além de guardar informações adicionais sobre cada transação. *TransactionOperation* representa uma operação de leitura ou escrita de uma transação. Os objetos da coluna resultado serão explicados nas seções 6.3, 6.4 e capítulo 8. Em seguida, a tabela 4 descreve cada um dos serviços mencionados nas tabelas 12 e 13.

Tabela 3 - Detalhes sobre os serviços prestados pelos agentes DASE.

Serviço	Parâmetros	Resultado	Constante
Kill itself	-	-	KILL
DASEProxy AID registry	-	-	SYSTEM_REGISTRY
System Shutdown	-	-	SHUTDOWN
Environment Creation	Environment Descriptor	-	ENVIRONMENT_CREATION
Node Abortion	-	-	NODE_ABORTION
Node Ready	-	-	NODE_READY
Service Dispatch	Service Descriptor	MessageTemplate	SERVICE_DISPATCH
Kill itself	-	-	KILL
Data Request	SQL Sentence	DASEResult	DATA_REQUEST
Transaction Observer	TransactionId	-	TX_OBSERVER
Request Transaction Operation	TransactionOperation	- | TransactionId	REQUEST_TX_OPERATION
Confirm Transaction Operation	-	-	CONFIRM_TX_OPERATION
Transaction Finished	TransactionId	-	TX_FINISHED

Tabela 4 - Descrição dos serviços prestados pelos agentes DASE.

Serviço	**Descrição**
Kill itself	Este serviço é prestado por dois tipos de agentes DASE, o *DASEProxy* e o *DASENode*. O *DASEManager* utiliza este serviço do *DASEProxy* para avisá-lo de que o sistema será encerrado, e portanto deve encerrar sua execução. De forma análoga, um agente *DASEEnvironment* utiliza este serviço de um agente *DASENode* para avisá-lo de que o ambiente de dados será encerrado e, consequentemente, todos os seus agentes *DASENode* deverão fazer o mesmo.
DASEProxy AID registry	Os agentes *DASEProxy* registram seu AID no *DASEManager* para que possam comunicar-se sempre que necessário. Por exemplo, quando o *DASEManager* utiliza o serviço *Kill itself* dos agentes *DASEProxy*.
System Shutdown	Serviço que permite a um agente *DASEProxy* encaminhar uma solicitação de um agente cliente ao agente *DASEManager* de encerramento do sistema.
Environment Creation	Este serviço é prestado pelo *DASEManager* aos agentes clientes, permitindo a solicitação de criação de novos ambientes de dados.
Node Abortion	Através deste serviço um agente *DASENode* pode avisar o agente *DASEEnvironment* de que alguma situação inesperada o impede de operar normalmente, e por isso deve ser abortado. Baseado neste aviso o *DASEEnvironment* pode até mesmo decidir por abortar o próprio ambiente de dados, caso não reste nenhum *DASENode* em condições de operar.
Node Ready	Este serviço permite que cada agente *DASENode* informe ao agente *DASEServiceLocator* que já estão aptos a receberem requisições de acesso a dados. Assim que o agente *DASEServiceLocator* recebe tal notificação de todos os nós ele publica seus serviços para que os mesmos possam ser utilizados pelos agentes clientes.
Service Dispatch	Um agente cliente utiliza este serviço para iniciar o processo de requisição de acesso a dados. Quem presta este serviço é o agente *DASEServiceLocator* do ambiente de dados cuja informação o agente cliente deseja acessar. O agente *DASEServiceLocator* definirá de qual nó virá o agente *DASEDataProvider* que proverá o serviço de acesso a dados. Isto é feito conforme descrito na seção 6.3, sobre balanceamento de carga. Como resposta o agente *DASEServiceLocator* envia ao agente cliente um objeto *MessageTemplate*, que permitirá ao agente cliente identificar a mensagem que contém o resultado da requisição de acesso a dados. Este objeto é um recurso do JADE, e seu uso é explicado com detalhes na subseção 8.3.3.
Data Request	Este é o serviço de requisição de dados. Tal serviço é sempre prestado a um agente cliente, e quem o oferece é um agente *DASEDataProvider*. Entretanto um agente *DASEServiceLocator* sempre intermedia tal solicitação.

Transaction Observer	Este serviço é oferecido a agentes *Scheduler* por agentes *DASEDataProvider* para evitar reinícios cíclicos. Para tal o agente *Scheduler* não reinicia a transação abortada enquanto a transação que causou o seu aborto não concluir.
Request Transaction Operation	Os agentes *Scheduler* utilizam este serviço, oferecido pelo agente *ConcurrenyController*, para obterem permissão de execução de uma determinada operação de uma transação. Enquanto a permissão não é concedida a operação não é executada.
Confirm Transaction Operation	Este serviço permite que um agente *Scheduler* confirme com o agente *ConcurrencyController* que uma determinada operação foi executada com sucesso. Caso algo inesperado ocorra durante a execução da operação, este serviço não será utilizado, e a notificação de aborto da transação será realizada através do serviço *Transaction Finished*.
Transaction Finished	Este serviço é oferecido pelo agente *ConcurrencyController* aos agentes *Scheduler* para que estes possam informá-lo sobre a conclusão de uma transação, seja ela aborto ou confirmação. Em seguida outras operações que dependiam da conclusão da transação recém confirmada poderão receber o direito de prosseguirem. O controle de concorrência do DASE é descrito com mais detalhes na seção 6.4.

A figura 18 ilustra a interação entre os agentes DASE, e agentes clientes também, considerando os seus serviços prestados. O destino da seta indica quem prestará o serviço, enquanto que a origem indica quem solicitou o serviço. Um losango interceptado por uma seta indica que tal serviço é intermediado por outro agente, identificado por uma seta tracejada.

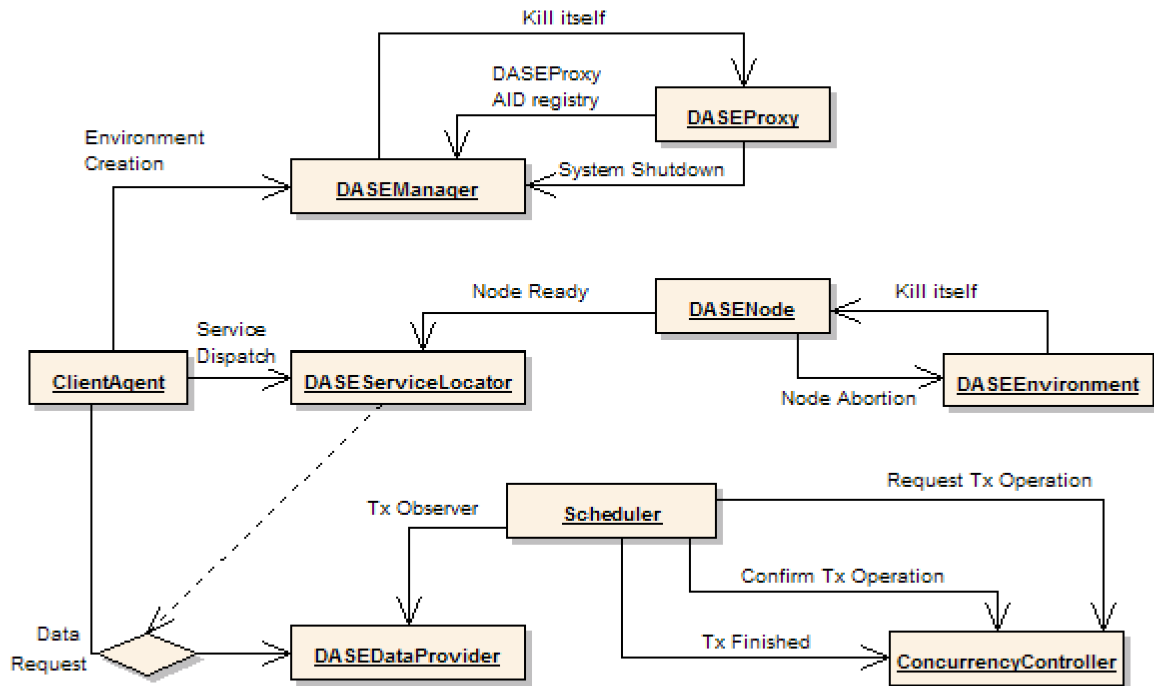

Figura 18 - Serviços prestados entre agentes DASE e entre agentes DASE e agentes clientes.

6.3 Balanceamento de Carga

O DASE possui balanceamento de carga, equilibrando as requisições recebidas a partir dos nós de um ambiente de dados. Este recurso é ativado e configurado através do ambiente de dados.

O balanceamento de carga do DASE funciona da seguinte forma: o agente cliente localiza através do agente JADE DF o agente *DASEServiceLocator* referente ao ambiente de dados a partir do qual deseja obter dados. O agente *DASEServiceLocator* encaminha a solicitação de acesso a dados do agente cliente a um agente *DASEDataProvider*, para que este lhe preste o serviço de requisição de dados. Cada nó do ambiente de dados possui um agente *DASEDataProvider*, e é através da escolha deste agente que o balanceamento de carga ocorre. Todo o processo de prestação de serviço do agente *DASEDataProvider* ao agente cliente, intermediado pelo agente *DASEServiceLocator*, ocorre de acordo com o protocolo *FIPA Recruiting*, descrito na subseção 2.5.5.

Para realizar o balanceamento de carga, o agente *DASEServiceLocator* não escolhe diretamente o agente *DASEDataProvider*, e sim um nó. Para isso três fatores são considerados:

1. Primeiro é dada prioridade a um contêiner indicado pelo agente cliente.
2. Caso o contêiner indicado não faça parte do ambiente de dados, então é dada prioridade ao próprio nó de onde partiu a requisição, ou seja, o nó onde está o agente cliente.
3. Caso o contêiner onde está o agente cliente também não faça parte do ambiente de dados, então um algoritmo de balanceamento de carga é adotado para que um determinado nó seja o indicado.

Na maioria dos casos o melhor nó a ser indicado é o nó em que está o agente cliente, pois isso evitaria uma sobrecarga com troca de mensagens adicionais na rede. No entanto, o agente cliente pode não estar em um nó participante do ambiente de dados. Assim, não é sempre que haverá um agente *DASEDataProvider* local ao agente cliente. Isto justifica a importância do balanceamento de carga e o papel do agente *DASEServiceLocator*.

Para aplicar um algoritmo de balanceamento de carga, é necessário que o agente *DASEServiceLocator* obtenha informações atualizadas e constantes sobre o estado dos agentes *DASENode*, ou seja, se cada nó do ambiente de dados está muito sobrecarregado ou não. Entretanto, o agente *DASEEnvironment* já possui parte dessas informações, além de possuir o AID de todos os agentes *DASENode*. Por isso, é na realidade o agente *DASEEnvironment* que executa o algoritmo de balanceamento de carga e indica ao agente *DASEServiceLocator* qual é o nó escolhido.

Atualmente só existe um algoritmo de balanceamento de carga implementado, chamado de *Round Robin*, ou circular. Tal algoritmo é extremamente simples e consiste em um revezamento seqüencial e circular dos nós que atenderão a requisições. Assim que o último nó for indicado, o primeiro será o próximo a receber uma indicação, em seguida o segundo e assim sucessivamente.

A figura 19 ilustra a arquitetura do sistema considerando um ambiente de dados disposto sobre dois nós, separados na figura através de uma linha tracejada vertical. Nesta figura os círculos representam agentes, com exceção de “CP” e ”DB”,

que são objetos e representam respectivamente *ConnectionProvider* e o *DataBridge*, ambos descritos na seção 6.1. O agente "AC" é um agente cliente, enquanto que todos os demais agentes são agentes DASE. Os agentes JADE foram omitidos nesta figura apenas para simplificar a ilustração.

Os retângulos representam o conjunto de agentes DASE, um em cada nó. Os dois tambores, na parte inferior da figura, representam dois nós do SBD distribuído acessado por agentes clientes através do DASE. As setas representam em seqüência a comunicação entre os agentes durante uma requisição de dados realizada por um agente cliente.

Algumas setas são bidirecionais, portanto são numeradas duas vezes. Esta figura ilustra um caso em que o balanceamento de carga do DASE está ativado e o controle de concorrência está desativado, por isso a ausência de agentes *ConcurrencyController* e *Scheduler*. A tabela 5 explica o que cada seta significa.

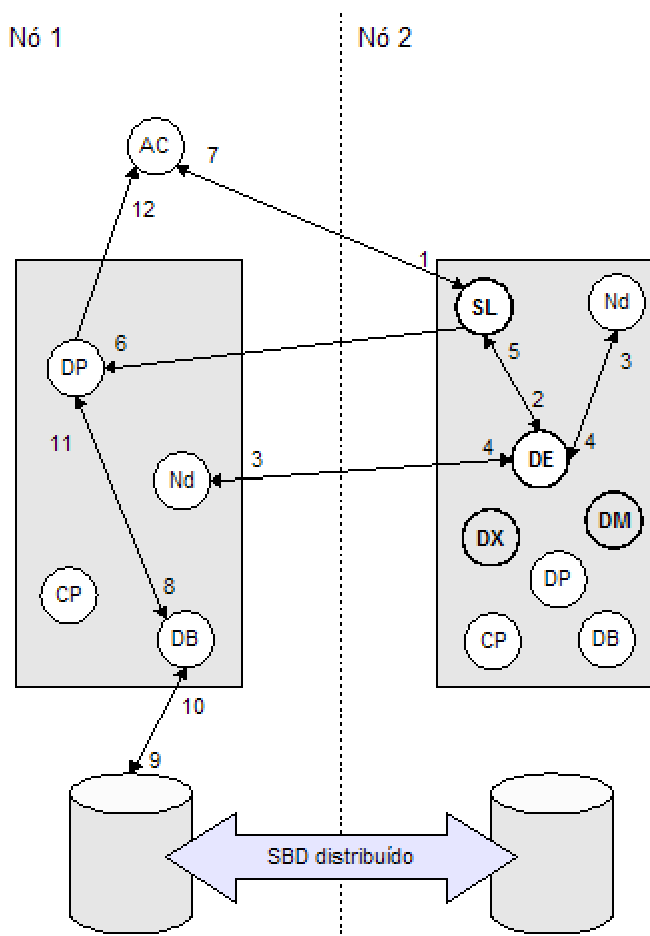

Figura 19 - Arquitetura DASE com dois nós.

Tabela 5 - Explicação das setas da figura 19.

Seta(s)	Explicação
1 e 7	Agente cliente solicita serviço *Service Dispatch* ao agente *DASEServiceLocator*(seta 1). Assim, o agente cliente obtém o objeto *MessageTemplate* (seta 7) necessário para identificar a resposta da requisição de acesso a dados (*Data Request*) a ser realizada em seguida.
2, 3, 4 e 5	Indicam a interação entre o agente *DASEServiceLocator*, o agente *DASEEnvironment* e um ou mais agentes *DASENode*. Tal interação ocorre devido ao balanceamento de carga.
6 e 12	A seta de número seis representa o papel intermediário do agente *DASEServiceLocator* durante a execução do serviço *Data Request* (requisitado indiretamente pelo agente cliente), enquanto que a seta de número 12 representa o resultado da prestação deste serviço, obtido pelo agente *DASEDataProvider*.
8, 9, 10 e 11	Tais setas ilustram o trabalho do agente *DataBridge* a partir de uma solicitação do agente *DASEDataProvider*, considerando um ambiente de dados em que o controle de concorrência está desativado. Considera-se que o objeto *DataBridge* já havia obtido previamente uma conexão com o SBD, graças ao auxílio do objeto *ConnectionProvider*.

Duas observações são importantes quanto ao balanceamento de carga do DASE:

- **Indicação de nós**: o agente *DASEServiceLocator* aceita requisições de agentes clientes contendo a indicação de um nó. Assim, sempre que o agente cliente indicar um nó juntamente com a requisição de acesso a dados, e tal nó pertencer de fato ao ambiente de dados, então o nó escolhido será o indicado pelo agente cliente, anulando, ou condicionando, o balanceamento de carga.
- **Técnica de distribuição de dados**: O balanceamento de carga do DASE considera que a localidade física dos dados distribuídos (replicados e segmentados) ao longo dos nós do SBD não é importante, o que ocorre quando a distribuição é feita utilizando distribuição circular (*round robin*), por *hashing*, ou qualquer outra técnica que não considere a semântica dos dados durante a

distribuição, como por faixa de valores. Esta restrição é importante porque não será vantajoso direcionar carga a um determinado nó se ele não tiver as melhores condições de acesso aos dados, ou mesmo se ele não contiver os dados. Pelo fato de o DASE não conhecer detalhes sobre a localidade dos dados do SBD[8], a técnica de balanceamento de carga circular torna-se a mais adequada, independentemente de ser a única disponível atualmente. Entretanto, conforme dito no item anterior, existe a possibilidade de as requisições serem processadas no nó em que foram submetidas, o que faria com que a responsabilidade de escolha do melhor nó fosse transferida aos agentes clientes.

6.4 Controle de Concorrência

O controle de concorrência garante a consistência dos dados mesmo que diversas transações os acessem ao mesmo tempo. Quando considerado um SBD distribuído o controle de concorrência passa a ter complexidade adicional.

O DASE possui controle transacional de dois modos diferentes:

1. Encaminhamento de transações ao SBD: o SBD oferece controle de transações e irá recebê-las, por isso o DASE simplesmente as encaminha. Este modo, também chamado de *FORWARD*, na realidade equivale a dizer que o controle de concorrência do DASE está desabilitado, além de ser o modo padrão de controle de concorrência do DASE.

2. Controle de concorrência do DASE: cada transação recebida pelo DASE será processada garantindo acesso consistente aos dados. Atualmente o único método de controle de concorrência implementado para o DASE é o *Timestamps*[9], descrito na seção 3.7. Por este motivo este modo será referenciado deste ponto do texto em diante como *TIMESTAMPS*.

[8] Para interagir com o SBD o DASE utiliza *drivers* JDBC, como qualquer aplicação Java. Entretanto o JDBC não foi projetado para atender Sistemas de Banco de Dados distribuídos, e sim somente centralizados.

Independentemente de quais dos modos acima for o utilizado, os seguintes elementos participam do controle de concorrência: *DASEDataProvider* e *DataBridge*. Os seus papéis para cada um dos dois modos, *FORWARD* ou *TIMESTAMPS*, diferem e serão descritos nas seções a seguir.

6.4.1 Modo FORWARD

Neste modo *DASEDataProvider* e *DataBridge* atuam da seguinte forma:

- ***DASEDataPovider***: atua como despachante de requisição de acesso a dados. O conteúdo das requisições que recebe de agentes clientes é indiferente, pois todas são repassadas integralmente ao *DataBridge*.
- ***DataBridge***: atua como gerenciador de dados. Recebe do agente *DASEDataProvider* em uma única chamada um conjunto completo de instruções SQL, presentes na requisição do agente cliente, e as repassa ao SBD, o que é possível graças ao *driver* JDBC. Os resultados são retornados diretamente ao agente *DASEDataProvider*, que os encapsula em um objeto de resposta do tipo *DASERequestAnswer*, que é encaminhado ao agente cliente.

6.4.2 Modo TIMESTAMPS

Neste modo as requisições de acesso a dados podem possuir transações do usuário, sentenças independentes de acesso a dados, ou ambos ao mesmo tempo.

Transações do usuário são conjuntos de sentenças SQL delimitadas pelo usuário e definidas dentro da cadeia de caracteres contida na requisição de acesso a dados. Este tipo de transação é tratado pelo DASE de acordo com as regras ACID, descrita na subseção 3.3.1, além de obedecer ao padrão SQL (Gulutzan e Pelzer, 1999).

São consideradas sentenças independentes de acesso a dados qualquer sentença SQL presente em uma requisição de acesso a dados que não tenha sido definida dentro dos delimitadores de uma transação do usuário. A figura 20 ilustra

---

[9] O motivo da adoção deste método de controle de concorrência é explicado na seção 7.5

uma requisição de acesso a dados que contém uma sentença independente, neste caso do tipo *INSERT*, e uma transação do usuário.

```
INSERT INTO contas VALUES(2, 300.50);

BEGIN TRANSACTION;
        UPDATE contas
                SET saldo=saldo+(SELECT saldo FROM contas WHERE id=0)
                WHERE id=1;
        UPDATE contas
                SET saldo=0 WHERE id=0;
COMMIT;
```

Figura 20 - Exemplo de uma requisição de acesso a dados.

Para cada transação do usuário e para cada sentença independente de acesso a dados uma transação do sistema será criada. Transação do sistema é um objeto *DASETransaction* que contém cada sentença SQL da transação do usuário, representando as operações de leitura e escrita da transação, ou a única sentença SQL de uma sentença independente de acesso a dados. Além disso, este objeto contém também meta-dados específicos do método de controle de concorrência adotado, auxiliando o processamento e identificação da transação. No exemplo da figura 20 duas transações do sistema seriam criadas, uma contendo a sentença *INSERT* e outra contendo todas as operações da transação do usuário contida ali.

O agente *DASEDataProvider* é o responsável por criar as transações do sistema. Assim, toda requisição de acesso a dados será transformada em uma ou mais transações do sistema.

No modo *TIMESTAMPS* de controle de concorrência além do agente *DASEDataProvider* e do objeto *DataBridge* há dois novos elementos, os agentes *Scheduler* e *ConcurrencyController*. Os quatro elementos atuam da seguinte forma:

- ***DASEDataPovider***: atua como gerenciador de transações. Tem a responsabilidade de identificar o conteúdo da requisição de acesso a

dados de modo a construir uma transação do sistema para cada transação do usuário, e uma transação do sistema para cada sentença independente de acesso a dados contidos ali. Ao receber individualmente os resultados de cada operação das transações do sistema, entregues pelo agente *Scheduler*, agrupa-os e constrói um objeto de resposta do tipo *DASERequestAnswer*. Através deste objeto os dados solicitados pelo agente cliente, ou mesmo informações referentes a um ou mais abortos caso ocorram, são apresentados de forma clara e organizada.

- ***Scheduler***: promove a integração entre o gerenciador de transação (agente *DASEDataProvider*) e o escalonador (agente *ConcurrencyController*). Há um em cada nó do ambiente de dados. Recebe as transações do sistema do agente *DASEDataProvider*, interage com o agente *ConcurrencyController* de modo a garantir a execução correta da transação, e repassa todas as operações das transações do sistema recebidas ao objeto *DataBridge*, responsável por efetivá-las junto ao SBD. Este agente ainda notifica o agente *DASEDataProvider* e o agente *ConcurrencyController* sempre que um aborto é necessário, seja ele oriundo de uma violação do método de controle de concorrência ou resultado do processamento pelo SBD de alguma sentença SQL. Tal agente ainda trabalha de modo a evitar a ocorrência de reinícios cíclicos de transações.

- ***ConcurrencyController***: atua como escalonador de transações e há um por ambiente de dados. Seu objetivo é gerar escalonamentos garantindo que as transações do sistema sejam executadas paralelamente, mas principalmente sempre mantendo um estado seriável, conforme explicado na subseção 3.3.3. Atinge esse objetivo escalonando as operações das transações através da execução do método de controle de concorrência *Timestamps*. Recebe requisições de execução de operações e verifica se há permissão para que determinada operação seja executada sem pôr em risco a consistência dos dados. Sempre que uma violação das regras de marcas de tempo ocorre, vide subseção 3.7.1, impede que a operação seja executada e

solicita ao agente *Scheduler* que a transação correspondente seja abortada e reiniciada, além de fornecer a este agente todas as informações necessárias como, por exemplo, a marca de tempo da transação que ocasionou o aborto. É importante destacar que neste caso o agente *Scheduler* irá apenas abortar a transação, pois quem tratará do seu reinício será o gerenciador de transações, ou seja, o agente *DASEDataProvider*.

- ***DataBridge***: atua como gerenciador de acesso a dados e gerenciador de recuperação. Enquanto no modo *FORWARD* o *DataBridge* recebia um conjunto completo de sentenças SQL do agente *DASEDataProvider* e as encaminhava integralmente ao SBD, no modo *TIMESTAMPS* ele recebe do agente *Scheduler* cada operação de cada transação do sistema e as processa individualmente, inclusive tratando-as de modo diferente de acordo com sua natureza, como por exemplo quanto ao fato de serem de leitura ou de escrita. Sempre que o *DataBridge* concluir o repasse de uma instrução SQL ao SBD, ele avisará o agente *Scheduler* para que este fique ciente de que tal instrução, que certamente compõe uma transação, já foi concluída e se obteve sucesso ou não. Caso alguma operação isolada não possa ser executada por qualquer motivo inerente à sentença SQL, portanto independentemente do método de controle de concorrência, então tal objeto informa a ocorrência ao agente *Scheduler*, que abortará imediatamente a transação e lançará uma exceção permitindo que o agente *DASEDataProvider* possa encapsular a ocorrência e o motivo do aborto em um objeto de resposta *DASERequestAnswer*, que será enviado ao agente cliente. Tal transação não será reiniciada. O agente *Scheduler* ainda notifica o agente *ConcurrencyController* para que este possa considerar esta ocorrência durante a aplicação do método de controle de concorrência no escalonamento das transações.

É importante relembrar que caso o controle de concorrência esteja desativado, modo *FORWARD*, então o objeto *DataBridge* relacionar-se-á diretamente com o agente *DASEDataProvider*, ao invés de o fazê-lo com o agente *Scheduler*.

No modo *TIMESTAMPS* os abortos de transações ocorrem por dois motivos diferentes:

1. Devido a uma violação de alguma regra inerente ao processamento da sentença SQL pelo SBD, como, por exemplo, erros de sintaxe ou violações de integridade referencial.
2. Devido a uma violação da regra de marcas de tempo.

No primeiro caso uma exceção é lançada pelo *driver* JDBC e é capturada pelo objeto *DataBridge*, que a encaminha ao agente *Scheduler,* que interrompe a submissão das operações subseqüentes da mesma transação, aborta-a e re-encaminha a exceção ao agente *DASEDataProvider*. Este agente encapsula a ocorrência de aborto e os detalhes de sua ocorrência em um objeto *DASERequestAnswer*, e o encaminha ao agente cliente sem reiniciar ou re-submeter a transação abortada.

No segundo caso quem detecta o evento é o agente *ConcurrencyController*, que informa o fato ao agente *Scheduler*, que interrompe a submissão das operações subseqüentes, aborta a transação e lança uma exceção ao agente *DASEDataProvider*. Este agente cria e submete uma nova transação com as mesmas operações, mas seguindo as regras do método de controle de concorrência em questão, ou seja, atribuindo uma marca de tempo maior do que a da transação que ocasionou o seu aborto.

Portanto, caso o controle de concorrência esteja desativado, o agente *DASEDataProvider* sempre receberá uma resposta diretamente do *DataBridge*, contendo ou não resultados. Caso contrário, o agente *Scheduler* intermediará esta relação de modo a garantir a consistência dos dados enquanto relaciona-se com o agente *ConcurrencyController*, que realiza o escalonamento das operações de todas as transações do sistema para um ambiente de dados em específico. Para os dois casos, enquanto aguarda o resultado o agente *DASEDataProvider* permanece em estado bloqueado.

O agente *Scheduler* recebe as transações do sistema através do agente *DASEDataProvider*. O agente *Scheduler* atua como o cliente do agente *ConcurrencyController*, que é de fato o escalonador de transações escolhendo a melhor ordem para execução das operações de todas transações do sistema. Em

seguida, o agente *Scheduler* repassa as operações ao *DataBridge*. Cada agente *Scheduler* de um nó contribui para o controle de concorrência ao trabalhar em conjunto com o agente *ConcurrencyController* do mesmo ambiente de dados.

6.4.3 Ambientes de dados diferentes acessando os mesmos dados

Conforme já afirmado anteriormente o DASE permite diversos ambientes de dados operando ao mesmo tempo. No entanto, é necessário ressaltar que o controle de concorrência do DASE é sempre referente a um ambiente de dados, ou seja, caso mais de um ambiente de dados tenha acesso a dados em comum, o DASE não evitará nem controlará quando operações conflitantes, mas de transações de ambientes de dados diferentes, ocorrerem.

Esse detalhe certamente coloca em risco a consistência dos dados, no entanto a única forma de evitar esse tipo de situação seria o uso de um sistema de controle de concorrência nativo ao SGBD, já que mesmo que o DASE identificasse e controlasse conflitos a partir de ambientes de dados diferentes, ainda assim o risco existiria, pois o DASE não seria capaz de evitar que, por exemplo, qualquer outra aplicação qualquer externa e independente do DASE acesso os mesmos dados colocando em risco sua consistência.

Ainda assim o controle de concorrência para ambientes de dados simultâneos é considerado um importante item na lista de melhorias futuras para o DASE. Basicamente este recurso não foi implementado até o momento, pois traria complexidade relativamente alta ao projeto e implementação do sistema, e tal esforço não compensaria, já que o escopo atual do projeto do DASE já atende suficientemente todos os requisitos a que se propôs inicialmente.

6.4.4 Escalonamento centralizado

É importante destacar que o controle de concorrência oferecido pelo DASE não é de fato distribuído, pois não utiliza transações distribuídas. Somente a captação de transações e o processamento de suas operações são distribuídos, mas o escalonamento é centralizado. Outra forma de constatar tal afirmação é observar que existe um agente *DASEDataProvider* (gerenciador de transações) e um objeto

DataBridge (gerenciador de dados) para cada nó do ambiente de dados, entretanto existe somente um agente *ConcurrencyController* (escalonador de transações) para todo o ambiente de dados.

Em outras palavras, os gerenciadores de transações e os gerenciadores de dados do DASE são distribuídos, mas seu escalonador de transações é centralizado. Isso ocorre porque o DASE não é capaz de identificar transações distribuídas, uma vez que o JDBC não permite o acesso à localidade física dos dados em relação aos nós do SBD distribuído. Aliás, o JDBC inclusive não foi projetado considerando SBD distribuídos, e sim somente os centralizados. Assim, não é possível que os dados e as operações das transações que atuam sobre eles sejam relacionados a cada nó ao longo de todo o sistema distribuído. Portanto, devido a esta limitação do JDBC, o DASE relaciona-se com o SBD distribuído como se o mesmo fosse um grande SBD centralizado, mas com pontos de acesso para conexão distribuídos.

A figura 12 ilustra a arquitetura de um SGBD que possui sistema de controle de concorrência distribuído. A figura 21 traz uma figura semelhante, porém representando a arquitetura de controle de concorrência oferecida pelo DASE, descrita nesta subseção. É possível notar a diferença entre as duas figuras claramente, e a partir disso é possível constatar que o controle de concorrência do DASE de fato não é distribuído, embora não seja verdade também afirmar que se trata de um sistema de controle de concorrência centralizado.

As linhas tracejadas representam a interação entre os agentes *Scheduler* de cada nó do ambiente de dados e o agente *ConcurrencyController*, que está no mesmo nó em que está o agente *DASEEnvironment*, embora não representado na figura.

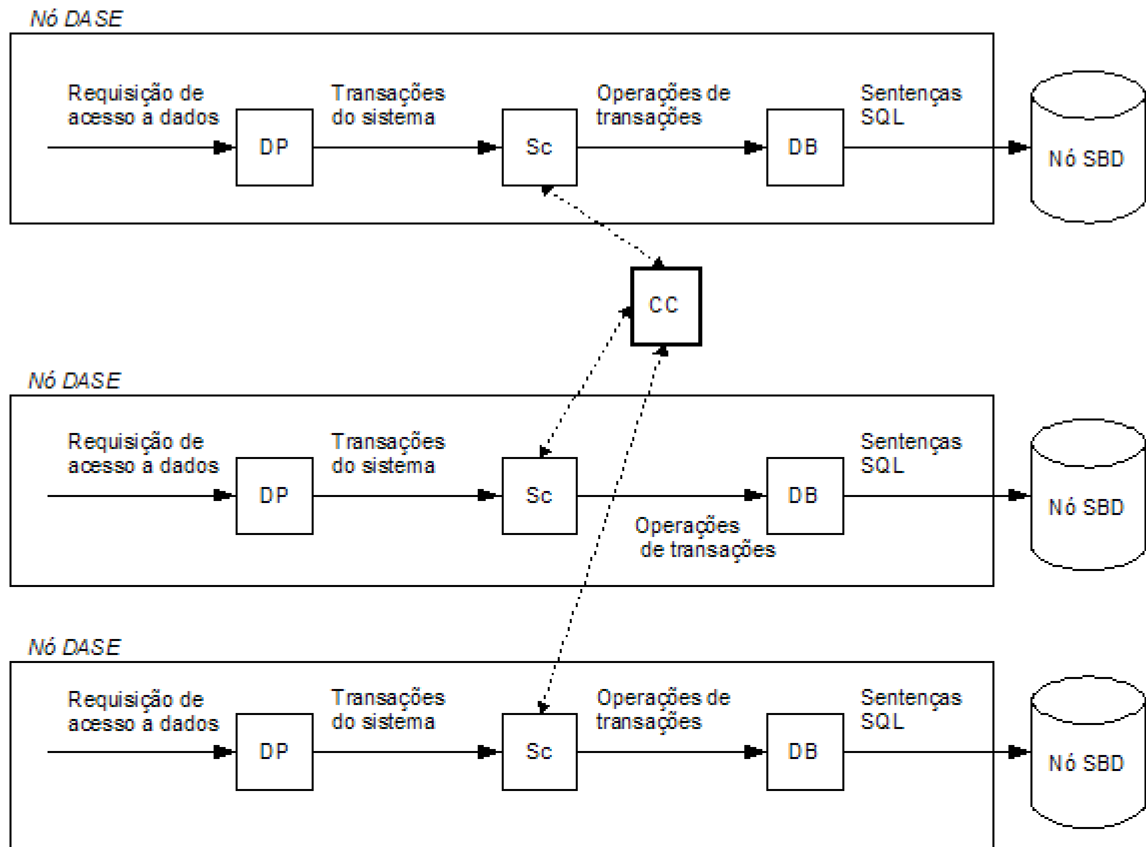

Figura 21 - Arquitetura do controle de concorrência do DASE.

6.4.5 A escolha do método de controle de concorrência

Conforme abordado nesta seção o controle de concorrência do DASE é baseado no método *Timestamps*, já descrito com detalhes na seção 3.7. No entanto este mesmo capítulo também descreveu outro importante método de controle de concorrência, o 2PL. Estes dois métodos foram analisados e para que um fosse o escolhido a ser implementado para o DASE.

Os motivos que fizeram com que o método escolhido fosse o *Timestamps* foram:

- Ausência de travamentos (*deadlocks*). Seria extremamente custoso em termos de recursos computacionais ter um mecanismo que prevenisse, identificasse e removesse travamentos.
- O projeto e implementação de um mecanismo para lidar com travamentos traria complexidade adicional ao projeto.

- Maior compatibilidade com escalonamentos estritos, já que o método *Timestamps* sempre gera escalonamentos seriáveis.

Vale destacar que a implementação do método 2PL exigiria o uso de bloqueios, no entanto a implementação do método *Timestamps* estrito, que é o caso do DASE, também faz uso de bloqueios para manter as filas de operações em execução, conforme explicado na subseção 3.7.2.

6.4.6 Granularidade do controle de concorrência

A granularidade do controle de concorrência basicamente reflete em quais elementos do SBD os bloqueios atuarão para que os dados mantenham-se consistentes durante a execução de cada operação. Estes elementos podem ser extremamente genéricos, abrangendo uma grande parte dos dados, ou podem ser mais restritos, abrangendo uma pequena porção de dados de forma mais precisa. O primeiro caso é chamado de granularidade grossa, enquanto que o segundo é chamado granularidade fina.

Quanto mais grossa for a granularidade maior será a quantidade de dados bloqueados e menor será o desempenho do sistema, pois uma quantidade menor de dados estará disponível para acesso concorrente. Quanto mais fina for a granularidade menor será a quantidade de dados bloqueados, o bloqueio será mais preciso, pois identificará com mais precisão os dados sobre os quais realmente há potencial de conflitos, e conseqüentemente haverá uma maior porção de dados ainda disponível para o acesso concorrente, o que certamente trará maior desempenho ao sistema.

Obviamente, um sistema de controle de concorrência com granularidade mais fina produziria uma sobrecarga de processamento maior do que um sistema que usasse uma granularidade mais grossa. No entanto, isso não seria suficiente para impedir que ainda assim o desempenho com uso de uma granularidade mais fina fosse maior.

Normalmente o elemento a ser bloqueado em casos de granularidade grossa é a tabela, enquanto que para granularidade média este elemento é a linha da tabela, e para granularidade fina este elemento é a coluna de cada linha da tabela.

O sistema de controle de concorrência projetado e implementado para o DASE, baseado no método *Timestamps*, utiliza granularidade grossa por um motivo simples: esta decisão requer uma complexidade menor de implementação. No entanto é importante destacar que o DASE foi projetado para ser facilmente estendido e oferecer uma granularidade média ou até mesmo fina, do mesmo modo que o DASE foi projetado para futuramente oferecer outros modos de controle de concorrência diferentes do *Timestamps*.

6.5 Controle de Recuperação

O sistema de controle de recuperação está intimamente relacionado ao controle de concorrência. Aliás, não há como um sistema de controle de concorrência ser de fato funcional caso não tenha o auxílio de um sistema de recuperação, caso contrário não seria possível abortar transações e retornar o SBD a um estado consistente.

O elemento responsável por realizar o controle de recuperação, ou inibi-lo quando desativado, é o *DataBridge*. E isto é feito de modo absolutamente transparente, o que significa que agentes como *DASEDataProvider* e *Scheduler* não alteram em nada sua maneira de operar de acordo com modo de controle de recuperação executado pelo *DataBridge*, ou mesmo quando o controle de recuperação do DASE está simplesmente desativado.

O DASE possui os seguintes modos de controle de recuperação: *FORWARD*, *JDBC CONNECTION* e *UNDO/NO REDO*. Os três são explicados com detalhes nas subseções a seguir.

6.5.1 Modo FORWARD

Neste modo o sistema de controle de recuperação do DASE está na verdade desabilitado, e todas as operações de transações enviadas pelos agentes clientes são encaminhadas e controladas completamente pelo SBD, mesmo se um aborto e conseqüentemente a recuperação forem necessários. Assim o DASE recebe as transações, mas é o SBD que se responsabiliza por processá-las e realizar eventuais recuperações.

Este tipo de controle de recuperação está intimamente relacionado ao modo *FORWARD* de controle de concorrência, tanto que ambos possuem o mesmo nome. Além disso, este modo de controle de recuperação só pode ser utilizado com o modo *FORWARD* de controle de concorrência, e vice-versa.

6.5.2 Modo JDBC CONNECTION

Neste modo o controle de recuperação do DASE está ativado e é baseado no desligamento de um recurso do JDBC chamado modo de confirmação automático. Neste caso o DASE passa a obter o controle total do driver JDBC sendo capaz de submeter comandos de confirmação e aborto (*rollback*) explicitamente sempre que necessário para que a recuperação e a manutenção de um estado consistente do SBD ocorram.

Outro detalhe sobre este modo é a necessidade de alteração do nível de isolamento de transações (*connection's transaction isolation level*) para "transações seriáveis" (*TRANSACTION_SERIALIZABLE)* pelo DASE. Isto significa que o SBD impedirá que uma transação leia dados escritos por outra transação que ainda não tenha se confirmado, o que caracteriza um comportamento estrito, independentemente do método de controle de concorrência adotado, conforme descrito no capítulo 4. Portanto neste modo quem na realidade "desfaz" o resultado da execução de uma operação é o próprio SBD, no entanto os eventos que fazem com que isso ocorra são gerenciados e disparados pelo DASE.

Esta técnica é a mais eficiente e é a mesma utilizada por ferramentas de mapeamento objeto-relacional. A única desvantagem desta técnica é que ela não pode ser aplicada caso o SGBD em questão não for transacional, em outras palavras, caso ele não possua um sistema de controle de concorrência. Assim, a única razão para que este modo não seja utilizado é quando o SGBD não é transacional e não oferece nível de isolamento para transações seriáveis.

Este tipo de controle de recuperação pode ser utilizado com qualquer modo de controle de concorrência do DASE, com exceção do modo *FORWARD*, pois neste caso não haveria como o DASE gerenciar a recuperação de transações sendo que nem mesmo as transações seriam visíveis ou controladas pelo DASE.

6.5.3 Modo UNDO/NO REDO

Neste modo o controle de recuperação do DASE está habilitado e opera baseado em uma técnica chamada UNDO/NO REDO. A idéia principal desta técnica é efetivar o resultado gerado por cada operação diretamente na base de dados, independentemente de a confirmação da transação já ter ocorrido ou não. Assim, caso seja necessário o aborto da transação por qualquer motivo, seja por violação do método de controle de concorrência ou por qualquer motivo inerente à sentença SQL, então as modificações realizadas por cada operação da transação já submetida devem ser desfeitas, *UNDO*. Mas, quando a transação confirma com sucesso, ou seja, sem a necessidade de aborto, nada é preciso ser feito para que o efeito de tal transação seja persistido no banco de dados, *NO REDO*.

Para a execução desta técnica, o *DataBridge* utiliza uma pilha que mantém um conjunto de operações que representam o inverso das operações já processadas da transação corrente. A tarefa de manter esta pilha não é tão complexa para o *DataBridge* porque ele sempre processa somente operações da uma mesma transação. Quando uma transação conclui-se, a pilha é limpa e as operações da transação seguinte são processadas.

Sempre que há um aborto as operações já processadas são desfeitas através da execução das operações inversas contidas na pilha. Quando há a confirmação não é necessário que as modificações no banco de dados sejam efetivadas, pois tais modificações já se tornaram persistentes desde o momento que ocorreram, antes mesmo de a transação concluir-se.

Diferentemente da técnica *JDBC CONNECTION*, esta possui a vantagem de poder ser usada com qualquer SGBD, mesmo os que não são transacionais. Apesar disso, esta técnica é menos eficiente e segura do que a *JDBC CONNECTION*, e deve ser utilizada somente em casos em que o SGBD adotado não é transacional.

Ela é menos eficiente por persistir os dados ao menos o dobro de vezes quando ocorrem abortos, uma vez para o processamento de uma operação e uma segunda vez para o processamento de sua operação inversa. Em alguns casos uma operação pode gerar mais de uma operação inversa, o que tornaria a recuperação ainda mais custosa.

E ela é menos segura, pois não é capaz de realizar recuperação em casos diferentes dos relacionados a exceções de software, como falhas de hardware e quedas de energia por exemplo.

Com o modo de recuperação *JDBC CONNECTION* o DASE controlaria o controle de recuperação do SGBD. O SGBD provavelmente manteria em memória as modificações referentes à transação antes de persisti-las de fato. Por isso o aborto da transação, para este caso, seria simplesmente o descarte das modificações em memória, já que os dados no disco permaneceriam inalterados, ou seja, coerentes com o estado anterior, e estável, do banco de dados. Esta técnica, mais complexa, porém mais eficiente, chama-se *NO UNDO/REDO* (Gulutzan e Pelzer, 1999).

O modo *UNDO/NO REDO* de controle de recuperação do DASE pode ser utilizado com qualquer modo de controle de concorrência, com exceção do modo *FORWARD*. Isso ocorre pelo mesmo motivo que não é possível utilizar o modo *JDBC CONNECTION* com o modo *FORWARD* de controle de concorrência.

A tabela 6 relaciona os modos de controle de recuperação compatíveis com cada modo de controle de concorrência.

Tabela 6 - Compatibilidade entre modos de controle de concorrência e modos de controle de recuperação.

Modo de controle de concorrência	Modos de controle de recuperação compatíveis
FORWARD	FORWARD
TIMESTAMPS	JDBC CONNECTION UNDO/NO REDO

7 ASPECTOS DE PROJETO E IMPLEMENTAÇÃO

Neste capítulo alguns aspectos importantes de projeto e implementação são discutidos e descritos, ilustrando decisões de projeto e como algumas dificuldades importantes foram superadas. São discutidas a seguir questões como critérios utilizados para definir agentes, aspectos de funcionamento e de interação entre agentes, a melhor forma de representar o resultado de uma requisição de acesso a dados, e pontos importantes sobre controle de concorrência.

7.1 Critério Utilizado para Definir Agentes

O JADE, plataforma de agentes adotada para o desenvolvimento do DASE, é um ambiente orientado a objetos, pois foi implementado através do uso de linguagem de programação orientada a objetos Java.

Com exceção de alguns tipos de dados primitivos como inteiros e caracteres, todos os demais tipos de dados em Java são representados como objetos, e o mesmo ocorre para os agentes do JADE. Mas, nem todo objeto Java criado sobre o JADE precisa ser necessariamente um agente, o que permitiu considerar de modo conveniente quando definir um elemento DASE como agente JADE, ou simplesmente um objeto Java.

Esta seção tem como objetivo esclarecer qual foi o critério utilizado durante o projeto do DASE para definir o que seria simplesmente um objeto, e o que seria um agente.

Os agentes desenvolvidos sobre a plataforma JADE, e de acordo com o padrão FIPA, comunicam-se exclusivamente via troca de mensagens. O JADE é um *middleware*, pois atua como um ambiente de execução, e é um sistema orientado a objetos, já que é implementado através de linguagem de programação Java. Entretanto, o JADE não é um *middleware* orientado a objetos, como os implementados a partir da especificação CORBA (Vinoski, 1997). É importante observar a correta aplicação dos termos neste momento.

Portanto, o uso de ORB (*Object Request Brokers*) não ocorre no JADE, já que não existe *proxy* para agentes, diferentemente do que ocorre com *middlewares* orientados a objetos, em que há *proxy* para objetos remotos. Tal característica do JADE não permite o manuseio de objetos de forma remota com transparência de localidade.

O paradigma adotado pelo JADE está mais próximo do modelo de troca de mensagens do que de memória compartilhada. Embora tal afirmação pareça óbvia, já que se trata de um sistema distribuído, na realidade não é, pois é possível notar, por exemplo, que um *middleware* orientado a objetos funciona de forma semelhante a um sistema DSM (*Distributed Shared Memory*) (Tanenbaum, 2003). Em um *middleware* orientado a objetos há acesso a memória distribuída de modo transparente, o que é possível graças à localização de objetos remotos e uso de *proxies*.

Portanto, partindo dessas premissas e considerando o uso do JADE sem subutilizar o paradigma de agentes, um mecanismo robusto, eficiente, e bem definido de interação entre os agentes através de mensagens ACL deve ser sempre necessariamente projetado, e a comodidade do uso de *proxy* deve ser ignorada.

Concluindo, todos os objetos DASE que deveriam ser acessados remotamente ao longo do sistema foram projetados como agentes, pois esta é a única forma natural, dentro do paradigma de agentes e considerando o ambiente de execução oferecido pelo JADE, de permitir a comunicação inter-processos. Além disso, vale destacar que é importante que o paradigma de agentes seja valorizado e utilizado, principalmente em favorecimento da qualidade do projeto, o que implica diretamente em flexibilidade e manutenibilidade do sistema. Portanto, este foi o critério para definir o que deveria ser somente objeto e o que deveria ser objeto e agente durante o projeto do DASE.

A seção 6.1 descreveu os principais agentes e objetos do sistema. Tanto o *DataBridge* quanto o *ConnectionProvider* não necessitam de acesso remoto, pois todos os elementos com quem interagem estão necessariamente no mesmo nó, ou seja, na mesma máquina virtual. Logo, chamadas a métodos são suficientes para permitir a interação entre eles, além de conferir, certamente, melhor desempenho. Por este motivo eles são objetos e não agentes. O mesmo não ocorre com os demais elementos DASE, os quais foram definidos como agentes.

É importante ressaltar que a funcionalidade de um agente, implementado no JADE ou em qualquer outro sistema, não se resume a somente a sua capacidade de troca de mensagens ACL. Entretanto, tal fato foi crucial para o critério utilizado para definir os agentes durante o projeto do DASE.

Além dos argumentos discutidos nos parágrafos anteriores, o escopo de agentes e objetos também foi importante nesta questão. De forma simplificada, uma aplicação orientada a objetos comum e implementada usando linguagem Java permite que qualquer classe, ou seus métodos, tenham acesso a variáveis de escopo local (variáveis locais definidas em métodos) ou global (atributos). Além disso, todas as instâncias (objetos) de uma mesma classe em uma mesma máquina virtual Java podem compartilhar uma mesma cópia de um atributo, caso este tenha sido definido como estático.

Porém, em um sistema distribuído, como um sistema multi-agente típico implementado utilizando o JADE, a questão do escopo vai além disto, pois não há memória compartilhada. Neste caso a definição de um agente pode ser a única forma coerente de garantir que objetos de uma mesma classe ao longo de todo o sistema distribuído compartilhem informações, ou seja, tenham informações em comum, mesmo que isso seja feito através de troca de mensagens ACL.

7.2 Agentes DASEProxy

A interação entre agente cliente e agente *DASEProxy* ocorre exclusivamente através de chamadas a métodos, pois ambos estão sempre na mesma máquina virtual Java. O agente cliente obtém uma referência ao objeto do agente *DASEProxy* através do método estático *getInstance* da classe *DASEProxy*.

Esta classe funciona como um *proxy* entre o agente cliente e o *DASEManager*, permitindo que o agente cliente não precise trocar mensagens ACL para utilizar alguns serviços do sistema, como o início e a solicitação de encerramento do sistema. O agente *DASEProxy* é criado de um modo semelhante ao padrão de projeto *singleton*, porém para agentes. Isso significa que existirá no máximo uma instância da classe *DASEProxy*, ou seja, um agente *DASEProxy*, em cada nó do sistema.

7.3 Interação entre o DASEManager e outros agentes

Conforme afirmado na seção 6.1, o agente *DASEManager* tem função muito importante para o DASE, já que possui papel de coordenação. Este agente relaciona-se indiretamente com todos os agentes que compõem o DASE. Entretanto, ele relaciona-se diretamente apenas com os agentes *DASEEnvironment*, *DASEProxy* e *DASEConfigurator*, além dos agentes clientes. Sua interação com agentes clientes, agentes *DASEProxy* e *DASEConfigurator* ocorre exclusivamente por meio de troca de mensagens ACL, enquanto que sua interação com agentes *DASEEnvironment* ocorre sempre através de chamadas a métodos, pois necessariamente agentes deste tipo residem sempre no contêiner em que está o *DASEManager*.

7.3.1 Interação entre o DASEManager e agentes clientes

Durante o projeto do agente *DASEManager* foram analisadas duas formas diferentes de tal agente interagir com agentes clientes ao oferecer seus serviços. Esta interação poderia ocorrer via *proxy* ou através de troca de mensagens ACL.

As vantagens em adotar o uso de *proxy* envolveriam a definição de uma interface de programação mais simples, intuitiva e compatível com o paradigma orientado a objetos, além da possibilidade de aumentar o desempenho para comunicação no mesmo *host*, já que não haveria a necessidade de mensagens ACL, e sim simplesmente chamadas a métodos.

Entretanto, a segunda vantagem descrita no parágrafo acima é discutível, pois não é possível afirmar que o JADE não possua otimização que aproveite o fato de os dois agentes comunicantes estarem no mesmo *host* durante a troca de mensagens ACL para efetuar o envio e o recebimento das mensagens.

As vantagens em adotar mensagens ACL envolveriam um projeto mais simples e flexível. Simples, pois não seria necessário implementar o *proxy*, e flexível pois o *proxy* poderia reduzir a flexibilidade das funcionalidades do agente *DASEManager*, principalmente considerando melhorias futuras. Além disso, a

preservação da coerência com o paradigma orientado a agentes promovendo o uso de troca de mensagens ACL também é importante.

Inicialmente a segunda solução foi a escolhida durante o projeto e implementação do DASE. Entretanto em seguida a primeira opção também foi implementada. Portanto atualmente o DASE oferece praticamente todos os seus serviços aos agentes clientes de duas formas, ou via troca de mensagens ACL ou via *proxy*, realizado através do agente *DASEProxy*.

7.3.2 Interação entre o agente DASEManager e agentes DASEEnvironment

Todos os agentes *DASEEnvironment* residem sempre no mesmo contêiner em que está o agente *DASEManager*. Esta decisão de projeto facilita o controle dos ambientes de dados, já que o agente *DASEManager* pode lidar com agentes *DASEEnvironment* utilizando referências a objetos, e não por meio de troca de mensagens ACL, o que é menos eficiente.

Isto é possível porque como tais agentes estão no mesmo contêiner, estão também na mesma máquina virtual Java. Mesmo assim, o agente *DASEManager* possui os AID dos agentes *DASEEnvironment*. A decisão de manter os agentes *DASEEnvironment* sempre no contêiner em que estiver o agente *DASEManager* simplifica consideravelmente o projeto e a implementação do sistema, além de oferecer maior desempenho. No entanto isso agrava a questão de um ponto único de falha, o que faz desta decisão um importante item a ser considerado para melhorias futuras.

7.3.3 Interação entre agentes clientes e agentes DASEEnvironment

Os agentes cliente não têm contato direto com agentes *DASEEnvironment*, e se relacionam com estes agentes indiretamente através de solicitações de serviços ao *DASEManager*. Esta decisão traz as seguintes vantagens:

- Maior simplicidade durante o projeto do *DASEEnvironment*, já que ele poderá concentrar-se em prestar serviços ao *DASEManager* e a outros agentes DASE, mas não a agentes clientes.

- Segurança e controle, já que todas as solicitações dos agentes clientes aos agentes *DASEEnvironment* são interceptados e intermediados pelo agente *DASEManager*.
- Reforça a idéia de que todos agentes *DASEEnvironment* devem ficar necessariamente no container onde está o *DASEManager*, interagindo através de chamadas a métodos e não através de mensagens ACL.

7.4 Resultado de Requisição

Todo sistema gerenciador de banco de dados que possui um *driver* JDBC implementa a classe Java *java.sql.ResultSet* de modo a representar o conjunto de resultados de uma consulta SQL. No entanto, um objeto *ResultSet* possui muito mais do que simplesmente as linhas retornadas pela consulta, oferecendo funções como acesso a meta-dados sobre as colunas relacionadas e opções que permitem até mesmo a modificação dos dados.

Além disso, normalmente tais objetos são projetados visando prover o melhor desempenho possível, o que significa evitar que a totalidade de dados gerados em uma requisição trafegue pela rede de uma só vez. Isso é possível graças à manutenção de uma conexão TCP persistente e o envio de linhas do conjunto de resultados à medida que há uma solicitação, ou seja, por demanda. Isso ocorre através do método *next* de um objeto *ResultSet*.

Portanto, um objeto *ResultSet* bem projetado significa uma conexão persistente com o servidor, que mantém o conjunto de resultados gerados após a requisição. O acesso a esses dados é oferecido pelo SGBD como um *proxy*, portanto não há o tráfego de dados do servidor à aplicação cliente do conjunto total de linhas de uma única vez, o que poderia ser volumoso e prejudicial ao desempenho de qualquer sistema.

Este detalhe confere melhor desempenho aos *drivers* JDBC, no entanto traz uma desvantagem a ser considerada, impede que um objeto *ResultSet* seja serializado. Todo objeto Java que é capaz de ser armazenado e depois recuperado sem perder suas características é chamado de objeto serializável. O termo “armazenado”, ou “persistido”, significa ser mantido em um arquivo texto, um banco

de dados, ou até mesmo ser transportado pela rede, já que se considera que ao ser recebido o objeto deve ter exatamente as mesmas características que apresentava antes de ser enviado.

A maioria dos *drivers* JDBC apresenta um objeto *ResultSet* que não é serializável justamente por causa das conexões persistentes que mantém com o SGBD, e que não podem ser armazenadas em um arquivo texto, por exemplo, ou até mesmo serem transportadas de um *host* a outro pela rede. Sem dúvida esta decisão de projeto, tomada pelos desenvolvedores de *drivers* JDBC, é excelente, pois faz uso do modelo cliente-servidor de um modo racional e coerente, já que este é o contexto típico de qualquer SGBD centralizado.

O DASE também implementa a interface JDBC *ResultSet*, e é exatamente neste formato que oferece os dados aos agentes clientes. Além disso, o DASE implementa a interface JDBC *ResultSetMetaData*, permitindo ao agente cliente obter informações auxiliares sobre os dados requisitados.

Apesar do ganho em desempenho possibilitado pelo uso de *proxy* para o manuseio de objetos *ResultSet*, explicado nos parágrafos anteriores, o *ResultSet* do DASE é serializável. Isso significa que não importa qual seja o tipo de requisição de acesso a dados realizada, e nem quão volumoso são os dados a serem transportados pela rede para o atendimento dessa requisição, o DASE sempre envia todos os dados de uma única vez, ou seja, sem fazer uso de *proxy*.

Conforme já explicado anteriormente, esta decisão pode prejudicar o desempenho do sistema e sobrecarregar a rede. Ainda assim foi a adotada. Dois motivos simples justificam esta decisão:

1. Esta decisão simplifica consideravelmente o projeto e a implementação do formato das respostas de acesso a dados, pois não é necessário projetar *proxies*.
2. O uso de *proxies* contradiz o paradigma de agentes, que possui como importante característica a troca de mensagens. Na verdade esta observação está mais relacionada à especificação da FIPA sobre agentes do que à natureza original deste paradigma.

Apesar destas duas observações, o desempenho é realmente muito prejudicado sem o uso de *proxy*. Portanto essa será uma das alterações de maior prioridade dentre os itens da lista de melhorias futuras do DASE.

8 ASPECTOS FUNCIONAIS

Este capítulo apresenta detalhes sobre como cada recurso do DASE é utilizado do ponto de vista tanto do desenvolvedor quanto dos agentes clientes projetados pelo desenvolvedor. Além disso, alguns aspectos sobre o funcionamento interno do DASE também são revistos ou explicados de forma contextualizada neste capítulo.

Apesar de a arquitetura do DASE ser composta por diversos tipos de agentes, um agente cliente pode relacionar-se apenas com três tipos de agentes DASE: *DASEProxy*, *DASEManager*, e *DASEServiceLocator*. Tais agentes são o suficiente para que operações como início do sistema, encerramento do sistema, criação de ambientes de dados e requisição de acesso a dados sejam oferecidas aos agentes clientes.

Entretanto, somente o agente *DASEServiceLocator* publica seu serviço no DF, pois a interação entre agentes clientes e o agente *DASEProxy* ocorre através de chamadas a métodos, e uma vez que o agente cliente possui uma referência ao agente *DASEProxy* ele pode utilizar o mesmo para obter o AID do agente *DASEManager*.

8.1 Início e Encerramento do Sistema

O início do sistema, a partir de uma solicitação de um agente cliente, ocorre por meio da interação entre os agentes DASE *DASEProxy* e *DASEManager*. No entanto somente o agente *DASEProxy* interage com o agente cliente durante esta operação. Detalhes sobre o funcionamento do agente *DASEProxy* foram apresentados na seção 7.2.

O processo de início do sistema acontece de forma transparente para o agente cliente, pois na realidade o que o agente cliente faz é solicitar uma instância do agente *DASEProxy*. Durante a obtenção desta instância o agente *DASEProxy*, que poderá ser criado neste momento ou já existir, conforme seção 7.2, verificará se o sistema já foi iniciado. Caso o sistema ainda não tenha sido iniciado, o agente

DASEProxy providenciará a criação do agente *DASEManager*, que iniciará o sistema.

Assim, independentemente de o sistema já ter sido iniciado ou não, o agente cliente sempre se preocupará apenas em obter uma instância do agente *DASEProxy*. O trecho de código presente na figura 22 exemplifica esta situação[10]:

```
22 public class StartSystem extends Agent {
23
24     private static final long serialVersionUID = -4777953418993432605L;
25
26     protected void setup() {
27         try {
28             DASEProxy dp = DASEProxyImpl.getProxy(getContainerController());
29             AID dm = dp.getDASEManagerAID();
30             System.out.println(getName() + " has gotten the DASEManager AID: " + dm.getName());
31         } catch (InitializationException e) {
32             e.printStackTrace();
33         } catch (IllegalArgumentException e) {
34             e.printStackTrace();
35         }
36         doDelete();
37     }
38
39 }
40
```

Figura 22 - Início transparente do sistema.

No código da figura 22 a classe *StartSystem* é um agente, definido pelo usuário, pois estende a classe *Agent* do JADE. Este agente obtém uma instância do agente *DASEProxy* de seu nó através de apenas um método, o método estático *getProxy* da classe *DASEProxy* (linha 28). Durante a chamada deste método duas operações distintas podem ocorrer: a criação do agente *DASEProxy* e o início do sistema, o que na prática significa a criação do agente *DASEManager*. Caso o agente *DASEProxy* do nó em que está o agente cliente já exista, a instância do *DASEProxy* simplesmente será retornada ao agente cliente. Caso contrário ela será criada e retornada ao agente cliente.

Logo após a criação do agente *DASEProxy*, este agente verifica se o sistema já foi iniciado. Caso verifique que isso ainda não aconteceu, então o agente *DASEProxy* providencia a criação do agente *DASEManager*, e consequentemente o sistema é iniciado.

O código da figura 22 ainda mostra como um agente cliente é capaz de obter o AID do agente *DASEManager* (linha 29). De posse deste AID o agente cliente poderá comunicar-se com o agente *DASEManager* para, por exemplo, requisitar a criação de um novo ambiente de dados.

8.2 Criação e Configuração de um Ambiente de Dados

Um ambiente de dados é representado e controlado por um agente *DASEEnvironment*. No entanto, para criar um novo ambiente de dados um agente cliente não tem qualquer contato com agentes *DASEEnvironment*. Esta tarefa fica a cargo do agente *DASEManager*, que receberá solicitações de criação de ambiente de dados dos agentes clientes e os criará.

Para que possa solicitar a criação de um ambiente de dados, o agente cliente deve enviar ao *DASEManager* um descritor de ambiente de dados, representado pelo objeto *DASEEnvironmentDescriptor*. Entretanto o agente cliente pode ainda solicitar que o agente *DASEProxy* realize esta tarefa por ele.

O objeto *DASEEnvironmentDescriptor* contém todos os detalhes sobre o ambiente de dados a ser criado. Há duas formas de o agente cliente definir um descritor de ambiente de dados DASE:

1. Instanciação e configuração de um novo objeto *DASEEnvironmentDescriptor*.
2. Criação de um objeto *DASEEnvironmentDescriptor* a partir de um arquivo XML de configuração que contenha a informação a ser mantida no descritor.

A forma mais indicada para a criação de um descritor de ambiente de dados é a segunda, pois permite maior flexibilidade e manutenibilidade do sistema por parte da aplicação cliente. O trecho de código da figura 23 mostra como um agente cliente solicita a criação de um ambiente de dados utilizando um arquivo XML de configuração e os serviços do *DASEProxy*.

---

[10] Para facilitar a explicação, todas as figuras neste capítulo que representam código de programa mostrarão somente alguns trechos do código completo.

```
8  public class DASETestConfigXML extends Agent {
9
10     private static final long serialVersionUID = -6823015599319634863L;
11
12     protected void setup() {
13         try {
14             DASEProxy dp = DASEProxyImpl.getProxy(getContainerController());
15             dp.configureEnvironment("env_descriptor", true);
16         } catch (Exception e) {
17             e.printStackTrace();
18         }
19         doDelete();
20     }
21 }
22
```

Figura 23 - Criação de ambiente de dados a partir de arquivo XML.

Na linha 14 o agente cliente obtém uma referência a um agente *DASEProxy*. Em seguida, o agente cliente solicita ao agente *DASEProxy* que crie e inicie um ambiente de dados cujo descritor é representado por um arquivo XML de nome *env_descriptor.ded.xml*. Isto ocorre na linha 15, sendo, no entanto, especificado *env_descriptor*. Isso ocorre porque a extensão padrão de arquivos XML para descritores de ambientes de dados para o DASE é *.ded.xml*, por isso o DASE assume isso como padrão, não sendo portanto necessário informá-la.

Neste caso o agente *DASEProxy*, de posse do AID do agente *DASEManager*, cria um novo objeto descritor de ambiente de dados a partir do arquivo XML. Em seguida, ele verifica se o descritor possui informações consistentes. Isso evita que futuramente uma requisição de criação de ambiente de dados seja enviada ao *DASEManager* contendo um descritor inválido, embora o *DASEManager* sempre verifique a consistência do descritor antes de processá-lo.

Caso o descritor esteja consistente, uma mensagem ACL é criada e configurada de modo a representar a requisição de criação de um novo ambiente de dados. Esta mensagem é enviada pelo agente *DASEProxy* ao agente *DASEManager*.

8.2.1 XML utilizado em um descritor de ambiente de dados

A figura 24 apresenta um exemplo de arquivo XML que representa um descritor de ambiente de dados. Neste exemplo a maioria das propriedades possíveis de um descritor de ambiente de dados é utilizada, no entanto nem todas são obrigatórias.

```
 1<dase-ed label="dase-test">
 2
 3    <driver>org.hsqldb.jdbcDriver</driver>
 4    <cctype>FORWARD</cctype>
 5    <rctype>FORWARD</rctype>
 6    <lbtype>ROUND_ROBIN</lbtype>
 7    <similarity>true</similarity>
 8
 9    <node-list>
10        <node container="node1" url="jdbc:hsqldb:hsql://localhost/db"
11            user="sa" password=""/>
12        <node container="node2" url="jdbc:hsqldb:hsql://localhost/db"
13            user="sa" password=""/>
14        <node container="node3" url="jdbc:hsqldb:hsql://localhost/db"
15            user="sa" password=""/>
16        <node container="node4" url="jdbc:hsqldb:hsql://localhost/db"
17            user="sa" password=""/>
18    </node-list>
19
20    <sentence-list>
21        <sentence name="update"
22            sql="UPDATE contas SET saldo=saldo+120 WHERE id=0;"/>
23
24        <sentence name="query_id"
25            sql="SELECT * FROM contas WHERE id=:id;"/>
26    </sentence-list>
27
28</dase-ed>
```

Figura 24 - Exemplo de um arquivo XML que representa um descritor de ambiente de dados.

Para que um arquivo XML como este possa permitir a criação de um descritor válido, a propriedade *label*, e a *tag driver* e ao menos um nó (*node*) devem ser definidos. As demais *tags* e propriedades são opcionais ou assumiriam valores padrão. A propriedade *label* mantém o rótulo do ambiente de dados, conforme explicado na subseção 6.1.5. A *tag driver* indica qual é o *driver* JDBC utilizado pelo

ambiente de dados. Consequentemente é possível identificar qual é o SGBD adotado através de seu *driver*.

A *tag node-list* representa uma lista de *node*. Um *node*, ou nó, deve conter necessariamente as propriedades *container* e *url*. As propriedades *user* e *password* de um node são obrigatórias somente se as *tags duser* e *dpassword* não tiverem sido definidas, já que são opcionais. Este é o caso do exemplo da figura 24. Todas as demais propriedades assumem valores padrão caso não sejam definidas.

A *tag sentence-list* representa uma lista de *sentence*. Uma *sentence* é uma sentença SQL pré-definida que poderá ser utilizada a qualquer momento em qualquer requisição de acesso a dados destinada ao ambiente de dados em questão. Tal *tag* deve conter necessariamente as propriedades *name*, que é o identificador da sentença, e *sql*, que guarda a sentença SQL a ser executada.

Tais sentenças podem ainda conter parâmetros, tornando-as flexíveis e mais úteis. Todo parâmetro é reconhecido através de um identificador precedido de um caractere de dois pontos. Por exemplo, o descritor da figura 24 possui duas sentenças. A sentença chamada *update* não possui nenhum parâmetro, enquanto que a sentença chamada *query_id* possui um parâmetro chamado *id*. O valor deste parâmetro deve ser definido em tempo de execução permitindo que o agente cliente modifique dinamicamente a sentença pré-definida.

A tabela 7 descreve a função e o valor padrão de todas as *tags,* e suas propriedades, possíveis em um arquivo XML descritor de ambiente de dados DASE. Nesta tabela somente as propriedades opcionais possuem valor padrão. A propriedade *path* é utilizada para que o próprio sistema gere um arquivo de configuração a partir de um objeto *DASEEnvironmentDescriptor*. Conforme dito anteriormente, a extensão padrão, embora não obrigatória, de todo arquivo XML que representa um descritor de ambiente de dados é *.ded.xml*.

Tabela 7 - *Tags* de um arquivo XML descritor de ambiente de dados.

***Tag* ou Propriedade**	**Descrição**	**Valor padrão**
label	Rótulo do ambiente de dados.	
driver	*Driver* JDBC do SGBD adotado para o ambiente de dados.	
cctype	Tipo de controle de concorrência adotado.	FORWARD
lbtype	Tipo de balanceamento de carga adotado.	ROUND_ROBIN
rctype	Tipo de controle de recuperação adotado.	FORWARD
path	Caminho do arquivo de configuração do descritor de ambiente de dados.	*label*.ded.xml
similarity	Indica se o sistema permitirá que mais de um nó possua a mesma URL.	*False*
duser	Usuário padrão a ser usado em nó, caso o mesmo seja definido sem a especificação do usuário.	*Null*
dpassword	Senha padrão a ser usada em nó, caso o mesmo seja definido sem a especificação da senha.	*Null*
node-list	Lista que contém a definição dos nós (*node*) do ambiente de dados.	-
node	Representa um nó do ambiente de dados. Esta *tag* possui as propriedades *container*, *url*, *user* e *password*. Dessas, somente *container* e *url* são obrigatórias.	-
container	Nome do contêiner JADE que será utilizado como um dos nós do ambiente de dados.	
url	URL que permite a conexão entre o nó do ambiente de dados e o nó do SBD.	
user	Utilizada para autenticação no nó do SBD. Esta propriedade é obrigatória somente se *duser* e *dpassword* não estiverem sido definidas.	
password	Utilizada para autenticação no nó do SBD. Esta propriedade é obrigatória somente se *duser* e *dpassword* não estiverem sido definidas.	
sentence-list	Lista que contém a definição das sentenças SQL pré-definidas do ambiente de dados.	-
sentence	Sentença SQL pré-definida.	-
name	Nome da sentença SQL pré-definida.	
sql	Código SQL da sentença SQL pré-definida.	

8.3 Requisição de Dados

Para que uma requisição de acesso a dados a partir do DASE seja realizada quatro etapas são necessárias:

1. Obtenção do AID do agente *DASEServiceLocator* que intermediará a requisição.
2. Envio da requisição de acesso a dados.
3. Obtenção do objeto *MessageTemplate*, utilizado para identificar a resposta da requisição.
4. Recebimento dos dados requisitados.

Essas quatro etapas podem ser realizadas diretamente pelo próprio agente cliente, ou então este pode solicitar a um agente *DASEProxy* que as execute, recebendo ao final o resultado da requisição. As subseções a seguir descrevem os quatros passos com detalhes para o caso de o próprio agente cliente realizá-las.

8.3.1 Obtenção do AID do agente DASEServiceLocator

O agente *DASEServiceLocator* é quem receberá a requisição de acesso a dados e a repassará ao agente DASE responsável pelo processamento da requisição. O agente *DASEServiceLocator* atua portanto como um intermediário "facilitador" do serviço, pois impede que o agente cliente seja obrigado a saber exatamente quais agentes deverão mobilizar-se para que sua requisição seja atendida. Portanto, graças ao agente *DASEServiceLocator* este serviço é transparente ao agente cliente.

Existe exatamente um agente *DASEServiceLocator* para cada ambiente de dados. O agente cliente deve, portanto, saber qual é o AID deste agente para que possa iniciar o processo de requisição de dados. O agente cliente obtém o AID do agente *DASEServiceLocator* da mesma forma que faria para obter o AID de qualquer agente existente em uma plataforma em conformidade com o padrão FIPA, ou seja, utiliza o DF.

A única diferença neste caso é que o DASE oferece de forma pronta ao agente cliente uma instância da classe JADE *DFAgentDescription*, necessária para a

localização de qualquer agente junto ao DF. Este objeto deve ser obtido através de um objeto DASE *ServiceDescriptor*, descrito com mais detalhes na próxima subseção. A figura 25 traz um trecho de código em que o AID do agente *DASEServiceLocator* é obtido junto ao DF.

```
 95        DFAgentDescription adr;
 96        DFAgentDescription[] ada = null;
 97
 98        adr = serviceDescriptor.createSLDescription();
 99        AID serviceLocatorAID = null;
100
101        try {
102            SearchConstraints constraints = new SearchConstraints();
103            constraints.setMaxDepth((long) 1);
104            constraints.setMaxResults((long) 1);
105            ada = DFService.searchUntilFound(this, getDefaultDF(), adr,
106                    constraints, requestTimeExpiration);
107
108            serviceLocatorAID = ada[0].getName();
109        } catch (FIPAException e) {
110            doDelete();
111        }
```

Figura 25 - Exemplo de agente cliente obtendo o AID do agente *DASEServiceLocator*.

8.3.2 Envio de requisição de acesso a dados

De posse do AID do agente *DASEServiceLocator* já é possível enviar-lhe a requisição de acesso a dados. Esta requisição é representada por um descritor de serviço de requisição de dados, cuja implementação refere-se ao objeto *ServiceDescriptor*.

Tal descritor de serviço é um objeto simples, sendo formado basicamente por três elementos:

1. O contêiner do agente cliente: container JADE onde está o agente cliente.
2. Sentença SQL: a sentença SQL utilizada para o acesso aos dados, ou então o nome e os parâmetros de uma sentença SQL pré-definida.
3. Um contêiner sugerido pelo agente cliente: container sugerido pelo agente cliente para que o DASE o considere durante o balanceamento de carga de

requisições de acesso a dados, conforme explicado na seção 6.3. Dos três elementos do descritor de serviço este é o único opcional.

Considerando que o descritor do serviço já está pronto, a requisição de acesso a dados resume-se a uma solicitação de serviço comum entre agentes. No entanto o objeto *ServiceDescriptor* possui um método que permite que o agente cliente obtenha o objeto da mensagem ACL, facilitando o trabalho do agente cliente, que não precisa instanciar nem configurar tal objeto. O trecho de código da figura 26 ilustra a instanciação de um objeto *ServiceDescriptor*. A figura 27 ilustrará o envio da mensagem ACL.

```
88    ServiceDescriptor serviceDescriptor = new ServiceDescriptor(environmentLabel);
89    serviceDescriptor.useStoredSentence(sentenceName);
90    if (parameters != null) {
91        serviceDescriptor.addStoredSentenceParameters(parameters, values);
92    }
```

Figura 26 - Exemplo de objeto *ServiceDescriptor*.

8.3.3 Como identificar a resposta da requisição

Após enviar a requisição ao agente *DASEServiceLocator*, o agente cliente deverá aguardar que o mesmo responda. A resposta do agente *DASEServiceLocator* não será o resultado da requisição de dados, mas sim uma mensagem que contém um objeto *MessageTemplate*. Este objeto, um recurso muito útil oferecido pelo JADE, permite que agentes consigam filtrar e identificar uma determinada a mensagem dentre diversas recebidas.

Assim, o agente *DASEServiceLocator* envia este *MessageTemplate* ao agente cliente para que este possa identificar a mensagem contendo a resposta da requisição de acesso a dados quando a receber. Nesta etapa não há nenhum método ou objeto específico do DASE que determine o modo como o agente cliente receberá o objeto *MessageTemplate* do agente *DASEServiceLocator*. Trata-se simplesmente do envio de uma mensagem ACL de um agente a outro, embora este procedimento seja uma importante etapa do protocolo de interação entre agentes *FIPA Recruiting*, explicado na subseção 2.5.5.

A figura 27 ilustra o envio de uma mensagem ACL contendo um objeto *ServiceDescriptor* ao agente *DASEServiceLocator*, e também o recebimento por parte do agente cliente do objeto *MessageTemplate* referente à requisição realizada.

```
206     private void sendRequestMessage() {
207         ACLMessage request = null;
208         try {
209             request = serviceDescriptor.createMessage(slAID);
210         } catch (IOException e) {
211             setFailureInformer(e);
212             myAgent.doDelete();
213         }
214         myAgent.send(request);
215         mtResult = MessageTemplate.and(MessageTemplate.MatchSender(slAID),
216                 MessageTemplate.MatchPerformative(ACLMessage.INFORM));
217         mtResultFailure = MessageTemplate.and(MessageTemplate.MatchSender(slAID),
218                 MessageTemplate.MatchPerformative(ACLMessage.FAILURE));
219         step = 1;
220     }
221 
222     private void receiveMessageTemplate() {
223         result = myAgent.receive(mtResult);
224 
225         if (result == null) {
226             result = myAgent.receive(mtResultFailure);
227             if (result == null)
228                 block();
229             else {
230                 LogGenerator.agentService(myAgent.getName(),
231                     "It has not gotten success in its data request",
232                     Services.DataProvider.DATA_REQUEST.name());
233                 step = -1;
234             }
235         } else {
236             try {
237                 MessageTemplate mt = (MessageTemplate) result.getContentObject();
238                 mtInform = MessageTemplate.and(mt, MessageTemplate
239                         .MatchPerformative(ACLMessage.INFORM));
240                 mtFailure = MessageTemplate.and(mt, MessageTemplate
241                         .MatchPerformative(ACLMessage.FAILURE));
242                 step = 2;
243             } catch (UnreadableException e) {
244                 setFailureInformer(e);
245                 myAgent.doDelete();
246             }
247         }
248     }
```

Figura 27 - Envio de mensagem ACL com *ServiceDescriptor* ao agente *DASEServiceLocator* e recebimento de objetos *MessageTemplate*.

8.3.4 Recebimento dos dados requisitados

O recebimento dos dados requisitados é a última etapa durante a requisição de dados. Conforme explicado na subseção anterior, o agente cliente só conseguirá identificar a mensagem que contém os dados que solicitou caso utilize o *MessageTemplate* enviado pelo agente *DASEServiceLocator*.

Os resultados das requisições de acesso a dados utilizando o DASE são representados sempre por meio de objetos do tipo *ResultSet*, de acordo com o padrão JDBC. Isto significa que um agente cliente pode tratar dados obtidos a partir do DASE exatamente do mesmo modo que faria caso os obtivesse de um driver JDBC de qualquer SGBD. A seção 7.4 descreve alguns detalhes sobre a implementação de *ResultSet* realizada pelo DASE. A figura 28 ilustra o recebimento do resultado de uma requisição de acesso a dados.

```
250    private void receiveRequestedData() {
251        inform = myAgent.receive(mtInform);
252        failure = myAgent.receive(mtFailure);
253
254        if (inform != null) {
255            LogGenerator.agentService(myAgent.getName(), "Has gotten the data for "
256                    + serviceDescriptor.toString(), Services.DataProvider.DATA_REQUEST.name());
257            try {
258                DASERequestAnswer answer = (DASERequestAnswer) inform.getContentObject();
259                if (answer == null) {
260                    LogGenerator.agentService(myAgent.getName(),
261                            "It has received a null message as answer for "
262                                    + serviceDescriptor.toString(),
263                            Services.DataProvider.DATA_REQUEST.name());
264                } else if (informer != null) {
265                    informer.setContent(answer);
266                    informer.setTRUE();
267                } else {
268                    PrintRequestAnswer pra = new PrintRequestAnswer();
269                    pra.print(answer);
270                }
271            } catch (UnreadableException e) {
272                setFailureInformer(e);
273            } catch (SQLException e) {
274                setFailureInformer(e);
275            }
276            step = -1;
277        } else if (failure != null) {
278            LogGenerator.agentService(myAgent.getName(),
279                    "It has not gotten success in its data request for "
280                            + serviceDescriptor.toString() + ". Reason: " + failure.getContent(),
281                    Services.DataProvider.DATA_REQUEST.name());
282            step = -1;
283        } else {
284            block();
285        }
286    }
```

Figura 28 - Recebimento do resultado de uma requisição de acesso a dados.

9 RESULTADOS

Para avaliar a funcionalidade e o desempenho do DASE, três testes foram realizados. As aplicações utilizadas foram baseadas em agentes responsáveis por gerar solicitações de acesso aos dados.

O SisBDPar e o PostgreSQL, um SGBD centralizado, foram os SGBD utilizados. Os SGBDs foram executados utilizando computadores com dois processadores *Athlon* MP 2400+, 1GB de memória e com acesso a discos locais. Apenas um processador de cada nó foi utilizado. Para o caso do SisBDPar um computador adicional foi utilizado para executar o processo *master* do NPFS, tal computador possui dois processadores *dual Xeon* de 3.20GHz , 4GB de memória e disco local. Os testes são descritos nas próximas seções, no entanto a tabela 8 apresenta algumas de suas características.

Tabela 8 - Testes elaborados para o DASE.

Teste	Objetivo	Quantidade de nós	SGBDs	Tipo de SQL
1	Desempenho e funcionalidade	1	PostgreSQL e SisBDPar	Consulta
2	Sobrecarga e funcionalidade	1	SisBDPar	Consulta
3	Acesso concorrente	4[11]	SisBDPar	Transações com instruções de leitura e escrita

Os testes 1 e 2 foram realizados sobre uma base de dados com apenas uma tabela, formada por quatro colunas e que continha 90.000 linhas. A sentença SQL relacionada à consulta em questão continha uma cláusula *WHERE* fazendo-a retornar sempre apenas uma pequena parte dos dados, o que correspondia a 84

[11] Exclusivamente para este caso cada nó representa uma máquina virtual.

linhas. Já o teste 3 envolveu o uso de transações que continham operações de leitura e escrita sobre os mesmos dados.

É importante destacar que embora o impacto do DASE sobre o desempenho do sistema que o utiliza seja fator importante e tenha sido considerado durante o seu projeto e implementação, o objetivo real do DASE está fortemente relacionado a aspectos funcionais e de controle de consistência dos dados, e não à melhora de desempenho de um sistema distribuído.

Portanto não faz parte da proposta do DASE atuar em um sistema de modo a permitir aumento de desempenho durante o uso dos recursos compartilhados e distribuídos, embora a sobrecarga que o próprio DASE gera também seja um fator importante a ser levado em consideração.

Assim, os testes descritos a seguir em nenhum momento estão relacionados à idéia de que o DASE deveria atuar significativamente de modo positivo sobre o desempenho do sistema. O objetivo principal é evidenciar sua funcionalidade.

9.1 Resultado do teste 1

Este teste compreende a comparação entre o tempo de acesso de uma consulta a partir do DASE ao SisBDPar, e uma consulta a partir do DASE ao PostgreSQL.

O primeiro item a ser obtido como resultado deste teste é funcional e está relacionado à capacidade do DASE de ser compatível com diferentes tipos de SGBDs. Este teste mostra que o DASE realmente é capaz de operar sobre diferentes SGBD, além de oferecer seus serviços aos agentes clientes sempre da mesma forma.

A tabela 9 contém os tempos em milissegundos referentes a este teste e mostra que o tempo de acesso aos dados para o SisBDPar foi consideravelmente menor do que o tempo de acesso aos dados para o PostgreSQL, ambos sob a intermediação do DASE.

Tabela 9 - Tempos referentes ao teste 1.

Quantidade de nós	SGBD	Tempo (ms)
1	SisBDPar	380
1	PostgreSQL	2151

9.2 Resultado do teste 2

Este teste compreende a comparação entre o tempo de acesso de uma consulta a partir do DASE ao SisBDPar utilizando 1 nó, contra o tempo de acesso da mesma consulta ao SisBDPar diretamente, sem utilizar o DASE, utilizando também 1 nó.

Com este teste é possível verificar qual é a sobrecarga que o DASE traz às consultas. O tempo para consulta com somente o SisBDPar foi de 13 milissegundos, enquanto que o tempo para consulta com o SisBDPar e com o DASE foi de 380 milissegundos. Através deste resultado nota-se que a sobrecarga inerente ao DASE foi de 367 milissegundos. A tabela 10 traz os tempos referentes ao teste 2.

Tabela 10 - Tempos referentes ao teste 2.

Tempo SisBDPar (ms)	Tempo DASE e SisBDPar (ms)	Sobrecarga (ms)
13	380	367

9.3 Resultado do teste 3

O objetivo deste teste é verificar a garantia da consistência dos dados mesmo após diversos acessos concorrentes e conflitantes a partir do DASE. Para a realização deste teste o DASE foi instalado em quatro máquinas virtuais, juntamente com o SisBDPar, com o controle de concorrência ativado em modo *TIMESTAMPS* e com o controle de recuperação ativado em modo *UNDO/NO REDO*.

O banco de dados utilizado para o teste continha apenas uma tabela, chamada "contas", a qual possuía somente duas colunas do tipo inteiro, chamadas

"id" e "saldo". A idéia, portanto, era simular contas corrente sobre as quais transações concorrentes executariam operações conflitantes. Ao final do teste as contas deveriam possuir saldos consistentes, ou seja, com o mesmo valor resultante caso todas as transações fossem executadas de modo seqüencial.

Um tipo de agente, chamado *RequestCreator*, foi projetado e implementado especificamente para esse teste com a função de gerar diversas requisições de acesso a dados seguidamente, que são encaminhadas ao agente utilitário do DASE chamado *Request*, descrito na subseção 6.1.4.

Um nó adicional aos quatro já existentes foi adicionado à plataforma, formando um total de cinco nós, para que fosse utilizado exclusivamente para a atuação do agente *RequestCreator*. A idéia foi permitir que o balanceamento de carga do DASE fosse circular, já que as requisições partiriam sempre necessariamente de um nó que não fazia parte do ambiente de dados.

Assim, após ser criado o agente *RequestCreator* disparava a criação de agentes DASE *Request*, que também operavam no nó adicional, mas ao solicitarem a requisição de acesso a dados obrigavam que o agente *DASEServiceLocator* balanceasse a carga de requisições de forma circular, o que fazia com que agentes *DASEDataProvider* fossem criados circularmente em cada um dos quatro nós do ambiente de dados.

Dessa forma as requisições solicitadas inicialmente pelo agente *RequestCreator* eram necessariamente executadas em diferentes nós de forma circularmente.

O agente *RequestCreator* é configurável, podendo operar de modo diferente dependendo do detalhamento do teste a ser executado. Ele recebe como parâmetros de configuração os seguintes argumentos:

1. Número de agentes *Request* a serem criados. Determina a quantidade de agentes *Request* criada pelo agente *RequestCreator*.
2. Rótulo do ambiente de dados. Rótulo do ambiente de dados sobre o qual todas as requisições de acesso a dados são executadas.

3. Expressão de requisição para cada agente *Request*. Tal expressão é formada pelo nome de uma sentença SQL pré-definida seguida de pares de parâmetros e seus valores, caso estes existam.

Durante este teste foram utilizados os seguintes parâmetros para o agente *RequestCreator*.

1. Número de agentes *Request* a serem criados: 600.
2. Rótulo do ambiente de dados: *env_test*.
3. Expressão de requisição para cada agente *Request: tx1*.

A sentença pré-definida *tx1* corresponde à sentença representada pela figura 29.

```
67        <sentence name="tx1"
68            sql="
69                DELETE FROM contas;
70                INSERT INTO contas VALUES(0, 0);
71                UPDATE contas SET saldo=saldo+10 WHERE id=0;
72            "/>
```

Figura 29 – Sentença pré-definida *tx1*.

Como o modo de controle de concorrência definido foi o *TIMESTAMPS*, então necessariamente as três instruções SQL da sentença pré-definida *tx1* representavam o conteúdo de uma transação. Dessa forma, as três operações deveriam ser executadas de forma atômica, ou seja, ou todas as operações eram realizadas, ou todas as operações já realizadas eram desfeitas, caso houvesse um aborto da transação.

Essa transação realiza três operações:

1. Exclui todas as linhas da tabela “contas”.
2. Inclui uma nova linha cujo “saldo” é zero e o “id” é zero também.
3. Adiciona o valor 10 ao “saldo” da conta cujo “id” é zero, ou seja, a conta recém-criada.

Nota-se que tal transação possui operações de escrita (as três) e de leitura também (somente a terceira). No entanto todas elas ocorrem sobre os mesmos dados. Portanto, caso tal transação fosse executada mais de uma vez concorrentemente, e caso o controle de concorrência não atuasse corretamente, surgiria o risco de inconsistência de dados.

Caso não houvesse controle de concorrência, adotando como exemplo duas transações, TA e TB, e como exemplo de escalonamento a seqüência de operações da figura 30, nota-se que a quarta operação não produziria o efeito desejado, pois não haveria nenhuma linha com "id" 0 a ser modificada, já que a transação B excluiu todas as linhas antes que a transação A pudesse concluir suas operações.

1. **TA executa operação 1.**
2. **TA executa operação 2.**
3. **TB executa operação 1.**
4. **TA executa operação 3.**
5. **(...)**

Figura 30 – Escalonamento das transações A e B.

Assim a transação A seria concluída produzindo um resultado diferente do resultado que teria produzido caso as duas transações tivessem sido executadas de forma seqüencial.

Caso as transações A e B, e qualquer quantidade de outras transações idênticas, fossem executadas de forma seqüencial, o resultado final correto deveria ser sempre apenas uma linha na tabela contas, cujo "id" seria igual a 0 e "saldo" igual a 10.

Assim, para este teste todas as requisições de acesso a dados definidas continham transações com operações de leitura e escrita sobre os mesmos dados. Tais requisições foram executadas ao mesmo tempo, criando uma possibilidade de inconsistência de dados, já que as transações envolvidas possuíam operações conflitantes entre si.

Ao final de todos os acessos, que ocorrem a partir de todos os nós “virtuais” e que foram disparados ao mesmo tempo, o *log* gerado pelo DASE foi verificado e notou-se que diversas transações foram abortadas, e iniciadas novamente em seguida, por infligirem as regras do *TIMESTAMP*, justamente por atuarem em dados conflitantes.

Outras transações também foram abortadas devido a *livelocks*, entretanto em seguida puderam ser tratadas pelo mecanismo de controle de concorrência e concluídas normalmente.

Os dados também foram verificados e mostraram-se consistentes, ou seja, com os valores esperados. Tal resultado comprovou a eficácia do controle de concorrência do DASE, assim como de seu controle de recuperação. Além disso, pode-se verificar que o DASE funcionou como esperado em um ambiente de dados com mais de um nó.

9.4 Projeto GIGA-CDM

Além dos testes descritos nas seções anteriores, o DASE teve sua estabilidade, funcionalidade e desempenho validados por meio de sua adoção em um projeto financiado pela Rede Nacional de Ensino e Pesquisa e cujo título é “Prototipagem de Um Sistema de Apoio à Decisão Compartilhada para Infra-estrutura Aeronáutica sobre Bases de Dados Paralelas e Distribuídas” (Carvalho, 2006).

Neste projeto foram desenvolvidos um protótipo de um sistema de controle de tráfego aéreo e um ambiente de simulação que permitisse o teste do protótipo. Tanto o protótipo quanto o sistema de simulação foram implementados utilizando sistemas multi-agentes e banco de dados distribuído, e para integrar ambos o DASE foi utilizado. A plataforma de sistemas multi-agentes utilizada foi o JADE e o SGBD distribuído utilizado foi o SisBDPar.

10 CONCLUSÃO

Neste capítulo são apresentadas as conclusões deste trabalho e uma relação de itens importantes para o aperfeiçoamento futuro do sistema aqui proposto.

10.1 Conclusões

A necessidade dos agentes por acesso a dados distribuídos é um requisito presente em qualquer sistema multi-agente, já que tais sistemas são normalmente complexos e de natureza distribuída. Este trabalho apresentou alguns detalhes acerca desta necessidade, além de como supri-la através da proposta de um sistema chamado DASE, *Distributed data Agent Service Environment*.

O projeto da arquitetura do sistema proposto neste trabalho teve como elemento direcionador a busca de flexibilidade e qualidade. Por este motivo ela foi pensada sempre focando a simplicidade. Assim, seus principais componentes foram especificados, projetados e implementados, além de integrados com a plataforma de agentes adotada, de modo a permitir diversas melhorias e correções futuras.

O DASE é capaz de oferecer a um sistema multi-agente o acesso a dados distribuídos sem que a complexidade do mecanismo de persistência e do manuseio dos dados seja um obstáculo, conferindo transparência e padronização. O DASE também é um sistema multi-agente, portanto oferece sua funcionalidade através de agentes, implementados sobre uma plataforma chamada JADE, *Java Agent DEvelopment Framework*.

Além do paradigma de agentes, e da necessidade que um sistema multi-agente possui por acesso a dados distribuídos, um ponto bastante importante neste trabalho é o controle de concorrência, tanto que este é também um dos principais recursos oferecidos pelo DASE, assim como o controle de recuperação. O controle de concorrência do DASE possui granularidade grossa e é baseado no método de controle de concorrência *TIMESTAMPS*.

10.2 Trabalhos Futuros

As principais melhorias futuras a serem feitas no sistema envolvem o amadurecimento do sistema de controle de concorrência, melhorias no algoritmo de manutenção de agentes provedores de dados, e evolução no balanceamento de carga. Outros aperfeiçoamentos importantes aparecem abaixo:

- Remoção da restrição de apenas um agente *DASEConfigurator* em todo o sistema. Embora esta melhoria não traga modificações no desempenho do sistema, ela será importante para permitir maior usabilidade e controle ao usuário.
- Incluir o uso de ontologia específica para o DASE, que possibilite a forma correta de identificar o conteúdo de mensagens ACL típicas de serviços oferecidos por agentes DASE. É possível cria uma ontologia usando alguns recursos oferecidos pelo próprio JADE.
- Possibilidade de modificação de ambientes de dados em tempo de execução. Operações como iniciar, interromper, reiniciar, remover ou salvar seriam funções úteis ao sistema.
- Inclusão de uma camada de persistência de agentes na arquitetura (Ambler, 2005), com um possível mapeamento agente-relacional.
- Desenvolvimento de recursos relacionados a controle de acesso, segurança e tolerância a falhas. Um exemplo disso é a replicação de alguns tipos de agentes DASE, como o *DASEManager*, visando fazer com que o sistema continue funcional mesmo após o encerramento inesperado de um ou mais agentes.
- Desenvolvimento de *ResultSet* utilizando conexões persistentes, conforme discutido na seção 7.4. Isso certamente conferiria maior desempenho durante a entrega do resultado de uma requisição de acesso a dados.
- Remoção da restrição que faz com que o controle de concorrência do DASE garanta a consistência dos dados considerando somente modificações realizadas por transações em um mesmo ambiente de dados.

- o Diminuição da granularidade do controle de concorrência do DASE.
- o Implementação de técnicas adicionais e mais eficientes de balanceamento de carga.
- o Desenvolvimento de uma especificação que estenda o JDBC permitindo que haja formas de obter informações específicas sobre a localização dos dados oriundos de sistemas de bancos de dados distribuídos. Isso permitiria ao DASE utilizar técnicas de controle de concorrência distribuído, e balanceamento de carga, muito mais precisas e eficientes.

REFERÊNCIAS BIBLIOGRÁFICAS

Aldo, C. E R. Giovanni. Multi-Agent Systems And Territory: Concepts, Methods And Applications: Ersa Congress 2003.

Almeida Junior, J. R. D., L. M. Sato, *Et Al.* Modelagem De Dados Para Gerenciamento De Tráfego Aéreo. Iv Simpósio De Transporte Aéreo (Sitraer), V.1, P.139-149. 2005.

Ambler, S. W. The Design Of A Robust Persistence Layer For Relational Databases: Ambysoft Inc. 2005.

Aparicio, M., L. Chiariglione, *Et Al.* Fipa - Intelligent Agents From Theory To Practice. Geneva: Telecom 99 1999.

Bellifemine, F., G. Caire, *Et Al.* Jade - A White Paper. 3 2003.

______. Jade Programmer's Guide: 23-33 P. 2005.

Bellifemine, F., A. Poggi, *Et Al.* Jade - A Fipa2000 Compliant Agent Development Environment. Agents'01, May 28-June 1, P.216-217. 2001.

Bernstein, P. A. Middleware: A Model For Distributed System Services. Commun. Acm, V.39, P.86-98. 1996.

Bernstein, P. A. E N. Goodman. Concurrency Control In Distributed Database Systems. Computing Surveys, V.13, No. 2, June. 1981.

______. Multiversion Concurrency Control-Theory And Algorithms. Acm Trans. Database Syst., V.8, N.Acm Press, P.465-483. 1983.

Bernstein, P. A., V. Hadzilacos, *Et Al.* Concurrency Control And Recovery In Database Systems: Addison-Wesley Publishing Company. 1987 (Computer Science)

Bratman, M. Intention, Plans, And Practical Reason. Cambridge, Ma (Usa): Harvard University Press. 1987

Busetta, P., R. Rönquist, *Et Al.* Jack Intelligent Agents - Components For Intelligent Agents In Java. Agentlink News. Melbourne, Australia.: Agent Oriented Software Pty. Ltd.: 2-5 P. 1999.

Carvalho, F. S. S., L. M.; Oliveira, Ítalo Romani De; Cugnasca, P. S.; Mayeda, L. Air Traffic Service Management Based On Collaborative Agents And Passenger Delay Criteria. Journal Of The Brazilian Air Transportation Research Society, V.1. 2006.

Conte, R., N. Gilbert, *Et Al.* Mas And Social Simulation: A Suitable Commitment. Proceedings Of The First International Workshop On Multi-Agent Systems And Agent-Based Simulation: Springer-Verlag, 1998. 1-9 P.

Corba™/Iiop™ Specification. Needham, Ma: Object Management Group. 2006 2002.

Downing, T. Java Rmi: Remote Method Invocation. Foster City, Ca: Idg Books Worldwide. 1998

Ellis, J., L. Ho, *Et Al.* Jdbc™ 3.0 Specification. October. 2001.

Endrei, M., J. Ang, *Et Al.* Patterns: Service-Oriented Architecture And Web Services: Ibm Redbooks. 2004. 39 P.

Ferreira, J. E. E M. Finger. Controle De Concorrência E Distribuição De Dados: A Teoria Clássica, Suas Limitações E Extensões Modernas. 2000.

Finin, T. E R. Fritzson. Kqml - A Language And Protocol For Knowledge And Information Exchange. Proceedings Of The 13th Intl. Distributed Artificial, 1994. P.

Fipa Acl Message Structure Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Agent Management Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2004.

Fipa Brokering Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Contract Net Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Propose Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Recruiting Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Request Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Fipa Subscribe Interaction Protocol Specification. Geneva, Switzerland: Foundation For Intelligent Physical Agents 2002.

Gnu Lesser General Public License. Free Software Foundation, Inc. 2006 1999.

Gordon, R. Essential Jni: Java Native Interface. Upper Saddle River, Nj: Prentice-Hall, Inc. 1998

Guardia, H. C. Considerações Sobre As Estratégias De Um Sistema De Arquivos Paralelos Integrado Ao Processamento Distribuído. (Doutorado). Escola Politécnica, Universidade De São Paulo, São Paulo, 1999.

Gulutzan, P. E T. Pelzer. Sql-99 Complete, Really: R&D Books. 1999

Hiebeler, D. The Swarm Simulation System And Individual-Based Modeling. Proceedings Of Decision Support 2001: Advanced Technology For Natural Resource Management, September 12. 1994.

Kinny, D. E M. George. Modelling And Design Of Multi-Agent Systems. Proceedings Of The Third International Workshop On Agent Theories, Architectures, And Languages. Australia: Springer In The Lecture Notes In Articial Intelligence. November, 1996. P.

Lubacheski, F. A. G. Uma Infra-Estrutura Para Banco De Dados Baseada Em Arquivos Paralelos E Distribuídos. (Mestrado). Escola Politécnica, Universidade De São Paulo, São Paulo, 2005.

Moreno, A., Aïdavalls, *Et Al.* Management Of Hospital Teams For Organ Transplants Using Multi-Agent Systems. Proceedings Of The 8th Conference On Ai In Medicine In Europe: Artificial Intelligence Medicine: Springer-Verlag, 2001. 374-383 P.

North, M. J., N. T. Collier*, Et Al.* Experiences Creating Three Implementations Of The Repast Agent Modeling Toolkit. Acm Transactions On Modeling And Computer Simulation, V.16, N.1, January, P.1-25. 2006.

Odell, J., H. V. D. Parunak*, Et Al.* Representing Agent Interaction Protocols In Uml. Agent-Oriented Software Engineering. Berlin: Springer-Verlag, 2001. 121–140 P.

Oliveira, E., K. Fischer*, Et Al.* Multi-Agent Systems: Which Research For Which Applications: Robotics And Autonomous Systems 1999.

Papadimitriou, C. H. E P. C. Kanellakis. On Concurrency Control By Multiple Versions. Acm Trans. Database Syst., V.9, N.Acm Press, P.89-99. 1984.

Rising, L. Patterns: A Way To Reuse Expertise. Communications Magazine. 37 1999.

Sonnessa, M. Jas: Java Agent-Based Simulation Library - An Open Framework For Algorithm-Intensive Simulations.: Department Of Computer Science, University Of Torino. 2004.

Sycara, K. P. Multiagent Systems. Ai Magazine. 10**:** 79-93 P. 1998.

Tanenbaum, A. S. Sistemas Operacionais Modernos. São Paulo: Prentice Hall. 2003

Vinoski, S. Corba: Integrating Diverse Applications Within Distributed Heterogeneous Environments. Ieee Communications Magazine. 35 No. 2 1997.

Weiss, G. Multiagent Systems, A Modern Approach To Distributed Artificial Intelligence. Cambridge, Massachusetts: The Mit Press. 2000

APÊNDICE A – DIAGRAMA DE CONTEXTO E LISTA DE EVENTOS

Na figura 31 o diagrama de contexto ilustra os requisitos atendidos pelo sistema, seus principais recursos e a sua interação com sistemas externos. A entidade "Plataforma de Agentes" é representada na prática pelo JADE. Em seguida a lista de eventos, apresentada na tabela 11, complementa o diagrama.

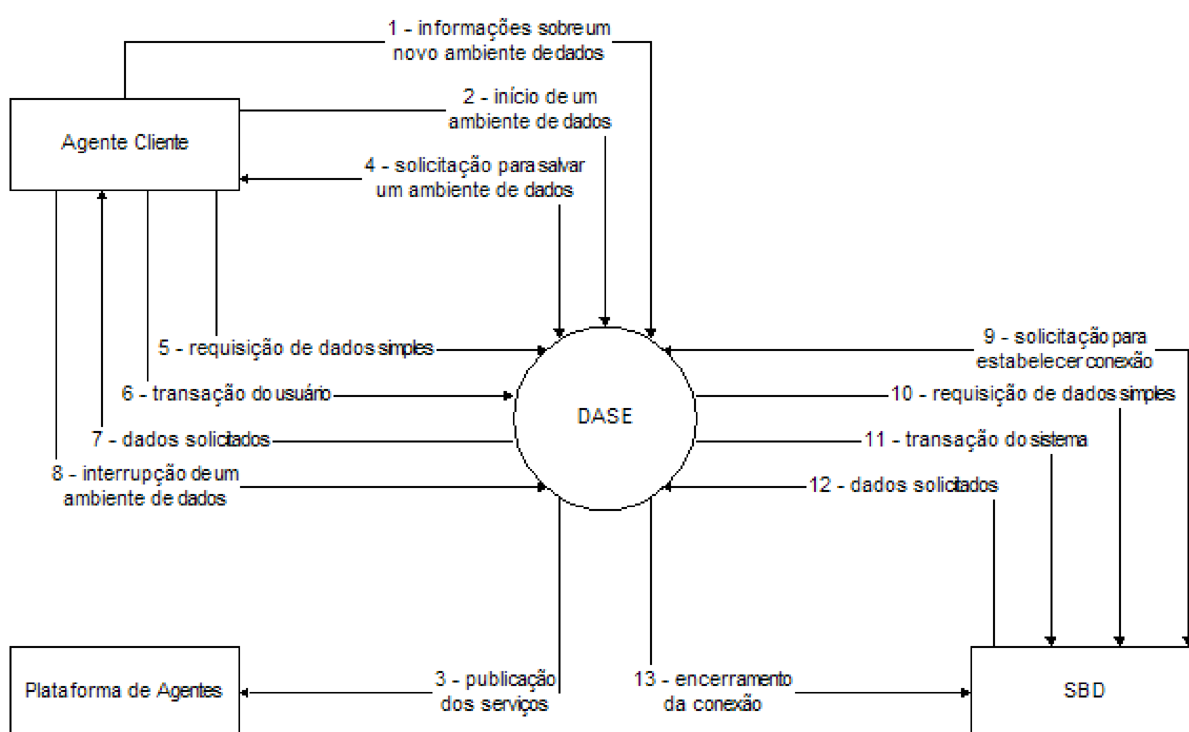

Figura 31 - Diagrama de contexto.

Tabela 11 - Lista de eventos.

	Evento	Entradas e Saídas
1	Agente Cliente solicita a criação de um novo ambiente de dados	Entrada: Informações referentes ao novo ambiente de dados
2	Agente Cliente solicita que um ou mais ambiente de dados sejam iniciados	Entrada: Identificação dos ambientes de dados a serem iniciados.

3	Plataforma de Agentes publica os serviços oferecidos pelos agentes DASE	Saída: Informações referentes à publicação dos serviços oferecidos pelos agentes do DASE aos agentes clientes.
4	Agente Ciente solicita que um ambiente de dados seja salvo	Entrada: Identificação do ambiente de dados a ser salvo. Saída: Arquivo contendo informações sobre o ambiente de dados salvo.
5	Agente Cliente faz requisição de dados simples	Entrada: Sentenças independentes de acesso a dados, referentes à requisição de dados solicitada.
6	Agente Cliente faz requisição de dados através da definição de uma transação do usuário	Entrada: Transação do usuário contendo as sentenças correlacionadas de acesso a dados, referentes à requisição de dados solicitada.
7	Agente Cliente recebe o resultado de uma requisição de dados	Saída: O resultado de uma requisição de dados (conjunto de dados ou alguma informação relacionada a um ou mais erros).
8	Agente Cliente solicita que um ou mais ambiente de dados sejam interrompidos	Entrada: Identificação dos ambientes de dados a serem interrompidos.
9	SDB recebe solicitação de estabelecimento de conexão	Saída: Requisição de estabelecimento de conexão com o SBD especificado a partir de informações contidas no ambiente de dados. Entrada: Conexão com o SBD ou alguma informação relacionada a um ou mais erros.
10	SBD recebe requisição de dados simples	Saída: Sentenças independentes de acesso a dados, referentes à requisição de dados solicitada pelo Agente Cliente.
11	SBD recebe requisição de dados através de uma transação do sistema	Saída: Transação do sistema contendo as sentenças correlacionadas de acesso a dados, referentes à requisição de dados solicitada pelo Agente Cliente.
12	SBD envia ao DASE o resultado de uma requisição de dados	Entrada: O resultado de uma requisição de dados (conjunto de dados ou alguma informação relacionada a um ou mais erros).
13	SBD recebe solicitação de encerramento de conexão	Saída: solicitação ao SBD de encerramento de conexão.

APÊNDICE B – CASOS DE USO

A figura 32 traz o diagrama de casos de uso do DASE. Neste diagrama “Plataforma de Agentes” representa na prática o JADE. Em seguida cada caso de uso é descrito.

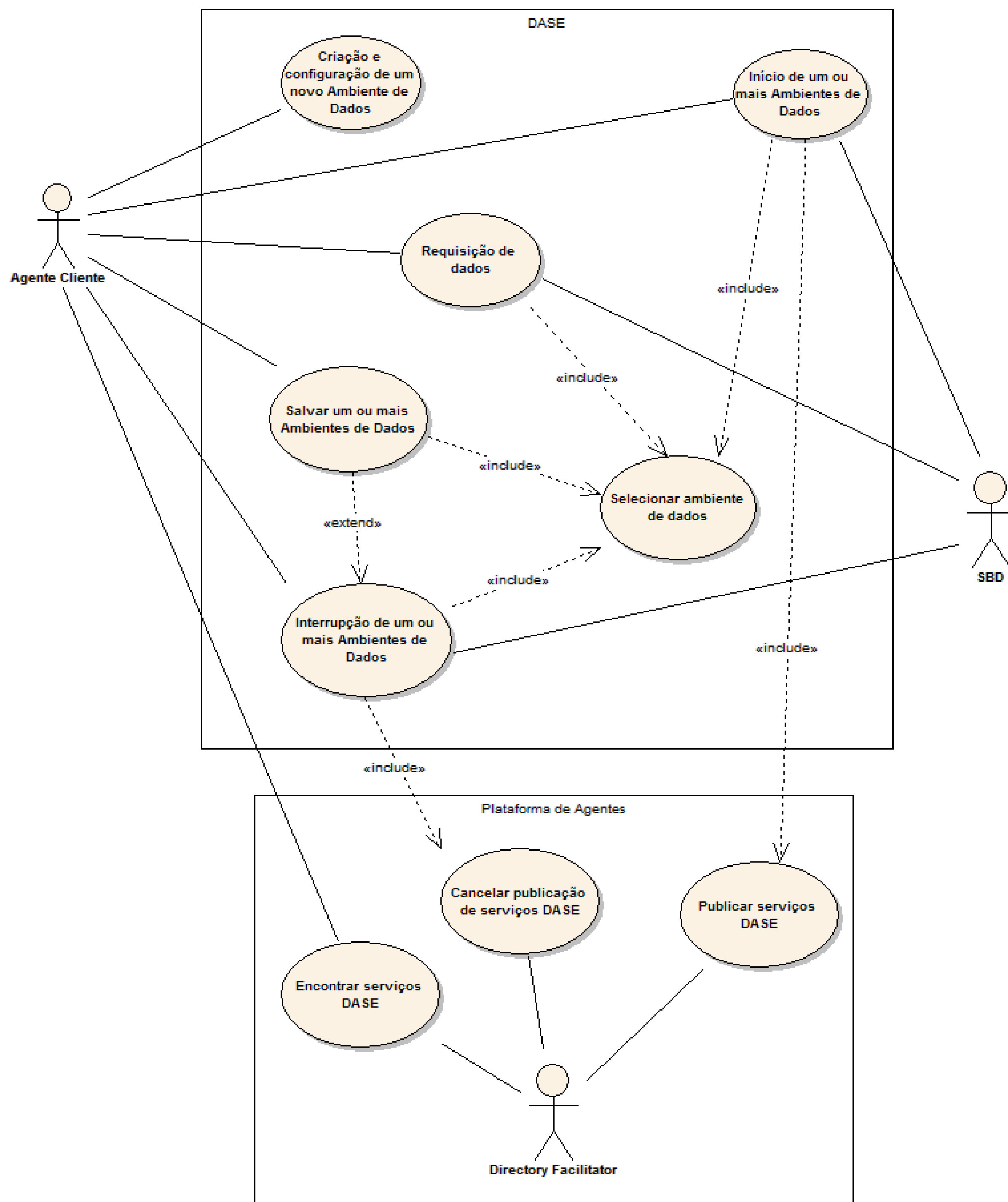

Figura 32 - Diagrama de casos de uso.

Tabela 12 - Caso de uso “Criação e configuração de um novo ambiente de dados”.

Ator principal	Agente Cliente
Cenário Principal 1. Agente Cliente cria um novo descritor do ambiente de dados 2. Agente Cliente especifica detalhes referentes ao SBD utilizado pelo ambiente de dados 3. Agente Cliente especifica em que nós do DASE o novo ambiente de dados estará disponível 4. Agente Cliente especifica recursos do DASE ativados no novo ambiente de dados 5. Agente Cliente solicita a inclusão do ambiente de dados junto ao DASE 6. DASE inclui o novo ambiente de dados	
Cenários Alternativos 2a: Carregar informações de um ambiente de dados previamente salvo 1. Agente Cliente solicita ao DASE que configure o novo ambiente de dados a partir de informações em um arquivo XML, de um ambiente de dados previamente salvo 2. Ir ao 6º passo do cenário principal 2b: O SBD escolhido é distribuído 1. Agente Cliente especifica detalhes referentes ao SBD distribuído utilizado pelo ambiente de dados 2. Agente Cliente especifica o(s) nó(s) do SBD utilizado 3. Ir ao 3º passo do cenário principal 5a: Especificação inválida de ambiente de dados 1. DASE informa ao Agente Cliente que há algo de errado com o ambiente de dados 2. A inclusão não é realizada 3. Voltar ao 2º passo do cenário principal	

Tabela 13 - Caso de uso “Início de um ou mais ambientes de dados”.

<table>
<tr><td>Ator principal</td><td>Agente Cliente</td></tr>
<tr><td>Outros atores envolvidos</td><td>SBD</td></tr>
<tr><td colspan="2">Cenário Principal

1. Agente Cliente seleciona o(s) ambiente(s) de dados a ser(em) iniciado(s)
2. Agente Cliente solicita ao DASE que inicie os serviços dos ambientes de dados
3. DASE inicia os serviços de cada ambiente de dados
4. DASE publica os serviços de cada ambiente de dados</td></tr>
<tr><td colspan="2">Cenários Alternativos

3a: Ambiente de dados não pode ser iniciado
1. DASE informa ao Agente Cliente que há algo de errado com o ambiente de dados
2. Um, ou mais, ambiente de dados não é iniciado
3. Voltar ao 1º passo do cenário principal</td></tr>
</table>

Tabela 14 - Caso de uso “Salvar um ou mais ambientes de dados”.

<table>
<tr><td>Ator principal</td><td>Agente Cliente</td></tr>
<tr><td colspan="2">Cenário Principal

1. Agente Cliente seleciona o(s) ambiente(s) de dados a ser(em) salvo(s)
2. Agente Cliente especifica detalhes de cada ambiente de dados referentes a sua salvação (em um arquivo XML)
3. Agente Cliente solicita ao DASE que realize o salvamento dos ambientes de dados
4. DASE realiza o salvamento de cada ambiente de dados</td></tr>
<tr><td colspan="2">Cenários Alternativos

3a: Especificação inválida de ambiente de dados
1. DASE informa ao Agente Cliente que há algo de errado com o ambiente de dados
2. O salvamento não é realizado
3. Voltar ao 2º passo do cenário principal</td></tr>
</table>

Tabela 15 - Caso de uso “Requisição de dados”.

<table>
<tr><td>Ator principal</td><td>Agente Cliente</td></tr>
<tr><td>Outros atores envolvidos</td><td>SBD</td></tr>
<tr><td colspan="2">Cenário Principal
1. Agente Cliente seleciona o ambiente de dados que deseja acessar
2. Agente Cliente define uma sentença SQL a ser utilizada na requisição de dados
3. Agente Cliente solicita ao DASE que realize o acesso aos dados
4. DASE realiza o serviço de acesso a dados solicitado pelo Agente Cliente
5. DASE envia ao Agente Cliente os resultados do acesso a dados solicitado</td></tr>
<tr><td colspan="2">Cenários Alternativos
2a: Definição de uma transação do usuário
1. Agente Cliente define uma ou mais sentenças SQL a serem utilizadas na requisição de dados
2. Agente Cliente define uma transação do usuário a partir das sentenças SQL
4a: O acesso aos dados não foi possível
1. DASE recupera informação de erro e a inclui nos resultados
2. DASE retorna resultados ao Agente Cliente comunicando a falha e o seu motivo
3. Voltar ao 1º passo do cenário principal</td></tr>
</table>

Tabela 16 - Caso de uso “Interrupção de um ou mais ambientes de dados”.

Ator principal	Agente Cliente
Outros atores envolvidos	SBD
Cenário Principal 1. Agente Cliente seleciona o(s) ambiente(s) de dados a ser(em) interrompido(s) 2. Agente Cliente solicita ao DASE que interrompa os serviços dos ambientes de dados 3. DASE cancela a publicação dos serviços referentes aos ambientes de dados a serem interrompidos 4. DASE interrompe os serviços de cada ambiente de dados	
Cenários Alternativos 2a: Antes é solicitado que os ambientes de dados sejam salvos 1. Agente Cliente informa ao DASE que antes de interromper os ambientes de dados é preciso salvá-los 2. Voltar ao 2º passo do cenário principal 4a: Ambiente de dados não pode ser interrompido 1. DASE informa ao Agente Cliente que há algo de errado com o ambiente de dados 2. DASE não interrompe um, ou mais, ambiente de dados 3. DASE republica os serviços dos ambientes de dados que tiveram sua interrupção cancelada 4. Voltar ao 1º passo do cenário principal	

Tabela 17 - Caso de uso “Selecionar ambiente de dados”.

Ator principal	Agente Cliente
Cenário Principal	
1. Agente Cliente solicita ao DASE que lhe permita escolher um ambiente de dados dentre os existentes 2. DASE retorna ao Agente Cliente algo que o possibilite escolher o ambiente de dados entre todos os existentes 3. Agente Cliente informa ao DASE o ambiente de dados escolhido	
Cenários Alternativos	
1a: Agente Cliente faz uma escolha específica de ambiente de dados 1. Agente Cliente especifica alguma informação que ajuda a identificar o ambiente de dados desejado 2. DASE retorna ao Agente Cliente os ambientes de dados que satisfazem o que foi informado anteriormente pelo Agente Cliente 3. Voltar ao 3º passo do cenário principal 1b: Não há nenhum ambiente de dados 1. DASE informa ao Agente Cliente que nenhum ambiente de dados foi criado ainda	

Tabela 18 - Caso de uso “Publicar serviços DASE”.

<table>
<tr><td>Ator principal</td><td>Directory Facilitator</td></tr>
<tr><td colspan="2">Cenário Principal

1. DASE solicita ao Directory Facilitator que publique os serviços DASE relacionados a um determinado ambiente de dados
2. Directory Facilitator recebe as informações e realiza a publicação
3. Directory Facilitator avisa ao DASE que a publicação ocorreu normalmente</td></tr>
</table>

Tabela 19 - Caso de uso “Encontrar serviços DASE”.

<table>
<tr><td>Ator principal</td><td>Agente Cliente</td></tr>
<tr><td>Outros atores envolvidos</td><td>Directory Facilitator</td></tr>
<tr><td colspan="2">Cenário Principal

1. Agente Cliente informa ao Directory Facilitator quais serviços DASE deseja encontrar
2. Directory Facilitator localiza os serviços publicados que correspondem à solicitação do Agente Cliente
3. Directory Facilitator retorna ao Agente Cliente os agentes responsáveis por prestar os serviços especificados</td></tr>
<tr><td colspan="2">Cenários Alternativos

2a: Serviços não encontrados
1. Directory Facilitator não encontra serviços publicados que correspondem à solicitação do Agente Cliente
2. Directory Facilitator informa ao Agente Cliente que não encontrou qualquer serviço
3. Voltar ao 1º passo do cenário principal</td></tr>
</table>

Tabela 20 - Caso de uso “Cancelar publicação de serviços DASE”.

<table>
<tr><td>Ator principal</td><td>Directory Facilitator</td></tr>
<tr><td colspan="2">Cenário Principal

1. DASE solicita ao Directory Facilitator que cancele a publicação dos serviços DASE relacionados a um determinado ambiente de dados
2. Directory Facilitator recebe as informações e realiza o cancelamento da publicação
3. Directory Facilitator avisa ao DASE que o cancelamento da publicação ocorreu normalmente</td></tr>
<tr><td colspan="2">Cenários Alternativos

2a: Serviços não encontrados
1. Directory Facilitator não encontra serviços publicados que correspondem à solicitação do DASE
2. Directory Facilitator informa ao DASE que não encontrou qualquer serviço
3. Voltar ao 1º passo do cenário principal</td></tr>
</table>