Ministério da
Ciência e Tecnologia

INPE-15773-TDI/1516

AGENTES MÓVEIS PARA APOIO AO SISTEMA DE CONTROLE DE SATÉLITES DISTRIBUÍDO E DINÂMICO

Josué Oliveira Silva

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Tatuo Nakanishi e João Bosco Schumann Cunha, aprovada em 11 de agosto de 2000.

Registro do documento original:
<http://urlib.net/sid.inpe.br/mtc-m18@80/2008/11.13.17.07>

INPE
São José dos Campos
2009

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6911/6923
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

Ministério da
Ciência e Tecnologia

INPE-15773-TDI/1516

AGENTES MÓVEIS PARA APOIO AO SISTEMA DE CONTROLE DE SATÉLITES DISTRIBUÍDO E DINÂMICO

Josué Oliveira Silva

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Drs. Tatuo Nakanishi e João Bosco Schumann Cunha, aprovada em 11 de agosto de 2000.

Registro do documento original:
<http://urlib.net/sid.inpe.br/mtc-m18@80/2008/11.13.17.07>

INPE
São José dos Campos
2009

Dados Internacionais de Catalogação na Publicação (CIP)

Silva, Josué Oliveira.
S38a Agentes móveis para apoio ao sistema de controle de satélites distribuído e dinâmico / Josué Oliveira Silva. – São José dos Campos : INPE, 2009.
263p. ; (INPE-15773-TDI/1516)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2000.
Orientadores : Drs. Tatuo Nakanishi e João Bosco Schumann Cunha.

1. Inteligência artificial. 2. JAVA. 3. Controle de satélite. 4. Sistema distribuído. 5. Agentes móveis. I.Título.

CDU 004.048

Aprovado pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre** em **Computação Aplicada.**

Dr. Tatuo Nakanishi — Orientador/Presidente

Dr. João Bosco Schumann Cunha — Orientador

Dr. Solon Venâncio de Carvalho — Membro da Banca

Dr. Carlos Ho Shih Ning — Membro da Banca

Drª Maria Carolina Monard — Membro da Banca Convidado

Candidato: Josué Oliveira Silva

São José dos Campos, 11 de agosto de 2000.

AGRADECIMENTOS

A Deus cuja mão e graça, e sob todos os aspectos, têm me fortalecido diuturnamente.

Aos doutores Tatuo Nakanishi e João Bosco S. Cunha que, sem economias na dosagem de sabedoria, “profissionalismo” e camaradagem, conduziram a orientação deste trabalho.

A meus pais David e Simey por toda “herança” espiritual, familiar e educacional; às minhas irmãs Jane e Joyce pela amizade e carinho especiais; e à Kelli pelo amor e companhia indispensáveis.

A meus familiares, em especial aos tios Azor e Benedito, às tias Laide, Marlei e Marci; e aos primos César, Gustavo, Rafael, Carina, Lilian, Débora, Mariana e Juliana pela contribuição e amparo.

Ao amigo Maurício G. V. Ferreira por todo suporte técnico e investimento pessoal.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo subsídio financeiro.

RESUMO

Acompanhando a tendência mundial das companhias de Tecnologia de Informação e de institutos de pesquisas renomados, os sistemas computacionais preconizam a assimilação vantajosa dos atributos de distribuição e dinamismo às arquiteturas atualmente em desenvolvimento. À bordo de perspectivas promissoras associadas a tais atributos, o Sistema de Controle de Satélites do Instituto Nacional de Pesquisas Espaciais, em particular, avança rumo à "reformulação" de sua infraestrutura ao propor uma modelagem na qual as aplicações, além de se submeterem aos "ditames" da tecnologia de objetos distribuídos (ou seja, as aplicações são modeladas como objetos que podem interoperar em ambientes de redes com diferentes plataformas), apresentam características que permitem a sua "mobilidade" (ou seja, as aplicações podem atuar como objetos móveis). Concomitante à tal "reestruturação", surge a necessidade por serviços específicos e indispensáveis à efetivação satisfatória das funções idealizadas para o novo modelo. Dentre outros planejados, se destaca o serviço de monitoração dos objetos que compõem a arquitetura distribuída e dinâmica. A partir deste contexto e de sinais evidentes do inter-relacionamento da tecnologia de objetos distribuídos com a recente e promissora tecnologia de agentes móveis, o presente trabalho advoga o estudo e a consequente aplicação prática de um serviço de monitoração de objetos através da integração de um ambiente distribuído de agentes, os quais desfrutam de habilidades peculiares como autonomia e mobilidade, ao sistema de controle de satélites baseado em objetos distribuídos.

MOBILE AGENTS ON SUPPORT TO DISTRIBUTED AND DYNAMIC SATTELITE CONTROL SYSTEM

ABSTRACT

Following world-wide trend of both Information Technology Companies and renowned research institutes, computer systems preconize advantageous of distribution attributes and dynamism to the architectures currently under development. On board of promising prospects associated with such attributes, the Satellite Control System of the National Institute for Space Research, in particular, moves towards the reformulation of its infrastructure by the proposition of a modeling in which the applications – in addition to their submitting to the "dictates" of distributed objects technology (that is, the applications are shaped as objects that can interoperate in mixed networks with different platforms) – have present features that allow its mobility (that is, the application can act as mobile objects). Concomitant to this "restructuring" there is a need for essential and specific services to satisfactorily accomplish the idealized functions for the new model. Among others planned services, it highlights the one service of monitoring the objects that make up the distributed and mobile architecture. From this context and clear evidence of inter-relationships of distributed objects technology with the recent promising technology of mobile agents, this work advocates the study and practical application through implementation of a service for monitoring objects based on the integration of a distributed agents environment, which enjoy unique abilities such as autonomy and mobility, to the Satellite Control System based on distributed objects.

SUMÁRIO

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE SIGLAS E ABREVIATURAS

ACL	Agent Communication Language
ADA	Ambiente Distribuído de Agentes
APD	Ambiente de Processamento Distribuído
API	Application Programming Interface
ASDK	Aglet Software Development Kit
AWT	Advanced Windowing Toolkit
BDGO	Base de Dados Global dos Objetos
BDI	Belief-Desire-Intention Model
CCS	Centro de Controle de Satélites
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DoS	Denial of Service
DCOM	Distributed Component Object Manager
DES	Data Encryption Standard
DOT	Distributed Object Technology
DT	Distributed Technology
FIPA	Foundation for Intelligent Physical Agents
FTP	File Transfer Protocol
GUID	Global Unique Identifier
GUI	Graphic User Interface
IDC	International Data Corporation
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronic Engineers
IIOP	Internet Interoperable ORB Protocol
IN	Intelligent Networks
IOR	Interoperable Object Reference
IP	Internet Protocol
JDBC	Java Data Base Connectivity

JFC	Java Foundation Classes
KQML	Knowledge Query and Manipulation Language
MAF	Mobile Agent Facility
MASIF	Mobile Agent System Interoperability Facility
MAT	Mobile Agent Technology
NMS	Network Management Systems
OMG	Object Management Group
OOP	Object Oriented Programming
ORB	Object Request Broker
OT	Object Technology
PAM / MAP	Plataforma de Agentes Móveis / Mobile Agent Platform
PDA	Personal Digital Agents
PKI	Public Key Infrastructure
QoS	Quality of Service
RAD	Rapid Application Development
RMI	Remote Method Invocation
RPC	Remote Procedure Calling
RSA	Rivest-Shamir-Adleman (*sistema de autenticação e criptografia*)
SICSD	Software de Controle de Satélites Distribuído e Dinâmico
SSL	Secure Socket Layer
SMS	Short Message Services
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TI	Tecnologia da Informação
TMS	Objeto que representa o serviço de Telemetria
UDP	User Datagram Protocol
URL	Universal Resource Locator
WT	Web Technology
WWW	Wide World Web

CAPÍTULO 1
INTRODUÇÃO

A segunda metade dos anos 90 presenciou a luta por sobrevivência da maior parte das organizações de Tecnologia da Informação (TI), frente aos desafios em relação às exigências de qualidade dos produtos e serviços aos clientes.

Tais empresas foram pressionadas a gerenciar ambientes heterogêneos e complexos que incorporavam, por sua vez, hardware, software, aplicações, redes e sistemas de banco de dados díspares, ou ainda integrar sistemas legados, originalmente desenvolvidos há 10 ou 20 anos atrás, com as mais recentes tecnologias cliente-servidor e Internet.

Num retrospecto, não tão distante, a comunidade da Tecnologia da Informação presenciou nos últimos 15 anos mudanças significativas e vitais na maneira de se projetar, desenvolver e manter sistemas de informações corporativas. Tal período iniciou com o nascimento dos sistemas monolíticos cuja arquitetura é descrita abaixo na Figura 1.1.

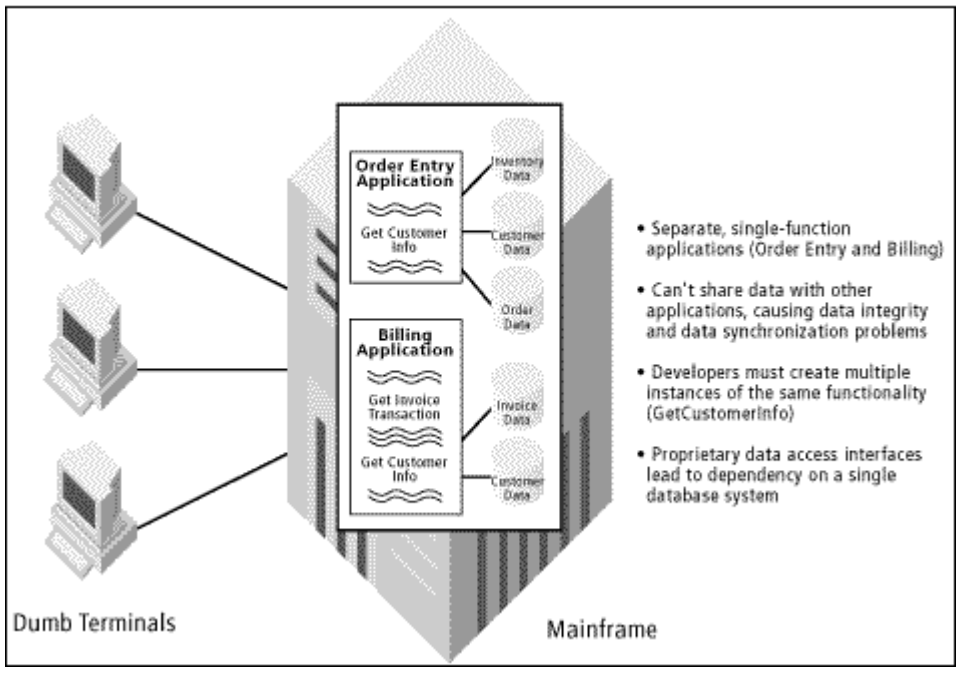

FIGURA 1.1: Arquitetura para aplicativos em Mainframe “Monolíticos”
FONTE: Visigenic (1997).

Tais ambientes, por razões de ineficiência e alto custo, acrescidas pela notada convergência de tecnologias – redes, computadores pessoais de baixo custo, interface gráfica para o usuário (GUI) e banco de dados relacionais, foram superados pela computação cliente-servidor. Este modelo prometia, descentralizando os sistemas monolíticos em componentes, facilitar o desenvolvimento e manutenção de aplicações complexas.

A arquitetura cliente-servidor é essencial, mas não suficiente. As atuais ferramentas e técnicas submetem-se a problemas como escalabilidade, modularidade, granularidade e manutenibilidade das aplicações (Visigenic 1997).

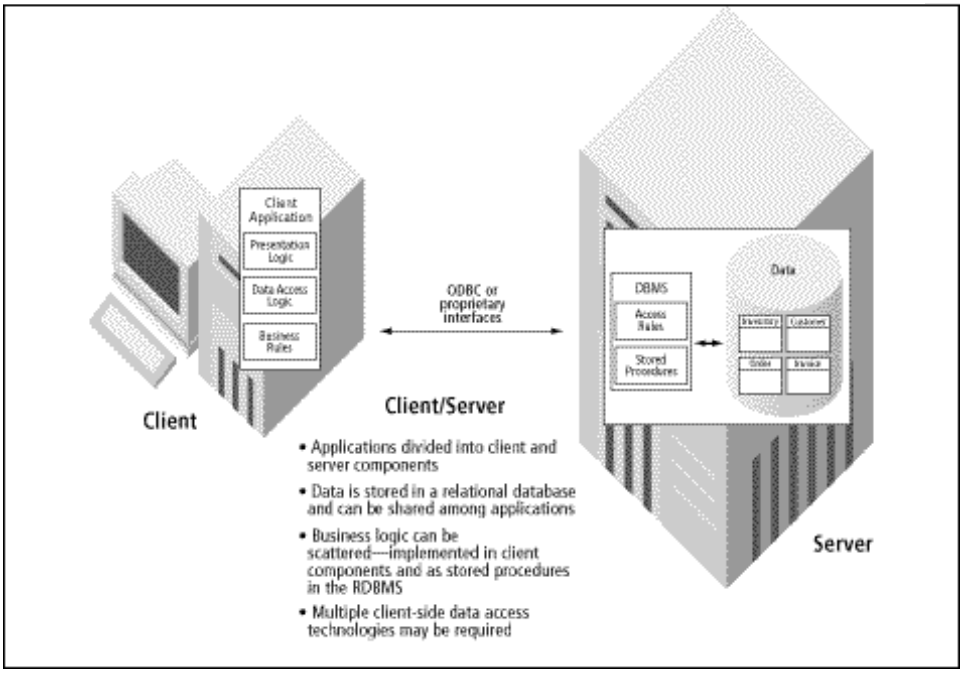

FIGURA 1.2: Arquitetura para aplicativos Cliente-Servidor
FONTE: Visigenic (1997).

Em suma, a combinação harmônica da integridade semântica e cognitiva dos objetos com o potencial de distribuição da arquitetura cliente-servidor se acopla aos trilhos revolucionários da Internet gerando o novo paradigma da computação de objetos distribuídos. A tecnologia Distributed Object Technology (DOT) rompe as barreiras dos sistemas monolíticos através da disponibilização de mecanismos, compreendendo

transparentemente as tecnologias já mencionadas, rumo à distribuição e globalização no desenvolvimento de aplicações.

A Tecnologia de Objetos Distribuídos (DOT) abrange três vertentes que, sinergicamente unidas, resultaram em um conceito maior do que a soma de suas partes. Em ordem de emergência, despontaram: Tecnologia de Objetos -> Tecnologia de Distribuição -> Tecnologia Web.

A **Tecnologia de Objetos** - "Object Technology" (**OT**) - foi introduzida ao círculo principal da computação no fim dos anos 70 por Adele Goldberg e Alan Kay com uma linguagem chamada Smalltalk (Goldberg 1983). No modelo orientado a objetos, sistemas são visualizados como objetos cooperantes que encapsulam estrutura e comportamento pertencendo a classes hierarquicamente construídas. Nos últimos 20 anos os benefícios da tecnologia orientada a objetos têm sido demonstrados.

A orientação a objetos modificou a maneira de se construir e manter sistemas. Existem ferramentas e experiências substanciais para Análise Orientada a Objetos, Projeto Orientado a Objetos e Programação Orientada a Objetos. A transição de modelos e abordagens estruturalmente tradicionais não se completou, nem a OT tem sido totalmente explorada, especialmente nos sistemas legados que pré-datam os objetos. Mas a OT, maturando rapidamente, se difunde em aceitação como tecnologia que lida com complexidade, realça manutenibilidade, promove o reuso e reduz os custos de ciclo de vida do software.

A **Tecnologia de Distribuição** - "Distributed Technology" (**DT**), a qual envolve computadores autônomos, não compartilhando memória física, conectados através de uma rede, data o início dos anos 80. O advento de "CPUs" menores, mais poderosas e baratas precipitou no interesse de migrar os aplicativos dos "mainframes" para computadores pessoais e estações de trabalho e ao mesmo tempo distribuir funcionalidade pelos múltiplos computadores interligados.

Com o passar do tempo, a noção de distribuição modificou-se a partir de ambientes rígidos e geograficamente restritos com máquinas homogêneas para estruturas flexíveis, robustas e geograficamente ilimitadas com máquinas heterogêneas. Os sistemas distribuídos tradicionais empregavam o modelo cliente-servidor

freqüentemente implementado com os mecanismos de Remote Procedure Calling (RPC).

Os modelos atuais de distribuição evoluíram a partir da arquitetura tradicional passando a utilizar a tecnologia “middleware”. Enquanto os detalhes e definições - ver (Rymer 1996) deste conceito variam consideravelmente, os produtos “middleware” geralmente fornecem uma infra-estrutura “runtime” de serviços para a interoperação de componentes (objetos). Vale registrar o foco dispensado pela DT em revestir o ambiente computacional com a transparência de localização entre os computadores e objetos.

A **Web Technology** (**WT**) nasceu nos anos 90 e rapidamente causou uma explosão no uso da Internet. O advento dos “web browsers” e máquinas de busca provocou excitação comparável à introdução dos computadores pessoais em meados dos anos 70. A WT universalizou e globalizou a computação de um modo nunca visto. Não se anexa referências do aspecto mais popular da tecnologia somente vislumbrado na habilidade de criar páginas de informação que podem ser acessadas em qualquer ponto do planeta com uma interface humana atraente e descomplicada.

Além disso, interessa desvendar os benefícios e/ou concentrar esforços da tecnologia para a construção de grandes sistemas corporativos ou para reengenharia dos sistemas antigos. O surgimento da linguagem orientada a objetos Java em 1995 trouxe, entre outros inumeráveis destaques, as “applets” que provêem “conteúdo executável” à WT, antes estática.

A tecnologia de Objetos Distribuídos, portanto, fundamentalmente revolucionou o desenvolvimento de aplicações. Acoplado a uma infra-estrutura robusta de comunicação, tal modelo divide as aplicações cliente-servidor, ainda “monolíticas”, em componentes auto-gerenciáveis ou objetos que podem interoperar através de sistemas operacionais e redes heterogêneos.

Destaca-se desde 1989, dentro do contexto de distribuição de objetos, o trabalho do grupo Object Management Group (OMG) – consórcio composto por aproximadamente 700 companhias – ocupando-se na especificação de uma arquitetura para um “software bus” aberto por meio do qual os objetos implementados em

diferentes “tecnologias” pudessem interoperar através de redes de computadores e sistemas operacionais heterogêneos.

Como fruto do esforço imensurável da OMG, foi anunciada a arquitetura do Commom Object Request Broker Architecture (CORBA). A especificação CORBA descreve um “open software bus” conhecido como Object Request Broker (ORB) o qual provê uma infra-estrutura para a computação de objetos distribuídos permitindo que aplicações cliente se comuniquem com objetos remotos através da invocação (estática ou dinâmica) de operações. A DOT e o padrão CORBA, em particular, provêem capacidade extremamente poderosa, especialmente em relação a interoperabilidade de hardware, software e linguagens de programação díspares. Num ambiente que requeira as melhores soluções para cada problema, CORBA pode atuar como “cola” entre tecnologias diferentes (Nelson 1999).

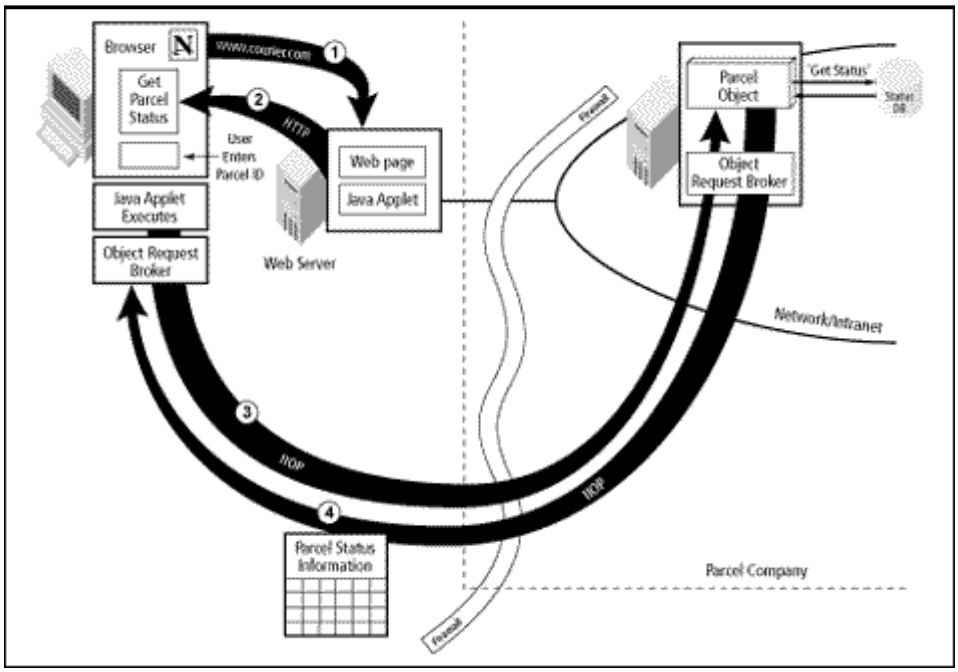

FIGURA 1.3: Nova arquitetura “Internet-enabled” e distribuída
FONTE: Visigenic (1997).

Um panorama mais detalhado sobre a tecnologia de objetos distribuídos, enriquecido por uma explanação minuciosa quanto à conceitos e padrões, é o foco principal do trabalho recentemente apresentado no corpo da dissertação de mestrado da pesquisadora Samira R. da Costa (Samira 2000). Servindo como suporte de referência e conceituação, e visto que foi idealizado e concebido dentro do mesmo ambiente de

aplicação, destaca-se a contribuição proporcionada pelo estudo de Samira para o desenvolvimento da presente dissertação.

Paralelamente às considerações levantadas até o momento, se percebe o burburinho de líderes do mercado de rede, por exemplo Novell, em busca de plataformas que injetem "vida" dentro dos códigos e estruturas de dados, transformando-os em objetos distribuídos que possam ser utilizados dinamicamente. A migração de modelos estáticos para dinâmicos trará benefícios, tornando os sistemas corporativos mais robustos e úteis aos simplificar a sua administração enquanto flexibiliza as redes de computadores.

Encerrando, portanto, o "ciclo de inovações", entra em cena a Tecnologia de Agentes, uma das mais recentes e interessantes áreas de pesquisa em Tecnologia da Informação. Alavancando a infra-estrutura de objetos distribuídos, os agentes podem proporcionar um grau elevado de autonomia para o software, permitindo que "componentes-software" e aplicativos se comportem "pro-ativamente" em termos de sua habilidade de satisfazer requisitos específicos do usuário. O intento da tecnologia é prover os meios para o tratamento da crescente complexidade das aplicações atribuindo um foco "dinâmico" ao paradigma de objetos.

Segundo estudos recentes do instituto International Data Corporation (Garone 1998) que intensificou esforços no exame da tecnologia, os agentes apresentam algumas características peculiares que o definem em geral:

a) **Agentes são autônomos:** tal característica originou-se a partir do desejo de se designar agentes que "representem" o usuário em tarefas para as quais possuem capacidade suficiente de tratamento e efetivação.
b) **Agentes podem "sentir" o ambiente no qual estão inseridos**: "coletando" informações de um ambiente particular, um agente pode desenvolver aprendizado e orientar-se pelo conhecimento adquirido modificando suas ações.
c) **Agentes podem realizar ações**: Fundamentado em informações colhidas, um agente pode decidir como agir em determinadas formas.
d) **Agentes são móveis**: agentes que apresentam aptidão para migrar entre diferentes pontos de uma rede de computadores a fim de realizar localmente tarefas pelas quais são responsáveis. A maioria dos atuais sistemas distribuídos se fundamenta em entidades de

software estaticamente instaladas em determinados nós de uma rede. A comunicação entre estes componentes estáticos requer continuamente o suporte e a presença de uma estrutura de rede. Logo, as falhas de rede podem comprometer ("breakdown") parcial ou completamente um sistema. Migrando e se comunicando localmente com os servidores, os agentes podem reduzir a dependência da estrutura de rede apenas para os intervalos de tempo requeridos nas suas migrações (Grasshopper 1998[a]). A Tecnologia de Agentes Móveis - Mobile Agent Technology (**MAT**) - focaliza prioritariamente os agentes móveis, autônomos e assíncronos. O termo "agente inteligente" se baseia no paradigma da Inteligência Artificial Distribuída que representa outra "religião" dentro da área de pesquisa de agentes (Magedanz 1998b).

e) Agentes são "orientados a meta": não apenas reagindo ao seu ambiente, os agentes podem agir com propósito e de acordo com objetivos específicos na escolha e decisão de suas ações.

Num ambiente de computação distribuído baseado em agentes móveis, os " hosts" devem prover um tipo de "docking station[1]" atuando como infra-estrutura local ou plataforma para agentes. Tal plataforma é responsável pelo despacho, recepção e "residência" (temporária ou persistente) dos agentes além de fornecer serviços, recursos e suporte "runtime" necessários. As principais tarefas de uma plataforma para agentes podem ser sintetizadas como segue: Suporte à Mobilidade, Gerenciamento de Recursos, Suporte à Execução, Suporte para Comunicação, Serviço de Informação e Diretório, Suporte à Segurança, Serviço de Apoio e Despacho de Eventos e Suporte à Tolerância a Falhas.

A aplicação do paradigma de agentes móveis, permitindo a delegação dinâmica e remota de tarefas para entidades apropriadas, oferece um grande potencial de distribuição e flexibilidade. Atualmente existe um consenso de que a Tecnologia de Agentes Móveis não substitui a Tecnologia de Objetos Distribuídos mas surge como fator de incremento. Ambas oferecem vantagens em diferentes aspectos, as quais, coletivamente, atribuirão máxima flexibilidade no projeto de aplicações (Magedanz 1998b).

Nesta "visão integrada", a DOT permitiria a interoperabilidade e reusabilidade de componentes distribuídos e heterogêneos (serviços), enquanto que a MAT contribuiria para a distribuição (provisão) e extensibilidade dinâmicas dos componentes.

A faceta de "inteligência" da tecnologia de agentes habilita o reforço no comportamento autônomo e cooperativo dos componentes distribuídos.

A tendência difundida atualmente quanto à integração destas duas tecnologias – DOT & MAT – pode ser comprovada por companhias e institutos de pesquisa que avançam em direção à aplicabilidade dos agentes móveis em diversas áreas de sistemas distribuídos. Algumas áreas de aplicação, em particular, estão ganhando ímpeto neste contexto, entre as quais destacam-se: controle de serviços em redes móveis ou fixas (Baumer 1999), gerenciamento de redes (falhas, "accounting", configuração, performance, segurança) (Bieszczad 1998), ambientes ou sistemas de redes inteligentes – "Intelligent Network" (IN) – (Magedanz 1998b) e sistemas de informação para empresas virtuais (Papa 1998).

Seguindo os rumos destas inovações tecnológicas e científicas introduzidas, foi aprovada recentemente (Novembro 1999) no curso de Computação Aplicada do Instituto Nacional de Pesquisas Espaciais (INPE) a proposta para tese de doutorado ***"Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao Software de Controle de Satélites"*** (Ferreira 1999). Em síntese, a arquitetura idealizada na pesquisa de Ferreira propõe um estudo através do qual se apresenta uma nova abordagem, para o software de controle de satélites, baseada fundamentalmente na integração da tecnologia de objetos distribuídos e na tecnologia de agentes móveis.

Como conseqüência prevista e necessária ao desenvolvimento da arquitetura supra citada, nasceu a linha paralela de pesquisa evidenciada pelo título da presente dissertação. O apoio da tecnologia de agentes móveis, não se traduzindo apenas como tentativa de aplicação de uma área tecnológica amplamente testada e consagrada, prevê uma análise cuidadosa de diversos conceitos e tecnologias de suporte relevantes ao panorama das soluções distribuídas baseadas em agentes.

[1] "estação âncora", ambiente básico para execução

Estrutura da Dissertação

Concluindo este capitulo introdutório e objetivando estabelecer uma organização adequada ao desenvolvimento da dissertação, destaca-se a seqüência dos tópicos compilados em capítulos na seguinte ordem:

O Capítulo 2 apresentará, além de uma visão geral da pesquisa e do contexto da arquitetura proposta por Ferreira, a descrição dos objetivos definidos para a presente dissertação. A Tecnologia de Agentes, tema central do Capítulo 3, é dissecada em diversas "facetas" através das quais se concebe um panorama geral deste inédito e já difundido assunto. Aprofundando-se no terreno da tecnologia, o Capítulo 4 introduz a não menos alardeada Tecnologia de Agentes Móveis. O Capítulo 5 contribui na complementação de conceitos oferecendo uma comparação e subsequente avaliação das principais Plataformas de Agentes Móveis. Sustentado pelos resultados provenientes desta comparação, o Capítulo 6 evidencia os principais atributos da plataforma de agentes que se destacou entre as candidatas para uma futura fase inicial de implementação. O cerne do trabalho, ou melhor, a efetivação dos objetivos estipulados no Capítulo 2 é abordada dentro dos limites do Capítulo 7. O Capítulo 8 contém os detalhes da implementação prática de um ambiente para o protótipo modelado no capítulo anterior. As conclusões e considerações finais da dissertação são delineadas no Capítulo 9.

CAPÍTULO 2
MOTIVAÇÃO E OBJETIVOS

2.1 – Introdução

Sintonizado aos recentes avanços da ciência e tecnologia, alguns dos quais abordados no capítulo de introdução, o departamento Centro de Controle e Rastreio de Satélites (CCS) do Instituto Nacional de Pesquisas Espaciais passou a contabilizar recentemente mais um trabalho de pesquisa que, apesar de concentrar o foco de aplicação ao contexto local do instituto pode, sem exageros, trazer inúmeros benefícios para a comunidade científica na forma de acúmulo de "know-how" ou de propriedade intelectual frente às inovações abordadas.

Esta pesquisa encontra sustentação na tese de doutorado intitulada: "***Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao Software de Controle de Satélites***" (Ferreira 1999), a qual, oportuna e posteriormente, será referenciada neste trabalho, sem qualquer desmerecimento, como proposta ou arquitetura "***SICSD***".

Apresenta-se, em primeiro lugar, um resumo geral da proposta SICSD nos moldes do "paper" aprovado para publicação e apresentação em um simpósio internacional na França em Julho de 2000[1], para que se defina a seguir a motivação e sejam estipulados os objetivos da presente dissertação

2.2 – Resumo Geral da Proposta SICSD

Após uma longa fase de sistemas centralizados baseados em "mainframes", a quebra do processamento em cliente-servidor foi utilizada no software de controle de satélites com diversas vantagens. Atualmente, a tecnologia de Objetos Distribuídos divide aplicações cliente-servidor em componentes auto-gerenciáveis ou objetos que podem interoperar em ambientes de redes com diferentes plataformas, introduzindo vantagens adicionais como o aumento da disponibilidade e flexibilidade.

[1] 6th International Symposim Space Ops 2000 "Space operations at the start of the 3rd millenium"

Hoje, as aplicações que utilizam objetos distribuídos contam com uma infra-estrutura relativamente robusta de comunicação e de recursos de software como, por exemplo, os pacotes Distributed Component Object Manager (DCOM) da Microsoft e outros baseados no padrão CORBA. Além disso, idéias e mecanismos originados em outras áreas estão sendo utilizados no desenvolvimento de sistemas de software aplicativos. Em suma, a tecnologia de agentes, concebida nos domínios da Inteligência Artificial, é um exemplo da soma de resultados de áreas diferentes. Tal ferramenta de software pode ser utilizada na melhoria da monitoração da rede e das informações entre as partes dos aplicativos.

A Proposta SICSD tem como objetivo mostrar como o software para controle de satélites poderá se tornar uma aplicação distribuída com o apoio da tecnologia de agentes, salientando as vantagens desse direcionamento. A idéia consiste em organizar o software em objetos que poderão migrar dinamicamente de uma máquina para outra conforme as solicitações dos controladores e do estado da rede de computadores disponível para o controle de satélites.

Os objetos distribuídos irão disponibilizar os serviços comuns de qualquer aplicativo para controle de satélites tais como: processamento de telemetria e envio de telecomando. Estes objetos migram de um nó para outro dependendo do tipo e do número de solicitações que estejam recebendo e após a análise do estado de utilização da rede como um todo.

Todo o processo de migração de um objeto servidor será feito baseado em informações coletadas pelos agentes móveis ou fixos que estarão distribuídos pelo sistema. Os agentes estarão monitorando, periodicamente, as transações que chegam em um servidor.

As vantagens deste tipo de sistema são:

a) A ***concorrência*** na utilização do software poderá ser diminuída para melhorar o desempenho do sistema pois cópias de um mesmo objeto poderá estar em máquinas distintas para atender as solicitações dos usuários.

b) Aumento da ***confiabilidade*** do sistema possibilitando, por exemplo, que a falha de um computador não impeça o envio de telecomandos para o satélite e que a operação continue quase como se não tivesse ocorrido o problema.
c) ***Flexibilidade*** para atender as diferentes situações de controle de uma missão, como a de lançamento, de regime e de manobras atendendo a dinâmica do controle de satélites que varia ao longo do tempo.
d) O ***balanceamento de carga*** automático do processamento dos computadores também poderá significar ganhos com o ambiente dinâmico em comparação aos sistemas estáticos correntes. Por exemplo, um objeto servidor poderá migrar de um nó da rede para outro de acordo com a situação de "carga" de cada nó.
e) A ***disponibilidade*** de serviços é um motivo forte para se criar cópias de um objeto em diferentes nós. O processo é similar ao de migração, com a diferença de que o objeto continuará existindo no nó origem.

A arquitetura proposta modela a aplicação para controle de satélites através de objetos, os quais são distribuídos dentro de uma rede pré-definida. A localização inicial das instâncias desses objetos é definida na carga do sistema, momento a partir do qual eles podem migrar de uma máquina para outra de acordo com a demanda de solicitações de serviços, ou seja, não se fixam estática e permanentemente aos "sites" alocados de início.

Um protótipo da arquitetura proposta, em desenvolvimento, utiliza produtos que se submetem ao padrão CORBA. A Figura 2.1 ilustra alguns destes objetos que poderão estar interoperando em um sistema distribuído e dinâmico.

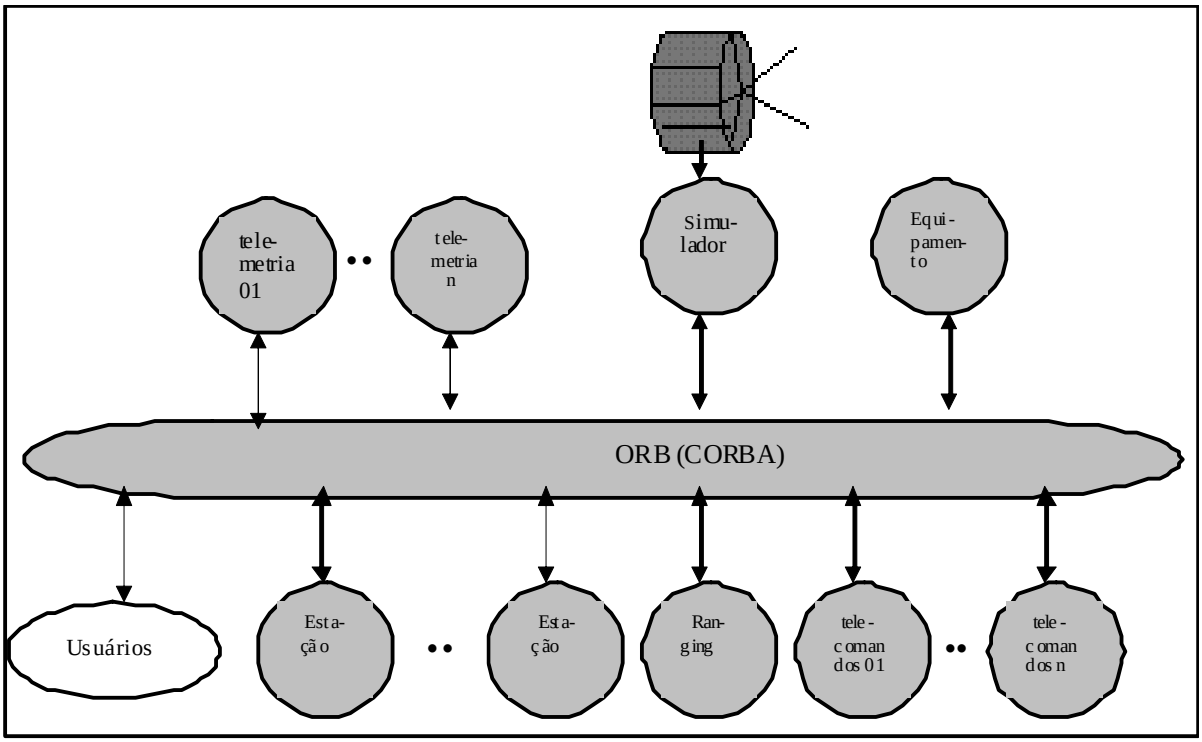

FIGURA 2.1: Sistema de Controle de Satélites distribuído e dinâmico
FONTE: Ferreira (1999).

Os objetos da Aplicação para Controle de Satélites se comunicam através de um "middleware" que implementa a especificação CORBA, podendo existir mais de uma cópia de um objeto instanciado em nós diferentes da rede:

Telemetria: Este objeto descreve o 'estado' interno do satélite, como por exemplo a voltagem de uma determinado circuito, o posicionamento de uma determinada chave (ligado ou desligado), etc.;

Telecomando: Contém os telecomandos que podem ser enviados para satélites;

Estação: Este objeto contém a descrição das estações utilizadas para recepção dos dados dos satélites que são controlados pelo INPE. Atualmente existem 2 (duas) estações: Cuiabá e Alcântara.

Ranging: Este objeto é responsável por calcular a medida de distância entre a terra e o satélite.

Equipamento: Contém a descrição dos equipamentos instalados para a recepção e transmissão de dados para o satélite.

Simulador: Este objeto é capaz de simular os possíveis "estados" de um satélite e disponibilizar estes dados através de um interface para todos os outros objetos. Entende-se por "estado" de um satélite as possíveis condições internas apresentadas pelo mesmo em um determinado instante.

Os objetos da Figura 2.1 são exemplos baseados nos atuais sistemas de controle de satélites servindo como ilustração do funcionamento da arquitetura proposta, ressaltando-se porém, que estudos futuros poderão indicar uma forma diferente para a "quebra" ou "decomposição" do sistema que determine assim, um conjunto ou modelo diferente de objetos. Os usuários interagem com os objetos distribuídos através de um "middleware". A solicitação de serviços dos usuários são recebidas pelo "broker" que possui a incumbência de localizar o objeto capaz de atender determinada solicitação.

A replicação de um objeto em mais de um nó da rede está relacionada com o número de usuários que utilizam os serviços disponíveis deste objeto. Por exemplo, o objeto telemetria é o mais solicitado visto que existem vários usuários interessados na visualização do "estado" interno do satélite. Os usuários responsáveis pelos painéis solares têm interesse pela visualização das telemetrias que mostram a posição do satélite em relação ao sol. Os usuários responsáveis pela bateria priorizam o acompanhamento da voltagem e da corrente gerada pela mesma. Outros objetos, o telecomando por exemplo não apresenta a necessidade de replicação por constatar-se que apenas um usuário, regularmente, deve utilizar seus serviços.

2.3 – Os Serviços da Arquitetura

A arquitetura proposta para o software de controle de satélites é composta pelos objetos da aplicação - telemetria, ranging, telecomandos, etc. - e pelos serviços disponíveis, ou seja, os serviços de Agentes, Persistência, Segurança e Balanceamento.

A Figura **2.2** apresenta uma visão da arquitetura proposta com os serviços.

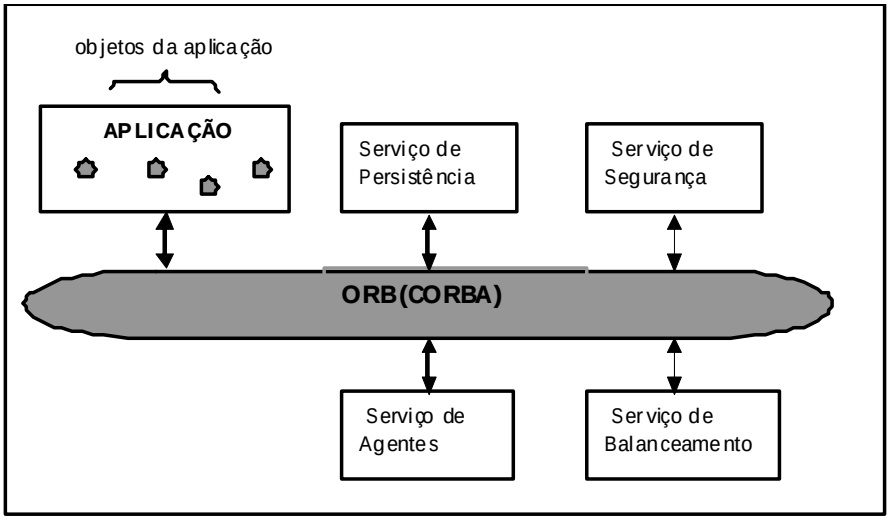

FIGURA 2.2: Uma visão dos serviços da arquitetura proposta
FONTE: Ferreira (1999).

2.3.1 – O Serviço de Agentes

Os agentes são responsáveis por manter atualizada todas informações, indispensáveis ao funcionamento dinâmico e flexível do software de controle de satélites, dos diferentes "sites" da rede. Várias informações são gerenciadas pelos agentes, dentre elas pode-se citar: a localização física de cada objeto instanciado; o estado de um objeto; podendo ser "ready" se o objeto está disponível para receber solicitações de serviços do sistema ou "failed", caso contrário; a carga da CPU e de I/O de um determinado "site"; o número de conexões atuais de um objeto, etc..

Os serviços dos agentes são implementados em cada nó pelos agentes fixos e móveis. O agente fixo, denominado *agente supervisor*, atua dentro do contexto como um ponto de referência em cada nó local. Uma vez instanciado, o agente supervisor, responsabiliza-se pelo controle do armazenamento, controle de acesso, atualizações, disseminação e disponibilização das informações referentes aos objetos instalados naquela máquina. Todas estas informações são armazenadas nas *tabelas de configurações*. Ele administra a estrutura de maneira que as informações geridas por ele não se restrinjam ao contexto limitado de apenas um nó. Tal abordagem permite o acesso às *tabelas de configurações* distribuídas por todos os nós do sistema.

Os agentes móveis têm como atividade principal a disseminação das informações coletadas pelo agente supervisor por todos os nós da rede. Seu funcionamento é similar ao do agente supervisor, ou seja, de tempos em tempos ele é acionado, recupera as informações gerenciadas pelo agente supervisor e transfere para todos os outros nós da rede, até retornar ao nó de origem. Cada agente móvel tem uma base de dados que contém os nós por onde ele deve percorrer. Instanciados pelos agentes supervisores de cada nó, os agentes móveis realizam tarefas pré-determinadas movendo-se por todos os nós do sistema e retornam à sua origem após "efetivação" de suas "missões".

Após a carga do sistema, o agente supervisor inicia a coleta de informações dos objetos "ativos" no "host" onde está instanciado, as quais são transportadas pelos agentes móveis para os "hosts" remotos. Após um determinado período de tempo, cada agente supervisor tem conhecimento dos objetos instanciados nos "hosts" remotos.

2.3.2 - Serviço de Balanceamento

O Serviço de Balanceamento de carga é responsável pela análise dos dados coletados e armazenados pelos agentes. O objetivo principal deste serviço é distribuir o processamento entre os nós pré-definidos para o controle de satélites, tentando otimizar a utilização dos recursos computacionais disponíveis.

A partir de análises periódicas nos dados armazenados nas *tabelas de configurações,* o serviço dispara o processo de balanceamento de carga que consiste normalmente na replicação ou migração de um objeto entre os nós. As características de *flexibilidade* e *dinamismo* propostos na arquitetura são obtidos graças a este serviço. A flexibilidade advém da capacidade de se adequar ao número de usuários do sistema. Ou seja, se houver um aumento na demanda por solicitações de serviços, o Serviço de Balanceamento se responsabiliza por instanciar novos objetos no sistema que atendam a estas solicitações. Tal característica é explorada na *condição 4*, mostrada a seguir.

A característica de *dinamismo* corresponde a capacidade atribuída aos objetos de se moverem de um nó para outro, de acordo com a demanda por solicitação de serviços. Ou seja, se a demanda aumenta, o objeto da aplicação que atende esta

solicitação pode mover-se para o nó remoto solicitante. Esta característica é explorada na *condição 1*, mostrada a seguir. Várias situações poderão provocar o disparo dos serviços de balanceamento de carga, dentre elas pode-se mencionar:

a) **condição 1**: um nó está congestionado, ou seja, vários objetos foram criados e a taxa de CPU disponível neste nó tende a zero. Neste caso faz-se um levantamento entre os nós remotos para identificar o nó com maior número de conexões com o nó "sobrecarregado". Finalmente é instanciada uma cópia do objeto mais solicitado em o nó remoto;

b) **condição 2**: Se existem vários nós ociosos na rede, pode-se instanciar cópias de objetos de máquinas saturadas em máquinas ociosas;

c) **condição 3:** Uma necessidade operacional como por exemplo a desativação de um nó para manutenção. Neste caso se define uma intervenção do administrador de sistemas suspendendo todas as conexões dos objetos naquele nó. Em seguida cria-se tais objetos em máquinas remotas mais ociosas;

d) **condição 4:** A busca por disponibilidade de serviços: existem poucas instâncias de um determinado objeto "espalhadas" pelos nós da rede o qual recebe várias solicitações de serviços dos usuários. Neste caso justifica-se a cópia deste objeto do nó onde ele se encontra instanciado para o nó que contenha um maior número de solicitações de serviços com o mesmo;

e) **condição 5:** Queda de um nó: A queda de um nó será automaticamente detectada na primeira tentativa de comunicação de um agente com o respectivo nó. De acordo com a arquitetura proposta, o agente supervisor tem as informações de todos os objetos instanciados nos nós remotos. Desta forma, o nó mais próximo ao "host" com problemas se responsabiliza por instanciar os objetos deste nó em outros nós que estejam disponíveis. O estado deste nó passa de "ready" para "failed" e é atualizado em todas as tabelas de configurações do sistema. As conexões existentes com o nó que entrou no estado "failed" serão perdidas;

f) **condição 6**: Inclusão de um novo nó: A inclusão de um novo nó no sistema é detectada por agentes. Pela ação dos serviços de balanceamento de carga, objetos migrarão para este nó.

2.3.3 - Serviço de Persistência

O Serviço de Persistência disponibiliza operações de armazenamento, recuperação e exclusão no banco de dados. Os objetos persistentes da aplicação não precisam conter a implementação destas operações. Acionado por mensagens dos objetos, os serviços de persistência definem onde esses serão armazenados.

Como a arquitetura proposta é dinâmica, os objetos persistentes nem sempre poderão estar instanciados em o nó onde eles devem ser armazenados. Desta forma o objetivo do serviço de Persistência é tornar transparente o acesso e a localização do banco de dados e armazenar o objeto no nó que tenha a maior taxa de I/O disponível.

2.3.4 - Serviço de segurança

O Serviço de segurança é responsável pela garantia de acesso ao sistema somente a pessoas previamente autorizadas e de que os usuários somente tenham acesso às funções previamente definidas de acordo com o seu perfil. Por exemplo, as funções disponíveis pelo sistema para um engenheiro de satélite são diferentes das funções disponíveis para um controlador de satélites.

Visto que a arquitetura proposta permite que objetos migrem de uma máquina para outra, mecanismos de segurança devem ser elaborados para garantir que somente objetos autorizados possam migrar para máquinas remotas

2.4 - A carga do Sistema

O sistema é carregado em um conjunto de nós pré-definido, ou seja, existe uma relação de nós onde os objetos pertencentes ao sistema podem ser instanciados. A rotina de carga do sistema começa por um nó que assume o papel de gerente, podendo este ser qualquer um dos nós disponíveis para a carga.

O nó gerente acessa uma base de dados que contém a relação dos outros nós pertencentes ao sistema, bem como suas respectivas rotinas de carga. As rotinas de carga dos nós remotos são disparadas em paralelo, veja Figura 2.3.

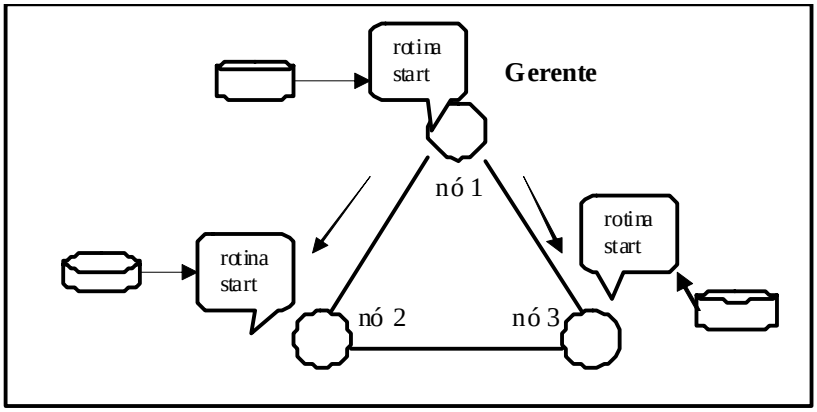

FIGURA 2.3: Carga do sistema proposto
FONTE: Ferreira (1999).

A rotina de carga em cada nó é responsável pelas seguintes tarefas: carga dos objetos da aplicação para aquele nó; carga dos serviços de persistência, segurança, balanceamento e agentes.

2.5 - Interface do usuário com o sistema

O agente supervisor apresenta-se como o responsável[2] pela interface entre a arquitetura proposta e o usuário. Após a identificação do usuário, o agente supervisor disponibiliza para o mesmo um conjunto de funções de acordo com o seu perfil (por exemplo, o controlador de satélites tem um conjunto de funções diferentes de um engenheiro de satélites).

A solicitação de um serviço do usuário é analisada pelo agente supervisor e o processo de atendimento segue os seguintes critérios: 1) Verificar se existem, em o nó local, objetos que atendam a solicitação desejada; 2) Caso existam, a conexão é estabelecida no próprio nó. Caso contrário, faz-se uma busca na *tabela de objetos* recuperando-se aqueles que atendam o serviço solicitado. 3) Em seguida, envia-se um "ping" para estes objetos. 4) A conexão é estabelecida com o objeto que responde ao

[2] situação não satisfeita na atual implementação do protótipo

"ping" e esteja instanciado em o nó que apresente a maior taxa de CPU disponível (esta taxa se encontra na *tabela de nós*)

2.6 - Funcionamento do sistema

Como já mencionado, a arquitetura proposta para controle de satélites é implementada em um domínio de rede pré-definido. Cada um dos nós pertencentes a este domínio contém uma rotina de carga responsável por instanciar os objetos residentes neste nó. A primeira ação realizada pela rotina de carga de cada nó é instanciar os objetos responsáveis pelos serviços de Persistência, Agentes, Segurança e Balanceamento. Após a carga destes serviços instancia-se os objetos da aplicação para controle de satélites. A dinâmica de interação em cada nó entre estes serviços e os objetos da aplicação podem ser mostradas na Figura 2.4 e comentadas a seguir:

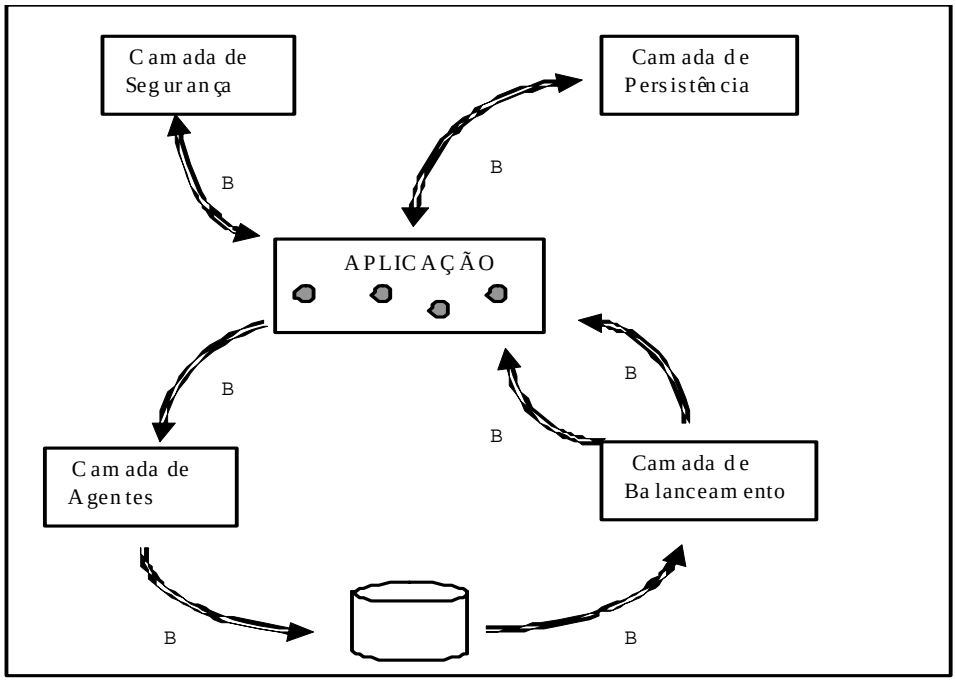

FIGURA 2.4: A dinâmica de interação em cada nó entre as Camadas de serviços e os objetos
FONTE: Ferreira (1999).

2.7 - Comentários

Após a carga dos objetos da aplicação, o "serviço de agentes" inicia a monitoração dos objetos instanciados. As informações são armazenadas no banco de dados para utilização posterior pela "camada de balanceamento". Dentre os vários parâmetros monitorados, pode-se citar: o estado de um objeto, podendo ser "ready" se o

objeto puder receber solicitações de serviços do sistema ou "failed", caso contrário; a carga da CPU de um determinado nó; o número de solicitações de serviços recebidos pelo objeto, etc..

A "camada de balanceamento" analisa em intervalo pré-definido e configurável os dados armazenados pelo serviço de agentes. A análise consiste basicamente em levantar os pontos críticos do ambiente configurado para controle de satélites, ou seja, identificar os nós que estão com uma sobrecarga de trabalho. Dois motivos podem ser a causa deste problema: primeiro, existe um número elevado de objetos instanciados naquele nó; segundo, os objetos instanciados naquele nó estão recebendo várias solicitações de serviços de outros nós da rede. A "camada de balanceamento" resolve o primeiro problema migrando objetos de máquinas saturadas para máquinas ociosas. O segundo é resolvido através da replicação dos objetos de um nó saturado para nós ociosos. Todo processo de migração é analisado pela "camada de segurança" para garantir que somente objetos autorizados possam migrar para máquinas remotas.

A "camada de persistência" tem como objetivo armazenar (recuperar) os objetos persistentes da aplicação. Os métodos de armazenamento dos objetos persistentes não acessam diretamente o sistema de armazenamento de dados visto que acionam, através de mensagens, a "camada de persistência" para tais operações.

2.8 – Motivação

De modo geral e amplo, define-se a proposta SICSD como o fato gerador da motivação da presente dissertação. Ou seja, a partir das propostas e considerações idealizadas para o ambiente dinâmico e flexível introduzido anteriormente, serão determinados o limite de abrangência e os objetivos deste trabalho.

Especificamente, o interesse despertado se concentra nos domínios do Serviço de Agentes mas pela constatação indelével de que este mesmo "módulo" participa como integrante parcial de uma estrutura maior e interdependente, justifica-se como necessários e sem exageros a consideração e o nível de detalhamento dos tópicos delineados nas seções anteriores. Logo, define-se o escopo estabelecido pela seção 2.3.1

como o limite de abrangência do trabalho. Os objetivos do trabalho são traçados na próxima seção.

2.9 - Objetivos

Os objetivos do atual trabalho compreendem os itens delineados abaixo:

a) Modelar um protótipo baseado na tecnologia de agentes móveis, para o Serviço de Agentes idealizado pela proposta SICSD.
b) O protótipo para o Serviço de Agentes visa considerar prioritariamente a integração de um ambiente baseado em agentes com a arquitetura baseada em objetos distribuídos (SICSD) e compatível à tecnologia "middleware" segundo o padrão CORBA.
c) Considerar a avaliação de plataformas de agentes móveis atualmente pesquisadas e desenvolvidas, orientando-se através de requisitos necessários para o cumprimento do "**item a**".
d) Compilar hipóteses através das quais se possa avaliar e/ou prever as propriedades, a aplicação, impactos e resultados do protótipo proposto no "**item a**".
e) O modelo do protótipo deve ser prático. Não prático no sentido de que supõe a implementação de um produto "manufaturado" ("acabado") para o mercado, mas que os princípios básicos e a confiabilidade do protótipo são aplicáveis em problemas do mundo real, em particular, na proposta SICSD. Enquanto fornecedor de uma experiência de aprendizado estimulante seja o objetivo, não interessa a exploração de tópicos puramente acadêmicos ou, mais especificamente, "esotéricos".

CAPÍTULO 3

TECNOLOGIA DE AGENTES

Agentes se tornaram um interessante tópico de pesquisa e estudo, pois figuram em diversas disciplinas da Ciência da Computação, incluindo arquiteturas orientadas a objeto e arquiteturas de objetos distribuídos, engenharia de software, sistemas de aprendizado adaptativos, inteligência artificial, sistemas especialistas, algoritmos genéticos, processamento distribuído, algoritmos distribuídos e segurança, para apenas enumerar alguns campos (Todd 1998)

O termo agente, entretanto, assim como objeto, não possui uma definição unanimemente aceita, em parte devido à multiplicidade de enfoques sob os quais é estudado. Além disso a área é criticada dentro da própria ciência da computação, onde algumas correntes nem mesmo consideram os agentes como uma nova tecnologia, alegando que qualquer coisa que possa ser feita com agentes "(...) pode também ser feita em C".

Os principais pontos que tornam os agentes alvos de críticas são (Her 1996):

a) Tudo com o rótulo ***agente*** vende (isto também seria verdadeiro na pesquisa), ou seja, com a mesma intensidade de adjetivos como ***plus***, ***turbo*** e ***super***, o termo ***agente*** desperta curiosidade soando atrativos peculiares das inovações tecnológicas;
b) Frente à constatação de que na maioria dos casos os agentes de software disponíveis no mercado apresentam arquiteturas as quais não acumulam características como complexidade e sofisticação, pergunta-se o que qualificaria os agentes como "blocos de código ***inteligentes".***

Os pesquisadores da área, por sua vez, refutam tais críticas com argumentos como os seguintes:

a) Os agentes tornam possível superar as diferenças entre os diversos tipos de redes (WAN, LAN e Internet) e plataformas, proporcionando maior comodidade no desenvolvimento de aplicações de grande escala;
b) As características de distribuição presentes cada vez mais nos ambientes computacionais podem ser muito mais facilmente tratadas por meio de agentes.

Estes prós e contras quanto à nova tecnologia não representam evidentemente a lista completa, e devem ser tomados como ilustração. O que se quer mostrar é a necessidade de uma definição tão clara e precisa quanto possível para "agente".

3.1 - O Conceito de Agente – Uma Definição?

Durante um debate ou uma simples conversa entre pesquisadores, levantam-se questões a respeito de agentes autônomos. A partir do momento em que uma breve explanação sobre o assunto deixa divagações duvidosas "no ar" como "*mas agentes se parecem justamente com um programa... como diferenciá-los?*", torna-se evidente a necessidade de uma explicação mais satisfatória. Logo, de imediato, busca-se defender a noção de um agente despida do perfil único e singelo de um programa, propondo-se uma nova conceituação.

Portanto, o que é um agente? Relaciona-se, abaixo, várias definições experimentalmente vinculadas a pesquisadores empenhados no estudo e desenvolvimento de agentes. O exame e comparação de algumas destas definições orientam a elaboração do novo conceito (Franklin 1996).

O Agente Mubot [http://www.crystaliz.com/logicware/mubot.html] - "O termo agente é utilizado para representar dois conceitos ortogonais. O primeiro é a habilidade do agente para uma execução autônoma. O segundo é a aptidão para raciocinar orientado a um domínio". A execução autônoma está centralizada na agência.

O Agente AIMA – Artificial Intelligence: a Modern Aproach – "Um agente é qualquer coisa que perceba seu ambiente através de sensors (sensores), e aja sobre este ambiente através de effectors (efetivadores)" . Esta definição depende crucialmente do que consideramos como um ambiente e do que significam os termos "sentir" e "agir". Desejando chegar a um contraste útil entre agente e programa, deve-se restringir, pelo menos, algumas noções referentes a esta dependência.

O Agente KidSim - "Vamos definir um agente como uma entidade-software persistente dedicada a um propósito específico." 'Persistência' distingue agentes de sub-rotinas; agentes têm maneiras peculiares para cumprir suas tarefas e agendas. 'Propósito específico' distingue-os dos aplicativos multifuncionais; agentes são tipicamente menores.

O Agente Hayes-Roth - "Agentes Inteligentes realizam continuamente três funções: percepção de condições dinâmicas no ambiente; ação para afetar condições no ambiente; e raciocínio para interpretar percepções, solucionar problemas, redigir inferências e determinar ações." Surge o raciocínio durante o processo de selecionar ações.

O Agente IBM [http://activist.glp.ibm.com:81/WhitePaper/pct2.html] - "Agentes inteligentes são entidades-software que manipulam conjuntos de operações em benefício de um usuário ou de outro programa com um certo grau de independência ou autonomia, e de alguma forma, empregar conhecimento ou representação das metas e anseios do usuário.". Esta definição vislumbra um agente inteligente atuando por concessão de um indivíduo ou de um outro agente.

O Agente Wooldridge&shyp; Jennings - "... um sistema, baseado em hardware ou software, que satisfaça as seguintes propriedades:

autonomia: agentes operam sem a intervenção direta de humanos ou outros, e têm algum tipo de controle sobre suas ações e estado interno;

aptidão social: agentes interagem com outros agentes (e, possivelmente, humanos) através de alguma linguagem de comunicação entre agentes;

reatividade: agentes percebem seu ambiente, (que pode ser o mundo físico, um usuário via interface gráfica, uma coleção de outros agentes, a Internet, ou talvez todos estes combinados) e respondem, em tempo hábil, às mudanças que ocorrem neste ambiente;

pro-atividade: agentes não respondem, simplesmente, ao ambiente. Tomam a iniciativa exibindo comportamento dirigido a um objetivo."

O Agente SodaBot [http://www.ai.mit.edu/people/sodabot/slideshow/total/P001.html] - "software-agentes são programas que se engajam em diálogos, negociam e coordenam transferência de informação." Aparentemente não encontramos intersecção entre esta definição e as anteriores, exceto pelo fato de que a negociação requer por exemplo, sensibilidade e ação, e diálogo exige comunicação.

O Agente Brustoloni - "Agentes autônomos são sistemas capazes de agir com autonomia e propósito no mundo real". Esta definição exclui software-agentes e programas em geral.

Todas as definições, não constituindo consenso a respeito de o que é um agente e como agentes se diferenciam de programas, provêem uma lista de atributos freqüentemente encontrados nos agentes.

3.2 - A Essência de um Agente

Normalmente evita-se prescrever argumentos a respeito de como uma palavra deveria ser utilizada. Russell e Norvig, a este respeito, colocam: "A noção de um agente converge para uma ferramenta de análise de sistemas, não uma caracterização absoluta que divida o mundo em agentes e não-agentes." (Russell 1995). Visto que agentes "vivem" no mundo real e os conceitos deste mundo produzem categorias desfocadas, tenta-se capturar a essência de ser um agente, e definir a ampla classe de agentes.

As definições da seção anterior parecem derivar de uma ou duas utilizações comuns da palavra agente. ***- alguém que age ou pode agir*** e ***- alguém que age no lugar de outro com permissão*** (ao se considerar este uso, pressupõe-se incluir o item anterior)

Pode-se tomar como exemplos de agentes, baseados no item 1, os humanos , alguns robôs móveis e autônomos vivendo no mundo real, os 'agentes-software' residindo em sistemas operacionais, base de dados, redes de computadores, e , finalmente, os agentes artificiais numa tela ou memória de computador. O que estes agentes compartilham que constitua a essência de ser um agente?

Destaca-se que cada um deles faz parte de e/ou se situa em algum ambiente. Sentem este ambiente e agem, com autonomia, sobre o mesmo. Atuam de modo que suas ações atuais possam afetar seus sensores (sentidos) e seu ambiente, de maneira contínua durante algum período de tempo. Tais requisitos constituem a essência de ser agente e podem ser formalizado como a seguir:

"Um agente autônomo é um sistema situado dentro e como parte de um ambiente, sente e age sobre este, ao longo do tempo, em busca de cumprir sua própria agenda e de maneira que afete o próprio sentido no futuro." (Franklin 1996)

Pode-se observar os casos extremos para exemplificar os limites desta definição. Os humanos no topo superior com múltiplos impulsos, sentidos e possibilidades de ação, estruturas de controle complexas e sofisticadas. No patamar inferior, um gerenciador controlando um simples termostato com um ou mais sensores. Os agentes autônomos "habitam" em um ambiente qualquer. Mude este ambiente e, certamente, desqualifique ou impossibilite um agente. Um robô que possua somente sensores visuais, em um ambiente sem iluminação, não é um agente.

Visto que a maioria dos programas convencionais desconsidera algumas condições previstas na definição acima, conclui-se que todos 'agentes-software' são programas, mas todos os programas não são agentes. Tais programas "normais" devem alcançar várias "marcas", "etapas" ou pré-requisitos que lhes caracterizem como um agente (ver seção 3.5). Os "agentes-software" não são definidos por suas tarefas. Um corretor ortográfico "inerte", ferramenta complementar à disposição do usuário, não é, por definição, um agente. Mas o mesmo corretor, rastreando e corrigindo simultaneamente a digitação, poderia ser considerado como tal.

3.3 – A Essência da Inteligência

A dimensão da inteligência corresponde ao grau de utilização absorvido por uma aplicação em relação ao raciocínio, aprendizado e outras técnicas correlatas à interpretação da informação ou do conhecimento aos quais tem acesso (Caglayan).

Tal dimensão compreende um número de passos os quais progridem a partir da inteligência marginal em direção às mais avançadas formas de inteligência. As mais limitadas formas de inteligência representam avanços modestos sobre atributos de customização das aplicações atuais, permitindo que o usuário especifique um "estilo", um "padrão" ou uma política.

O primeiro passo neste caminho conduz à expressão de *preferências*: instruções relativamente formais do interesse / vontade / desejo e dos potencialmente complexos comportamentos de um única aplicação ou de um grupo de aplicações.

O segundo passo conduz à provisão da capacidade de *raciocínio* na qual as preferências são expressadas através de um conjunto formalizado de regras e combinadas com conhecimento "s*hort-term"* e "*long-term"* em processos de inferência e de tomada de decisão. Tal cenário resulta em alguma ação executada pela aplicação ou, no mínimo, à criação de uma nova porção de conhecimento.

O terceiro passo circunda os limites da habilidade de uma aplicação-agente modificar seu comportamento de raciocínio fundamentada no novo conhecimento derivado a partir de uma vasta gama de fontes. Tal habilidade é geralmente definida como *aprendizado*.

3.4 – Modelo de definição baseado em taxonomia tri-dimensional

Tal modelo[1] propõe limites qualitativos para um agente inteligente. Tal definição de um agente simplesmente requer que a aplicação satisfaça as capacidades mínimas limitadas pela dimensões de agência, inteligência e mobilidade[2].

O software-aplicativo cujas habilidades posicionem o mesmo na área sombreada da figura abaixo "veste" a definição baseada em tecnologia de um "agente inteligente". O software-aplicativo situado fora da área sombreada pode satisfazer alguns aspectos de um agente, mas não possui a funcionalidade ou inteligência que o qualifique como um

[1] Modelo proposto por Donald Gilbert, IBM.
[2] A mobilidade e a Tecnologia de Agentes Móveis será detalhada posteriormente

"verdadeiro" agente. O modelo para a taxonomia tri-dimensional associa, na Figura 3.1, **agência** ao grau de autonomia e autoridade concedido ao agente, **inteligência** ao grau de comportamento, raciocínio e aprendizado exibido pelo agente e **mobilidade** ao grau de movimento do agente.

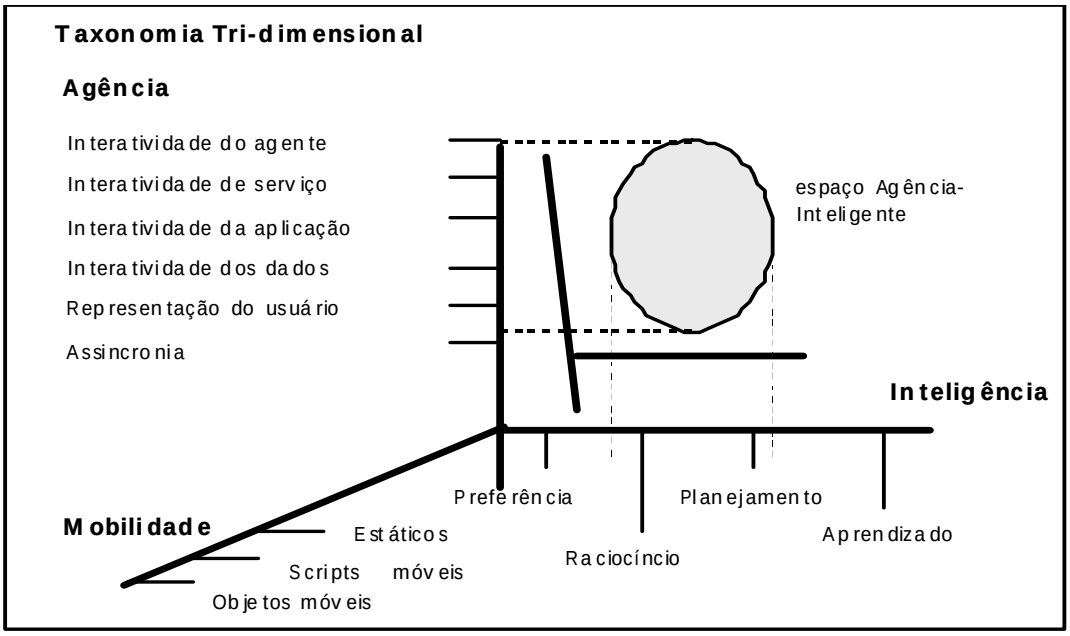

FIGURA 3.1: Taxonomia tri-dimensional para agentes
FONTE: Gilbert (1997) .

3.5 - Classificação de Agentes

A tabela abaixo apresenta uma possível classificação através das propriedades atribuídas a um agente:

TABELA 3.1: Classificação, por propriedades, dos agentes.

Propriedades	Outros Nomes	Significado
reativo	sensível e atuante	responde, em tempo hábil, a mudanças no ambiente
autônomo		exercita controle sobre suas ações
orientado à metas	atividade a um propósito	não age simplesmente em resposta ao ambiente
tempo contínuo		processo de execução contínua
comunicativo	hábil socialmente	comunica-se com outros agentes, inclusive, talvez, com pessoas
aprendizado	adaptativo	altera o comportamento frente à experiência prévia
mobilidade		apto para se locomover de uma máquina para outra
flexível		ações não seguem um "roteiro"
caráter		"personalidade" e estado emocional confiáveis

FONTE: Franklin (1996).

Os agentes podem então ser classificados de acordo com o subconjunto das propriedades que satisfazem, ou seja, de acordo com as tarefas que cumprem, pelo alcance e sensibilidade de seus sensores, pelo alcance e efetividade de suas ações, pela quantidade de estado interno que possuem ou ainda pelo ambiente no qual o agente atua e "vive". Representa-se na Figura 3.2 uma taxonomia conduzindo a agentes de monitoração e informação.

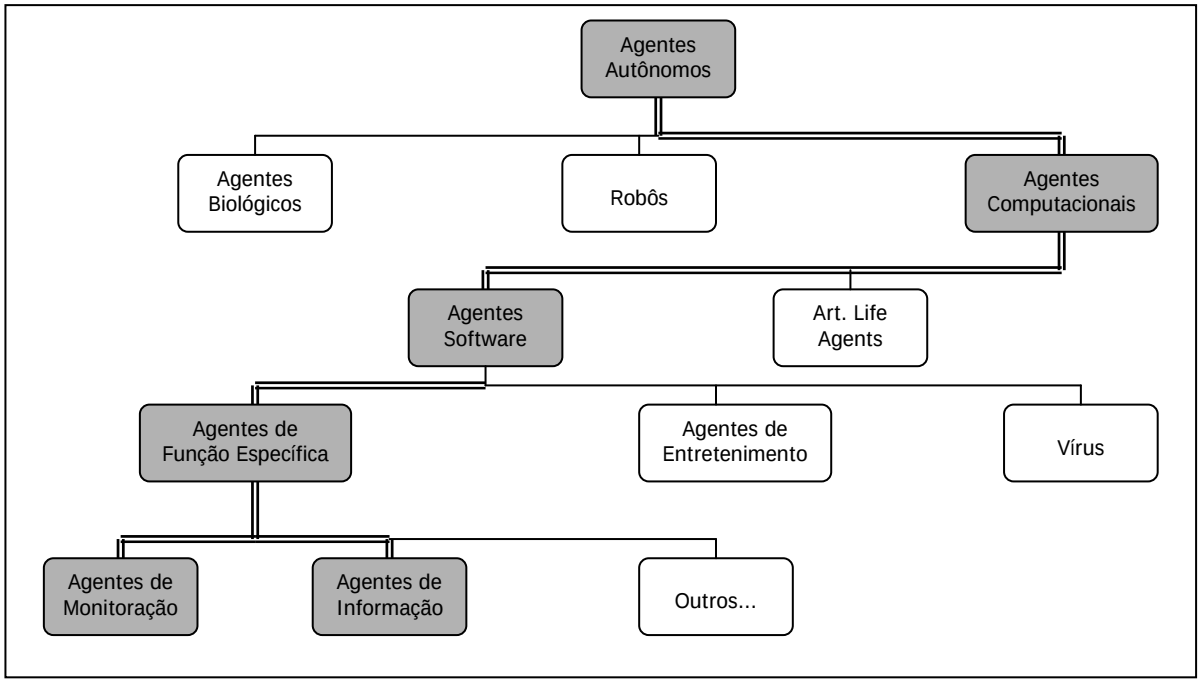

FIGURA 3.2: Agentes de Monitoração e Agentes de Informação
FONTE: Franklin (1996).

3.6. Tendências Atuais para a Utilização de Agentes

A indústria de agentes ainda se apresenta em um estado "embrionário". Assim, a abrangência da tecnologia e dos sistemas baseados em agentes ainda se encontra em crescimento. Atualmente, diversas classes de agentes têm sido desenvolvidas, entre as quais se destacam aquelas evidenciadas no "Green Paper" (OMG 2000) produzido pelo "Agent Working Group – OMG" sobre a Tecnologia de Agentes.

1) Agentes para Gerenciamento de Sistemas e Redes: As companhias de telecomunicações lideram o ranking dos setores mais ativos na área e, sem sombra alguma de dúvidas, surgiram como grupo mais comprometido com o paradigma de agentes. O objetivo principal destas companhias visa a utilização dos agentes para assistência às complexas tarefas de gerencimento de seus

sistemas e rede, tais como "balanceamento de carga", mecanismos de antecipação às falhas, análises de problemas e síntese de informações. Um alerta merece atenção: existe uma sobreposição confusa de termos nesta área. O gerenciamento de redes é, frequentemente, executado empregando-se agentes SNMP, os quais devem ser considerados como um mecanismo particular para gerenciamento. Evitando-se, portanto, a assimilação errônea de conceitos, ressalta-se que os tipos de agentes definidos neste tópico não se resumem a agentes SNMP.

2) Agentes de Suporte à Logística e Tomada de Decisões: Principalmente empregados em ambientes "exclusivos", os agentes são utilizados por companhias de serviços públicos e organizações militares para síntese de informações e suporte à tomada de decisões. Configurados a partir de estruturas tradicionais da comunidade de Inteligência Artificial, tais sistemas podem alertar um operador sobre um possível problema e prover informações vitais para o suporte à complexas decisões.

3) Agentes "Interest matching": Provavelmente trata-se da classe de agentes mais utilizada e, por "trabalharem às ocultas", estes agentes atuam sem o conhecimento da maior parte dos usuários que se utilizam dos seus serviços. Os agentes de "combinação de interesses" (interest matching) são adotados por "sites Web" comerciais para oferecer recomendações. Por exemplo, se determinado internauta aprecia obras como *"Frank Sinatra's Greatest Hits"*, este mesmo consumidor em potencial poderia desejar um trabalho musical como *"Tony Bennett's Songs for a Summer Day"*. Baseados em uma das pesquisas de Patti Maes (MIT Media Labs), e posteriormente no produto Firefly, estes agentes observam os padrões de interesse que podem ser utilizados na elaboração de "sugestões e/ou recomendações". Destaca-se a presença de tais agentes no "site" Amazom.com e em diversos outros "sites" de áudio e vídeo.

4) Agentes Assistentes para Usuário: Estes agentes, às vezes apresentados como "conselheiros-cartoon", operam ao nível de interface de usuário (User Interface), proporcionando informações ou avisos (conselhos) para os usuários. Companhias como Microsoft, Lotus, e Apple têm mostrado grande interesse nesta área. Como exemplo mais conhecido e publicamente difundido, na forma de figuras animadas de "help" dos produtos Microsoft Office, estes agentes utilizam "bayesean networks" para análise e predição de tópicos sobre os quais os usuários possam solicitar auxílio.

5) Agentes de Estrutura Organizacional: Estes agentes são estruturas que operam de maneira similar às organizações humanas. Por exemplo, sistemas multi-agentes para "supply chain"[3] compreenderiam agentes representando funções tais como: compradores, fornecedores, "brokers", estoque, "orders", itens de linha e células de manufatura. Sistemas de Operações teriam agentes-recursos, agentes-material, agentes-processos e assim por diante. Cada um destes exemplos representam alguns

[3] Rede de Fornecimento

aspectos das características de agentes. Entretanto, nenhum deles retrata a abrangência total dos possíveis atributos dos agentes.

3.7 - Tipos de Aplicações de Agentes

Os tipos de aplicações que empregam agentes ainda são limitados. À medida que os conceitos sejam progressivamente assimilados e que se perceba o surgimento de novas ferramentas, a abordagem baseada em agentes será naturalmente adotada nas aplicações da Tecnologia da Informação. Os principais destaques verificados (OMG 2000) no âmbito de aplicações da Tecnologia de Agentes podem ser agrupados como segue abaixo.

a) Aplicações Corporativas: "Smart documents" (isto é, documentos que possuem "conhecimento" sobre aspectos do próprio processamento); operações orientadas a metas (isto é, fluxo de trabalho "on steroids"); gerenciamento do rol de funcionários e de funções (isto é, ligação dinâmica de cargos e habilidades às pessoas).

b) Controle de Processos: Edifícios Inteligentes (por exemplo, aquecimento / refrigeração e segurança inteligentes - "smart"); gerenciamento de aparelhagem (por exemplo, em refinarias); robôs.

c) Agentes Pessoais (PDA – Personal Digital Agents): "Filtros" de e-mail e notícias; gerenciador de Agenda Pessoal; secretárias automáticas pessoais.

d) Tarefas de Gerenciamento de Informação - 1**) Pesquisa / Busca de Informação:** A quantidade gigantesca de informação disponível nas "entranhas" de uma intranet corporativa impõe barreiras à capacidade da maior parte dos usuários quanto a recuperação efetiva de informação útil. Agentes de busca possuem conhecimento sobre diversas fontes de informação. Tal conhecimento abrange os tipos de informação disponíveis em cada domínio, métodos para acessar determinadas informações e noções sobre a segurança e precisão das fontes de informação. Estes agentes utilizam tal habilidade na execução de tarefas de busca específicas; 2**) Filtragem de Informação:** Outra tarefa comum para agentes. Agentes de "filtragem" lidam com um conhecido problema de "sobrecarga", limitando ou selecionando o montante exagerado de informações que chega para um determinado usuário. A idéia básica se resume em desenvolver um "substituto on-line" com inteligência suficiente sobre as necessidades do usuário quanto ao tipo de informação tendo, portanto, a capacidade de selecionar apenas documentos de interesse. Esta classe de agentes, em geral, funciona como "porteiros eletrônicos"

evitando uma “inundação” desnecessária de “lixo informativo”. Os agentes de “filtragem” também atuam em cooperação com, ou às vezes incorporados em, agentes de busca a fim de manter a quantidade de resultados “baixada” em níveis razoáveis. Tipicamente apoiados por máquinas de aprendizado, estes agentes podem se adaptar às necessidades de cada usuário provendo mecanismos mais precisos quando comparados às abordagens de filtragem por “palavra-chave”; 3**) <u>Monitoramento de Informação:</u>** Muitas tarefas dependem da notificação “imediata” ou em tempo hábil sobre alterações em diferentes fontes de dados. Um operador responsável pelo planejamento de logística (planner), por exemplo, elabora um plano de tranporte para um equipamento de uma certa localização para outra, mas o andamento deste plano pode ser atrapalhado em virtude de uma abrupta alteração climática (uma tempestade) durante uma parada para reabastecimento. O “planner” gostaria de ter conhecimento, o mais rápido possível, a respeito de qualquer evento que, como o imprevisto citado, pudesse afetar seu plano. Os agentes são úteis no monitoramento de fontes de dados distribuídas e, como entidades-software, desenvolvem um comportamento “paciente e repetitivo” para constantemente monitorar as alterações em fontes de dados. Alternativamente, agentes móveis podem ser despachados para localizações remotas e/ou inascessíveis inspecionando fontes de dados às quais um usuário, normalmente, não tem acesso; 4**) <u>Mediação em Fontes de Dados:</u>** O ambiente de gerenciamento de dados é “povoado” por múltiplos sistemas diferentes que, em grande maioria, não interagem. Os agentes podem ser utilizados como “mediadores” entre as diversas fontes de dados, fornecendo mecanismos que permitam a interoperabilidade. O projeto SIMS (**ISI)** desenvolveu um “mediador” de informações que provê acesso a bases heterogêneas de conhecimento e de dados. Estes agentes utilizam uma linguagem de alto nível, protocolos de comunicação e ontologias específicas a um domínio para a descrição dos dados contidos em suas fontes de informação. Tudo isto permite que cada agente se comunique com outro em um nível semântico avançado; 5**) <u>Agentes-Interface / Assistentes Pessoais:</u>** Um agente-interface é um programa com habilidade para operar dentro de uma interface de usuário e para prestar assistência ativa ao usuário quanto a operação da interface e e manipulação do sistema base. Este tipo de agente pode interceptar dados “input” do usuário, examiná-los e tomar uma “atitude” conveniente. Enquanto não se relacionam diretamente com o gerenciamento de dados, os agentes-interface devem apresentar potencialidades para desempenhar diversas funções de assistência aos usuários dos sistemas de gerenciamento. Tal fato é de suma importância visto que os sistemas de gerenciamento acompanham as tendências de distribuição e agrupamento em enormes e complexos “sistemas de sistemas”. Os agentes atuando na interface pode funcionar como “ponte” entre o domínio de conhecimento dos sistemas de gerenciamento e os usuários. Estes agentes auxiliariam os usuários, entre outras tarefas, na formação de consultas, localização dos dados e “interpretar” a semântica dos dados. Como exemplos, cita-se Sistemas Tutorias Inteligentes e Assistentes de Navegação Web (Lieb 1995). Em tempo, pode se verificar o emprego de agentes-interface nos produtos “desktop” da Microsoft que “observam” as ações dos usuários e elaboram sugestões apropriadas.

3.8 - Armadilhas no Desenvolvimento Orientado a Agentes

Enquanto os fundamentos teóricos e experimentais dos sistemas baseados em agentes passam por uma assimilação global, pouco esforço, comparativamente, tem sido devotado na compreensão da pragmática do desenvolvimento de sistemas multi-agentes – a realidade crua e árdua na efetivação de um projeto baseado na tecnologia de agentes (Woo/Jen 1998). Não é interessante, de um modo geral, a repetição de "erros" que resultem, numa perspectiva otimista, em desperdício de recursos e, no outro extremo, acarretem na derrocada de todo um projeto.

Pode-se enumerar, com provas irrefutáveis, as soluções, aplicações, resultados teóricos e experimentais alcançados no avanço incremental da tecnologia de agentes. Por outro lado, desconsidera-se os aspectos pragmáticos do desenvolvimento de sistemas de agentes, através dos quais se garante uma exploração completa e eficaz do enorme potencial da tecnologia. Para tanto, deve-se reconhecer que os sistemas baseados em agentes apresentam problemas e "armadilhas" específicas os quais, uma vez identificados, não assegurariam o sucesso, mas orientariam como prevenir e evitar algumas das mais óbvias fontes de falhas no desenvolvimento de projetos (Webster 1995)

As constatações de metodologias errôneas de desenvolvimento de sistemas de agentes fundamentam-se em uma década de experiências que variam de sistemas de controle industrial (Jennings 1995) a sistemas de gerenciamento e recuperação de informação (Woold 1997).

A análise destas "armadilhas" adotou a estrutura fundamentada no livro "*Pitfall of Object-Oriented Development*" (Webster 1995) a partir da qual se identifica 7 categorias. Para cada uma destas categorias - ***Política***, ***Gerenciamento***, ***Conceitual***, ***Análise e Projeto***, ***Micro-Nível*** (agente), ***Macro-Nível*** (sociedade de agentes) e ***Tópicos de Implementação***, discute-se a natureza da "armadilha", como pode ser evitada e, quando possível, quais passos devem ser adotados no tratamento e /ou na recuperação.

3.8.1 - Categoria Política

a) **Supervalorização dos agentes:** Os agentes não perfazem um paradigma "mágico" de solução de problemas: tarefas que estavam além do escopo da automação sem as técnicas de agentes não necessariamente serão solucionadas simplesmente adotando-se a abordagem baseada em agentes. Os problemas que há décadas desafiam os engenheiros de software ainda permanecem com os sistemas de agentes. Na verdade, não existem evidências de que qualquer sistema que utiliza a tecnologia de agentes não possa ser construído com técnicas não "agent-based". Em resumo, os agentes podem facilitar a solução de uma certa classe de problemas (e há bons argumentos para supor tal possibilidade), mas não "torna possível o impossível". Outro aspecto surge quando se supões que os agentes são capazes de agir e raciocinar como humanos. Obviamente não é o caso: tal nível de competência está muito além do "estado da arte" da Inteligência Artificial. Logo, os agentes podem, às vezes, exibir um comportamento inteligente na solução de problemas, mas ainda muito limitado pela situação atual da inteligência de máquina.

b) **Agentes - entre dogmas e religião:** Os agentes não são uma solução universal, embora utilizados em uma vasta gama de aplicações (Woold 1998). Existem aplicações para as quais os paradigmas mais convencionais (tal como OOP) são mais apropriados. Na verdade, frente à imaturidade relativa da tecnologia e o número de aplicações, é recomendável que se certifique a respeito das vantagens da abordagem por agentes antes de contemplá-la como única. Se um problema pode ser resolvido, com resultados similares, por duas tecnologias, deve-se priorizar aquela mais compreendida pelos engenheiros de software. Muitos pesquisadores e desenvolvedores exibem a síndrome da "técnica única" – Se a única ferramenta que você possui é um martelo, então TUDO se parece com um prego. Resumo: Há perigo em acreditar que os agentes são a solução adequada para todos problemas. A outra forma de dogma se relaciona com sua definição: 1) Não existe consenso na definição => Coleção de definições (Franklin 1996); 2) Busca-se encaixar a solução individual para combinar com a definição. Por exemplo, aqueles que consideram a mobilidade como uma característica essencial, invariavelmente propõem soluções com agentes móveis mesmo que agentes estáticos representem uma abordagem óbvia e natural.

3.8.2 - Categoria Gerenciamento

a) **Não possuir razões definidas que justifiquem o uso de agentes:** Este é um problema comum à qualquer nova tecnologia sobre a qual se alardeiam prognósticos fantásticos. Os gerentes lêem previsões tais como a geração de 2.6 bilhões de dólares no ano 2000 e, sem surpresas, querem fazer parte da inovação. Resumindo, como tratar o vislumbre pela nova tecnologia? Entretanto, os gerentes que propõem um projeto baseado em agentes não têm realmente uma idéia formada sobre os retornos factíveis. Não têm uma

visão definida e clara de como os agentes podem inovar seus produtos existentes ou proporcionar a geração de novas linhas de produtos. Sem objetivos, não há critérios para alcançar o sucesso da iniciativa, nem mesmo ao menos prever se o projeto "caminha" ou sofre pela estagnação. A lição: realmente entender as razões que motivam um projeto de desenvolvimento de agentes e o que se espera ganhar.

b) Não se sabe em que áreas os "seus" agentes são especialistas: Está relacionada com o item **3.8.2.a** e abrange a falta total de clareza de propósito para o uso da tecnologia e carência de considerações sobre o grau de aplicabilidade. Lição: Estar convicto a respeito de COMO e ONDE a nova tecnologia pode ser adequadamente aplicada.

c) Pretensão em construir soluções genéricas para problemas "ONE-OFF" (solução única): Manifesta-se na tentativa de inventar uma arquitetura ou "testbed" que supostamente possibilite a construção de uma grande variedade de tipos potenciais de agentes, quando o que realmente interessa é um projeto na medida especial para o tratamento de uma aplicação singular. "Experts" em desenvolvimento orientado a objetos concordam que dificilmente se promove o reuso de código a menos que a implementação se dirija a uma pequena "faixa" (domínio) de problemas com características similares (Webster 1995). Além do mais, soluções amplas e gerais dificultam e aumentam os custos de desenvolvimento e freqüentemente exigem vários especialistas para lidar com diferentes aplicações. O que se dizer das arquiteturas genéricas? E os tempos áureos da Inteligência Artificial em que se proclamavam o desenvolvimento de "solvers" para propósito geral?

d) Confundir protótipos com sistemas: Após encontrar uma aplicação para a qual uma solução por agentes seja adequada e elaborar a solução num nível geral, o desenvolvimento de um protótipo consistindo de poucos agentes que interagem na conclusão de uma tarefa torna-se comparativamente fácil. Entretanto, tal cenário se distancia totalmente de uma solução robusta e confiável para o mundo real. Transpor a barreira erguida entre estas duas fases distintas, nos sistemas de agentes, inclui lidar com características tais como: 1) solução de problemas concorrentes e distribuído; 2) interfaces sofisticadas e flexíveis para os objetos; 3) complexidade individual de objetos cujo comportamento depende do contexto. Cada uma destas "parcelas", em particular, dificulta a fase de evolução entre um protótipo e a versão final do software, e quando se unem (as parcelas), o intervalo pode se tornar um abismo.

3.8.3 - Categoria Conceitual

a) Acreditar que os agentes são a "Bala de Prata" (silver bullet): O Santo Graal da engenharia de software é a "Bala de Prata": uma técnica que trará uma ordem de magnitude no aperfeiçoamento, melhoras e enriquecimentos para o desenvolvimento de software (Brooks 1986). Várias tecnologias já assumiram papéis ou foram promovidas a "bala de prata": programação automática, sistemas especialistas, programação gráfica e métodos formais como alguns exemplos. Trata-se de uma questão de tempo para que a

tecnologia de agentes seja aclamada como tal? Seria uma perigosa falácia? Os fatos não verificam isto na prática. Ainda é cedo. O que se dizer da abstração procedural, programação estruturada, tipos abstratos de dados e objetos? Poderosas abstrações de programação desenvolvidas nas últimas três décadas? Do mesmo modo, os agentes são apenas uma abstração. Espera-se que, com o passar do tempo, a tecnologia comprove os benefícios para o desenvolvedor. Por enquanto, sugerir que tais benefícios são matéria de fato se caracterizaria como ingenuidade e precipitação.

b) Confusão entre "buzzwords" e conceitos: Visualiza-se a popularidade da tecnologia na idéia extremamente intuitiva de um agente. Se por um lado, o fato de que o conceito percorre através de distintas disciplinas atesta sua vasta aplicabilidade, pode-se, no outro extremo surgir desenvolvedores que corajosamente acreditam entender o conceito quando de fato isto não pode ser confirmado. Um exemplo pode ser o modelo de agência BDI (Belief / Desire / Intention) (Georgeff 1987), interessante para desenvolvedores pois está sustentada por uma respeitável teoria de agência (humana) (Bratman 1987), tem uma elegante semântica lógica e talvez mais importante, comprovada em aplicativos de demandas extremas tais como diagnóstico de falhas em tempo real da Space Shuttle (Georgeff 1996). Infelizmente, a sigla "BDI" tem sido aplicada a muitos tipos de agentes – muitos dos quais não são legítimos sistemas BDI, causando a perda de sentido. Um clara ilusão constatada. Os dizeres "Our System is a BDI system" traduzindo um sistema BDI como sendo um PC com 64 MB de memória.

c) Esquecer que se trata de desenvolvimento de SOFTWARE: Sem confiança e teste das técnicas disponíveis, o desenvolvimento se torna um processo essencialmente experimental levando-nos ao esquecimento o intento de se produzir software. Os processos de engenharia de software – análise de requisitos, especificação, projeto, verificação e teste, são esquecidos e/ou ignorados. Tal negligência nos remete à conclusão previsível: o projeto "se arrasta" e/ou encalha, não por problemas específicos dos agentes, mas em virtude do total desprezo aos bons preceitos da engenharia de software. Logo, na ausência de técnicas orientadas a agentes, as técnicas de orientação a objetos podem surtir efeitos plausíveis. Podem não ser ideais, mas certamente melhor que "nada".

d) Esquecer que se desenvolve software DISTRIBUÍDO: Reconhecida como uma das mais complexas classes de sistemas computacionais em termos de projeto e implementação, os sistemas distribuídos geram um grande esforço de pesquisa na busca de compreensão da complexidade e definição de formalismos e ferramentas que possibilitem o seu gerenciamento (Ben-Ari 90). Há muito o que se fazer. Na construção de uma plataforma ou sistema de agentes, é vital a consideração das lições vivenciadas pela comunidade da tecnologia de distribuição, quanto a considerações relativas à sincronização, exclusão mútua para recursos compartilhados, deadlock e livelock.

3.8.4 - Categoria Análise e Projeto

a) **As tecnologias Correlatas não são exploradas:** Quando se desenvolve qualquer sistemas de agente, a porcentagem do projeto que se relaciona especificamente à tecnologia de agente (por exemplo, cooperação, negociação ou aprendizado) é comparativamente pequena. Tal situação se adequa à visão de desenvolvimento de sistemas "raisin bread" (pão de uvas passas), atribuída à Winston em (Etzioni 1996), na qual as partes do sistema que se baseiam na tecnologia de agentes correspondem à pequena porcentagem de uvas passas e a parte que se relaciona às tecnologias padronizadas corresponde relativamente à maior parte de pão. É importante que as tecnologias e técnicas convencionais sejam exploradas sempre que possível. acelerando o processo de desenvolvimento, evitando a famosa "reinvenção da roda" para que se devote tempo suficiente para o "componente de valor", ou seja, o agente. Muitos projetos podem extrair benefícios das tecnologias disponíveis tais como: plataformas para computação distribuída (Samira 2000), sistemas de banco de dados para manipulação de requisitos de processamento de grandes quantidades de informação e sistemas especialistas para raciocínio e solução de problemas.

b) **O projeto não explora a concorrência:** O projeto do sistema é um fator determinante crucial de sucesso – um projeto "pobre" acarreta uma exploração deficiente da metáfora "agente" que por sua vez prejudica todo o sistema. Por exemplo: Um agente processa alguma informação, produz alguns resultados e entra em estado inativo. Os resultados são transferidos para um outro agente (até então inativo) que processa, produz e retorna ao estado inativo. Este ciclo provém de um projeto insatisfatório pois existe um único "thread" de controle e a concorrência – alardeada como um dos mais importantes potenciais de solução multi-agentes, não é explorada. Assim, um dos alvos primordiais das fases de análise e projeto é produzir sistemas que garantam uma porção razoável e apropriada de atividades concorrentes na solução de problemas.

3.8.5 - Categoria Micro-Nível (Agentes)

a) **Decisão de produzir a própria arquitetura:** As arquiteturas de agentes são camadas essenciais à construção de agentes (Woold 1995) e facilmente pode ser contabilizada, através dos anos, a proposta de diversos exemplares. Surge uma grande tentação quando, inicialmente, se assume um projeto de agentes. Supõe-se, então, que não existe uma arquitetura que satisfaça os requisitos específicos do problema e que é necessário engendrar uma nova arquitetura "a partir do zero". Alguns fatores, entre outros, contribuem para tal tentação: 1) tendência "not designed here": confiar apenas nos próprios produtos desenvolvidos; 2) desejo de gerar propriedade intelectual em busca de lucros ou reconhecimento acadêmico. O erro ao se enfatizar tais fatores pode trazer as seguintes consequências: 1) dispêndio de anos de esforço (não pessoas/ano, mas anos) para desenvolver um produto confiável e pronto para utilização; 2) a menos que o processo seja realizado em cooperação com uma pesquisa de maior âmbito, o fato de se tratar de uma

arquitetura inédita pode dificultar o interesse pela mesma e geração de rendimentos. Portanto, recomenda-se o estudo das diversas arquiteturas descritas na literatura (Woold 1995), a obtenção de licenças e/ou a complementação de projetos em "fase de produção".

b) Considerar que a própria arquitetura é genérica: A opção pelo desenvolvimento de uma arquitetura própria aplicada com sucesso a um determinado problema deve estar amarrada a argumentos que justificam tal aplicação e à restrição de que a mesma pode ser adaptada apenas a problemas com características similares.

c) Os agentes possuem muita inteligência: Deve se evitar a sobrecarga do sistema com técnicas de Inteligência Artificial em conseqüência à leitura de definições dentro das quais os agentes têm a habilidade de raciocinar e aprender (ou planejar ou comunicar-se em linguagem natural) imaginando que tais características são essenciais ao projeto em questão. Em geral, uma estratégia de sucesso advém da construção de agentes "aditivados" com o mínimo de técnicas de AI e à medida que tais sistemas entram em fase de operação básica, pode se aplicar progressivamente complementos que personifiquem avanços da arquitetura. Etzioni em (Etzioni 1996) denomina tal estratégia como "useful first".

d) Os agentes NÃO possuem inteligência: Numa extremidade oposta ao cenário apresentado pela seção anterior (3.8.5.c), estão os agentes, supostamente definidos como tais, que não implementam funcionalidade alguma que justifique a utilização do termo. Por exemplo, tornou-se comum encontrar sistemas distribuídos anunciados como sistemas multi-agentes. Outro erro comum é praticado quando se considera como "agente" qualquer página WWW com algum processamento "nos bastidores". Tais práticas atrapalham, pois: 1) o termo "agente" perde o sentido (significado) que deveria ter; 2) frusta as expectativas do usuário que recebe, na verdade, um fragmento convencional de software; 3) dissemina o descrédito por entre os desenvolvedores (passam a considerar o termo "agente" como simplesmente outro jargão sem sentido).

3.8.6 - Categoria Macro-Nível (Sociedade de Agentes)

a) Agentes por toda a parte: Quando se inicia o aprendizado sobre sistemas multi-agentes, existe a tendência de se enxergar tudo como um agente. Como exemplo de um caso clássico, o cálculo do fatorial de **n** requer a geração de **n** actors (menor unidade no paradigma de computação concorrente (Agha 1986). Da mesma forma em que uma linguagem é considerada puramente orientada a objeto se a mesma enxerga tudo como um objeto, tal paradigma propõe actor (similar a um agente) para tudo (inclusive, actors para adição e subtração). Não é difícil prever que, neste caso, considerar ingenuamente tudo como um agente será extremamente ineficiente pois o "overhead" associado ao gerenciamento de agentes e à comunicação entre agentes resultará num custo maior que os benefícios alcançados pela solução baseada em agentes. Em geral, os agentes deveriam ser metodicamente elaborados em blocos, cada qual incorporando funcionalidades computacionais coerentes e significantes.

b) **Presença de muitos agentes:** É bem conhecida proposição de que um número de sistemas interagindo uns com os outros através de regras simples pode gerar cenários e/ou situações consideravelmente mais complexos do que a soma das partes (Steels 1990). Nisto repousa a força - e fraqueza - dos sistemas multi-agentes. A força advém da funcionalidade emergente explorada pelo desenvolvedor em busca de comportamento cooperativo simples e robusto. A fraqueza surge visto que tal funcionalidade está intimamente ligada ao caos (Woo/Jen 1998). Em resumo, a dinâmica dos sistemas multi-agentes é complexa e pode ser caótica.

c) **Presença de poucos agentes:** Enquanto alguns projetistas idealizam um agente diferenciado para cada tarefa possível, outros não reconhecem o valor dos sistemas multi-agentes. Logo, criam um sistema multi-agentes que peca completamente na exploração do paradigma apresentando um número insuficiente de agentes responsáveis por todo o trabalho. Tais soluções, além de prejudicarem a concorrência concentrando toda funcionalidade numa única classe, tendem a falhar no teste de coerência padronizado pela engenharia de software, o qual exige que um módulo deve ter função única e coerente.

d) **Dispender tempo demasiado implementando infra-estrutura:** Um dos grandes obstáculos que barram a disseminação da tecnologia de agentes é a ausência de uma plataforma eleita para o uso amplo e comum no desenvolvimento de sistemas multi-agentes. Um tópico relacionado com tal situação merece atenção visto que a infra-estrutura é, em geral, implementada por especialistas em Inteligência Artificial inadequadamente assumindo o papel dos “experts” em redes de computadores e sistemas distribuídos.

e) **O sistema é anárquico:** Uma concepção errônea e comum defende que sistemas de agentes podem ser desenvolvidos apenas através do agrupamento de um número de agentes, que tais sistemas não requerem estruturação prévia e que todos os agentes são do mesmo tipo. Mesmo dependendo do problema em questão, algumas questões mais comuns devem ser adotadas para a estruturação de uma sociedade de agentes que trabalhem na busca de um propósito comum: 1) agrupar equipes de agentes para atingir metas comuns; 2) utilizar abstrações para modelar hierarquicamente o problema em diferentes perspectivas; 3) definir intermediários que atuarão como ponto de contato para um certo número de agentes.

f) **Confundir simulação com a realidade:** A grande maioria dos sistemas multi-agentes nasce como um protótipo, com todos os agentes “rodando” em um único computador, de forma que a distribuição é meramente simulada, ou seja, os agentes não estão distribuídos pela rede. Não se deve ignorar as vantagens da simulação – economia de recursos, principalmente, entretanto, existe a tendência de se assumir que os resultados obtidos na distribuição simulada podem ser equiparadas aos resultados de uma situação real. Discute-se, ao mesmo tempo, que o cerne do problema se apega ao fato de que na distribuição simulada existe a possibilidade do controle centralizado. Na realidade, nos sistemas distribuídos tal recurso deve ser desconsiderado a menos que se desista das vantagens obtidas na distribuição.

3.8.7 – Categoria Tópicos de Implementação

a) **A *"tabula rasa":*** Quando se constrói sistemas utilizando tecnologias emergentes, apregoa-se a hipótese de que: "todo componente do sistema deve ser projetado e concluído a partir do zero". Freqüentemente, entretanto, os mais importantes componentes de um sistema de software são legados, ou seja, funcionalmente essenciais mas tecnologicamente obsoletos, tais componentes não podem ser prontamente redefinidos. Portanto a proposta de uma nova solução em software deve essencialmente prever a manipulação destes componentes que não podem ser ignorados ou substituídos. Os "sistemas legados" podem ser incorporados a um sistema de agentes através do "empacotamento" (wrapping) dos mesmos numa camada de agente (Woo/Jen 1998). A idéia básica prescreve a possibilidade de cooperação e comunicação entre os agentes e os componentes legados através de uma camada-software na forma de uma API. Desta maneira, a funcionalidade do software legado pode ser estendida habilitando-os a trabalhar com outros componentes desenvolvidos (agentes).

b) **Ignorar os padrões *de facto*?:** Num campo novo como sistemas de agentes, existem poucos padrões estabelecidos que possam ser utilizados para o desenvolvimento de componentes baseados nesta nova tecnologia para uma aplicação específica. As iniciativas para a implantação de tais padrões (Woo/Jen 1998), atualmente, se apresentam em estágios preliminares. Um dos padrões *de facto* na área é a KQML (KQML 1993), uma linguagem para comunicação de agentes (ACL) adotada em muitos projetos de desenvolvimento.

3.9 - Visão Geral dos “Padrões” da Tecnologia de Agentes

O resumo, apresentado abaixo, se fundamenta no relatório do panorama geral dos “padrões” para a Tecnologia de Agentes produzido por Frank Manola (Manola 1998) e no “Green Paper” (OMG 2000) produzido pela Agent Working Group - OMG.

3.9.1 - OMG MASIF

A especificação OMG- MASIF (Mobile Agent System Interoperability Facility) se destina, em suma, a suportar a interoperabilidade dos sistemas de agentes heterogêneos. Os objetivos principais se referem à padronização das seguintes “áreas”:

a) Agent Management (Gerenciamento): por exemplo, criação, suspensão, reinício e destruição de agentes;
b) Agent Transfer (Transferência para implementações de sistemas de agentes similares): mecanismos para “receber” os agentes transferindo, em conjunto, suas classes;
c) Agent and Agent System Names;
d) Agent System Type and Location Syntax;
e) Agent Tracking: rastrear (“tracing”) os agentes registrados nos “naming services” dos diferentes sistemas de agentes.

A especificação não abrange as seguintes áreas:

a) Segurança “Multi-hop”;
b) Circulação dos agentes entre sistemas díspares (por exemplo, conversão de código);
c) Comunicação dos Agentes: a definição dos protocolos de comunicação e negociação através de uma ACL.

A especificação MASIF define um modelo de referência o qual estipula conceitos tais como:

a) Agent System (sistema de agente): uma plataforma que pode criar, interpretar, executar, transferir e destruir agentes.

b) Agent System Type (tipo de sistema): descreve o "fornecedor", a linguagem de implementação, e os mecanismos de serialização – por exemplo, "Aglet - ASDK" é um tipo de sistema de agente.

c) Place : um ambiente dentro do sistema de agente no qual um agente pode executar. Um determinado sistema de agente pode suportar múltiplos "places". Conceitualmente, uma localização.

d) Region: um conjunto de sistemas, não necessariamente do mesmo tipo, com a mesma autoridade ("authority").

A especificação MASIF define a interface MAFAgentSystem a fim de suportar as funções de "Agent Management", "Agent Transfer" e "Agent Tracking". A interface MAFFinder, também definida, provê métodos para a manutenção de uma base de dados (nomes e localização) dinâmica dos agentes, "places" e sistemas de agente. A especificação examina seu relacionamento com os serviços CORBA (Naming, Lifecycle, Externalization e Security). Trabalhos complementares (para futuros "RFPs"[4]) têm sido, também, identificados como por exemplo: a) interoperabilidade (em alto nível) das plataformas baseadas em Java (formatos de serialização e habilidades internas dos sistemas de agente); b) interfaces IDL adicionais (por exemplo, para comunicação, em baixo nível, dos agentes; c) cooperação com o "padrão" FIPA (suporte de uma linguagem de alto nível para comunicação – ACL - para os agentes MASIF); d) integração com outros serviços CORBA (por exemplo, Trader); e) refinamento dos conceitos "region" e "place".

As interfaces MASIF são definidas ao nível de sistema e não ao nível do agente. Os sistemas de agente e os próprios agentes podem ser, opcionalmente, objetos CORBA. Logo, o "padrão" MASIF não representa uma integração completa e perfeita da tecnologia de agentes com o CORBA. Entretanto, quando os agentes *são* definidos

[4] "Request for Proposals" [OMG 00]

como objetos CORBA, estes, potencialmente, têm acesso a todos os serviços CORBA, a todo software legado “empacotado” (wrapped) por objetos CORBA, etc.

3.9.2 - FIPA

A “organização” FIPA (Foundation for Intelligent Physical Agents), comunga nos interesses da OMG para com a tecnologia de agentes. Desde 1996, o trabalho da FIPA busca orientação rumo ao desenvolvimento e promoção de padrões na área de interoperabilidade de agentes através de um programa avançado de trabalho que reúne a cada trimestre em torno de 50 organizações-membro. Com três anos e meio de experiência e sustentada pela ampla produção de especificações (ver http://www.fipa.org/), a FIPA é uma organização de referência em agentes inteligentes. Uma conexão “formal” foi estabelecida entre a FIPA e o Agents Working Group da OMG: Frank McCabe é o “representante de ligação” da FIPA para o Agents WG e David Levine, do Agent WG, assume o mesmo papel para a FIPA.

Os documentos já produzidos pela FIPA se referem às seguinte áreas:

a) **Agent communication language:** Detalhes de sintaxe e semântica de uma linguagem de alto nível para comunicação de agentes baseada em “speech acts” (ações por palavra). O principal benefício alcançado pelo emprego desta linguagem é a preservação, em escala “aberta” ou livre, da semântica da comunicação. **[**(http://www.fipa.org/spec/f8a22.zip)**]**

b) **Agent/software integration:** Esta especificação detalha uma forma padrão pela qual um software não baseado em agente pode ser integrado a uma plataforma de agentes FIPA. **[**(ftp://ftp.fipa.org/Specs/FIPA97/f7a13pdf.zip)**]**

c) **Agent management:** Tal especificação, também descrita em http://www.fipa.og/spec/f8a21.doc, esboça os detalhes necessários para o gerenciamento dos agentes em uma plataforma de agentes. Um ponto de particular interesse da OMG é a utilização obrigatória do protocolo de comunicação IIOP para os mecanismos básicos de transporte. Contempla-se, entretanto, que outros protocolos também sejam padronizados caso existam necessidades adicionais (por exemplo, para aplicações “wireless”). **[**(http://fipa.umbc.edu/mirror/spec/fipa8a23.doc)**]**

d) **Human/agent interaction:** Detalhes quanto à integração de agentes com a participação de usuários (humanos). **[**(http://www.fipa.org/spec/fipa8a24.zip)**]**

e) **Agent mobility:** Detalha diversos modelos e protocolos de suporte à mobilidade do agente entre as plataformas. **[**(http://www.fipa.org/spec/fipa8a27.doc)**]**

f) **Ontology services:** Detalha especificações para gerenciamento de serviços de ontologia e modelos de ontologia num "framework" consistente e aberto. **[**(http://www.fipa.org/spec/fipa8a28.zip)**]**

g) **Agent naming:** Esta especificação, ainda não adotada oficialmente pela FIPA, descreve modelos para "nomeação" (naming) consistente de agentes. **[**(http://www/fipa.org/spec/fipa9716.PDF)**]**

h) **Message transport:** **[**(http://www.fipa.org/spec/fipa9716.PDF)**]-** Esta especificação, ainda não adotada oficialmente pela FIPA, descreve os requisitos e detalhes dos serviços de transferência de mensagens ("messaging") entre agentes e entre plataformas. Recomenda-se, além disso, a consideração indispensável de: http://www.fipa.org/spec/fipa9710.pdf.

Os trabalhos da FIPA e MASIF são, de certa forma, complementares. Enquanto a especificação FIPA se preocupa, em princípio, com uma comunicação de alto nível para os agentes (ACL, protocolos de negociação, ontologias), a especificação MASIF considera questões de mobilidade e alguns aspectos quanto à interoperabilidade de sistemas de agente heterogêneos.

CAPÍTULO 4
AGENTES MÓVEIS: O FUTURO DA COMPUTAÇÃO DISTRIBUÍDA

4.1 – Introdução

Ao longo dos últimos anos o conceito de “agentes-software” recebeu muita atenção. Dependendo do ponto de vista particular, como apresentado no capítulo anterior, o termo "agente" é associado a diferentes propriedades e funcionalidades, variando de interfaces de usuário adaptáveis, processos inteligentes e cooperativos a objetos móveis. Um dos interesses particulares do trabalho defende a exploração da tecnologia de agentes móveis e dos benefícios chaves fornecidos pela aplicação desta nova tecnologia.

Os agentes móveis compreendem a definição de objetos que possuem comportamento, estado e localização. Tais agentes são autônomos visto que, uma vez invocados, decidem com autonomia a respeito de quais localizações serão visitadas e de quais instruções devem ser executadas. Este comportamento é definido implicitamente através do código do agente ou alternativamente especificado por um itinerário (modificável em tempo de execução).

Os agentes são caracterizados como móveis desde que possam migrar entre localidades que disponibilizam basicamente o ambiente para a execução destes agentes representando uma abstração da rede de computadores e do sistema operacional. O ciclo de vida de um agente é fundamentalmente determinado por um conjunto de eventos relacionados: Os eventos de criação e destruição (“disposal”) podem ser comparados ao escopo dos “constructor” e “destructor” definidos para um objeto. O evento de despacho (disparo para migração) sinaliza que o agente se encontra em fase de partida para uma nova localização. O evento de chegada indica que o agente atingiu, com sucesso, a nova localização. O evento de comunicação notifica ao agente a necessidade de manipulação de mensagens recebidas de outros agentes.

4.2 - Agente Móvel – Uma definição?

A Mobilidade é uma propriedade ortogonal dos agentes. Isto é, todos agentes não são necessariamente móveis. Um agente pode apenas fixar-se e comunicar-se com o ambiente por meios convencionais, tais como várias formas de RPC (Remote Procedure Calling) e "messaging". Denomina-se *agentes estacionários* aqueles que não podem se mover ou "não querem".

> ***Definição de um agente estacionário(Lange 1998b) :*** Um agente estacionário executa apenas em sistemas nos quais iniciou sua execução. Se ele necessita de informação que não se encontra neste sistema, ou precisa de interagir com um agente em um sistema diferente, tipicamente usa um mecanismo de comunicação tal como RPC.

Em contraste, um agente móvel não está confinado ao sistema onde iniciou sua execução. O agente móvel dispõe de liberdade para "viajar" entre os "hosts" de uma rede. Criado em um ambiente de execução, pode levar e/ou transportar seu código e estado para outro ambiente na rede no qual recomeça (retoma) sua execução.

Pelo termo "estado", tipicamente se considera os valores-atributos do agente que ajudam a determinar como proceder assim que recomeça sua execução no ambiente de destino. Pelo termo "código", assume-se, dentro do contexto orientado a objetos, o código da classe necessário para execução do agente.

> ***Definição de um Agente Móvel (Lange 1998b) :***Um agente móvel não está amarrado ao sistema onde iniciou sua execução. Possui a habilidade única de transportar a si mesmo de um sistema para outro em uma rede. Tal atributo permite que um agente se mova para um sistema que contém um objeto com o qual o agente quer interagir e conseqüentemente desfrutar da vantagem de localizar-se no mesmo "host" em que o objeto reside.

4.3 – Taxonomia da Mobilidade

Existe uma taxonomia (Rothermel 1998), esboçada na Figura 4.1, relacionada ao conceito de mobilidade dentro da qual se introduz os conceitos de "***Remote Execution***", "***Code on Demand***", "***Weak Migration***" e "***Strong Migration***".

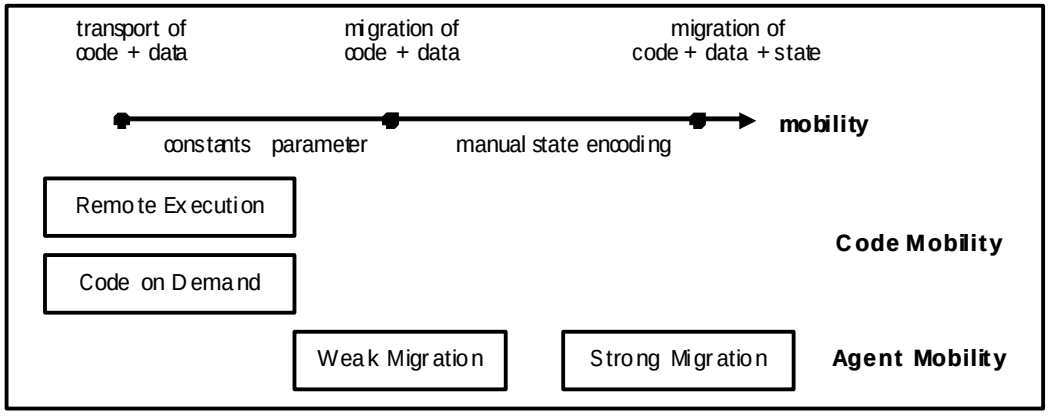

FIGURA 4.1: Graus de Mobilidade
FONTE: Rothermel (1998).

Dentro do conceito de Remote Execution, o "programa-agente" é transferido, antes da sua ativação, para algum nó remoto onde executa "até o fim", isto é, um agente é transferido uma só vez. A informação transferida compreende o código do agente e um conjunto de parâmetros (embora a transferência de código não ocorra necessariamente em tempo de execução, comparar, por exemplo (Hohl 1997), para uma discussão sobre os tópicos de transporte de código). Uma vez ativado, o próprio agente pode utilizar o mecanismo de Remote Execution para inicializar a execução de outros agentes.

Uma abordagem semelhante, configurada no estilo cliente-servidor de interação, é o esquema de Remote Evaluation (avaliação remota), onde uma operação (procedimento + parâmetros) é transferida para um local remoto no qual é executada completamente. Terminada a execução da operação, o local retorna os resultados da mesma ao emissor específico. O mecanismo de avaliação remota pode ser aplicado recursivamente resultando em um modelo de execução "tree-structured" (estrutura de árvore)

Obedecendo ao método descrito acima, o destino do agente-programa a ser transferido é determinado pela "entidade" que inicia e controla a execução remota. Em contraste, no mecanismo Code on Demand, o próprio destino inicia a transferência do código do programa. Se tal método é utilizado em ambientes cliente-servidor, os programas armazenados nas máquinas servidores são carregados (downloaded) para os "clientes" de acordo com as solicitações (on demand). Atualmente, as principais e mais populares tecnologias que suportam este tipo de mobilidade são ActiveX - ver por exemplo, (Aaron 1997) e Java Applets - ver por exemplo, (Sun 1994).

Tanto Remote Execution como Code on Demand implementam mecanismos de suporte à mobilidade de código ao invés de mobilidade de agente, transferindo os programas-agente antes da ativação dos mesmos. Nos dois esquemas a seguir, os agentes (isto é, execuções de programas-agente) podem migrar através dos nós de uma rede de computadores.

Obviamente se considera, para tais agentes, a migração do código e estado do agente. Aborda-se inicialmente o conceito de Strong Migration (migração completa e total) e em seguida a existência do Weak Migration (migração “fraca”). Para tal discussão, assume-se que o estado de um agente consiste em estado de dados (isto é, o conteúdo arbitrário de variáveis de instância ou globais) e o estado de execução (isto é, o conteúdo das variáveis e parâmetros locais e os “threads” de execução).

O grau principal e mais forte do conceito de mobilidade é o Strong Migration (Ghezzi 1997), onde o sistema base captura o estado completo (como um todo, dados e estado de execução) do agente e o transfere junto com o código para a próxima localização. Assim que o agente é "acolhido" pela nova localização, seu estado é reestabelecido automaticamente. Pela perspectiva do desenvolvedor, tal esquema se torna atraente pois a "captura", a transferência e a restauração do estado completo do agente é "transparentemente" realizado pelo sistema suporte.

Por outro lado, a disponibilização deste grau de transparência em ambientes heterogêneos requer, pelo menos, a definição de um modelo global de estado do agente como também de uma sintaxe de transferência para esta informação. Além disso, um sistema de agentes deve prover funções relativas ao estado do agente tais como: “externalize function” e “internalize function”.

Apenas algumas linguagens permitem a "externalização" em níveis satisfatórios tais como em “Facile” (Knabe 1995) ou “Tycoon” (Matthiske 1995). Visto que o estado do agente (estrutura contendo dados e estado de execução) pode ser relativamente grande - em particular para agentes “multi-threaded” - a migração dentro dos moldes da Strong Migration representará uma operação com custo e consumo de tempo altos.

Tais inconvenientes conduziram ao desenvolvimento do conceito Weak Migration, segundo o qual apenas a informação do estado de dados é transferida, permitindo-se que o tamanho de tais "blocos" sejam limitados e que o desenvolvedor selecione as variáveis que compõem o estado do agente.

Como conseqüência, o "programador" se responsabiliza pela codificação dos estados de execução relevantes em variáveis de programa, além de fornecer a forma em que um método **start** “decide”, baseado na informação de estado codificada, o "ponto de reentrada" através do qual se retoma a execução após a migração. Ao mesmo tempo em que pode reduzir substancialmente a porção quantitativa de estado a ser comunicada, tal método **start** impõe uma sobrecarga adicional ao "programador" e complexidade aos “programas-agente”.

4.4 – Os Benefícios Potenciais da Tecnologia de Agentes Móveis

A motivação e o interesse pelos agentes móveis não devem se fundamentar na tecnologia *per se*, e sim nos benefícios alcançados no desenvolvimento de sistemas distribuídos. Desta forma, enumera-se as possíveis vantagens para tal abordagem na tabela abaixo (IEEE 1998b).

TABELA 4.1: Benefícios potenciais da utilização de agentes móveis.

Benefícios Potenciais da Tecnologia de Agentes Móveis	
Redução no tráfego de rede	Os sistemas distribuídos frequentemente dependem dos protocolos de comunicação que envolvem múltiplas interações no cumprimento de uma determinada tarefa. Ainda mais quando medidas de segurança são vitais. O resultado é o enorme tráfego na rede. Os agentes móveis permitem que se "empacote" e se despache uma transação para um host-destino onde as interações se desenrolem localmente. A utilidade dos agentes móveis se percebe quando chegam a reduzir o fluxo de dados brutos (raw data) na rede. Quando armazenados, em grandes volumes, em hosts remotos, os dados deveriam ser processados localmente evitando-se a transferência através da rede. O lema é simples: levar a computação aos dados ao invés de levar os dados para a computação.
Superação (overcoming) à latência das redes de computadores	Sistemas críticos de tempo-real tais como robôs em processos de manufatura necessitam responder a mudanças no ambiente em tempo hábil. Controlar tais sistemas sobre uma rede industrial de tamanho substancial envolve latências significantes e inaceitáveis. Agentes móveis oferecem uma solução, visto que podem ser destacados de um controle central para agir localmente e diretamente executar as orientações do controlador.
Encapsula-mento de protocolos	Durante a permutação de dados num sistema distribuído, cada host possui o código que implementa os protocolos necessários à codificação apropriada dos dados de saída e interpretação dos dados de chegada, respectivamente. Entretanto, frente ao contínuo desenvolvimento dos protocolos, acomodando novos requisitos de eficiência e segurança, a atualização adequada do código de protocolo torna-se um incômodo e, sem exageros, uma missão impossível. Os protocolos se tornam um problema legado. Os agentes móveis, em contrapartida, se movem para hosts remotos a fim de estabelecer "canais", proporcionando tais serviços baseados nos protocolos proprietários.
Execução assíncrona e autônoma	Em geral, dispositivos móveis dependem de conexões de rede frágeis e caras. Ou seja, tarefas que requerem uma conexão contínua entre um dispositivo móvel e uma rede fixa tornam-se provavelmente inviáveis em termos técnicos e econômicos. Tarefas podem ser embutidas em agentes móveis, as quais, então, são despachadas pela rede. Após o despacho, os agentes independem do processo de criação e podem operar com autonomia e assincronia. O dispositivo móvel pode se reconectar, mais tarde, para "recolher" o agente.

FONTE: Green (1995) (continua)

TABELA 4.1: Conclusão.

Adaptação dinâmica	Agente móveis possuem a habilidade de sentir seu ambiente de execução e reagir às mudanças com autonomia. Múltiplos agentes móveis apresentam a capacidade de "auto-distribuição" entre os hosts de uma rede de maneira tal a manter a configuração ótima para satisfazer requisitos ou solucionar problemas particulares.
Suporte, por natureza, heterogêneo.	A computação em rede é fundamentalmente heterogênea de ambas as perspectivas de hardware e software. Como os agentes móveis independem, geralmente, de máquina e camada de transporte, considerando apenas seu ambiente de execução, fornecem condições ótimas para integração sólida de sistemas. O framework de mobilidade desobriga o vínculo de amarração dos agentes móveis a um host determinado. Os agentes podem atuar em qualquer local em que esteja instalado um framework. Os "chips" Java, no futuro, provavelmente dominarão, mas a tecnologia de suporte básico (underlying technology) também evolui em direção a footprint's[1] ainda menores (por exemplo, Jini [Jini]).
Robustez e tolerância às falhas	Reagindo dinamicamente à situações e eventos desfavoráveis, os agentes móveis facilitam a concepção de sistemas distribuídos robustos e tolerantes às falhas. Se um sistema apresenta sinais de mal funcionamento, os agentes móveis podem ser utilizados para aumentar a disponibilidade de certos serviços nas áreas relacionadas. Por exemplo, a densidade de agentes de detecção de falhas e agentes de reparos pode ser incrementada.
Economia de espaço / Eficiência	O consumo de recursos é limitado visto que um agente móvel ocupa apenas um nó por vez. Em contraste, múltiplos servidores estáticos requerem duplicação de funcionalidade em cada localização. Os agentes móveis "carregam" a funcionalidade enquanto migram evitando a duplicação da mesma. Os objetos remotos promovem benefícios similares, mas os custos do middleware podem ser altos. O consumo de CPU é limitado visto que um agente móvel executa apenas em um nó por vez. Outros nós não dispendem "gastos" na execução de um agente sem a justificada necessidade.
Interação com Sistemas de tempo-real	A instalação de um agente móvel próximo a um sistema de tempo-real pode prevenir e/ou evitar atrasos (delay) provenientes do congestionamento de rede. Em sistemas de gerenciamento de rede (Network Management Systems - NMS), agentes network-manager (NM) atuam *in loco* (próximos aos componentes de hardware/software) de modo que esta vantagem pode não ser tão evidente como as outras.
Atualizações de software / Extensão (on-line) de serviços	Um agente móvel pode ser atualizado virtualmente de acordo com a necessidade. Em contraste, a funcionalidade de "swapping" de servidores é complicada; especialmente se existe o propósito de se manter um nível apropriado da qualidade de serviço (QoS). Os agentes móveis podem ser utilizados para estender as capacidades de aplicativos, por exemplo, disponibilizando serviços. Isto permite o desenvolvimento de sistemas extremamente flexíveis.
Paradigma de desenvolvimento conveniente	A criação de sistemas distribuídos e flexíveis pode ser relativamente fácil. A grande dificuldade se relaciona com a exigência de um framework para mobilidade. Ambientes RAD (rappid application development) de alto nível são necessários à medida que a área (agentes móveis) amadurece. É muito provável que as ferramentas consagradas de programação orientada a objetos evoluirão para ambientes de desenvolvimento orientado a agentes, os quais incluirão funcionalidades específicas que facilitarão a mobilidade do agente.

[1] Espaço requerido para instalar ou colocar um dispositivo (externo ou interno). Quantidade mínima de memória RAM que um sistema operacional ocupa.

4.5 – O Paradigma de Agentes Móveis

Segundo a experiência da empresa[2] detentora da patente da tecnologia, os agentes móveis provêem um forte e uniforme paradigma para computação em rede, revolucionando o projeto de desenvolvimento de sistemas distribuídos.

O paradigma de agentes móveis (GMI 1998) integra um número de novos conceitos tais como “places” (espaços), agentes, “travel” (itinerário), “meetings” e “permits” – e outros mais familiares como conexões e autorizações.

Places: O paradigma de agentes móveis modela uma rede como uma coleção de places, que individualmente fornecem serviços aos agentes que chegam. Num mercado eletrônico, um mainframe pode funcionar como um shopping center. Os servidores provêem alguns places e os computadores de usuários fornecem outros. Por exemplo, um home-place que serve como ponto de despacho e retorno para os agentes que o cliente envia para os places do servidor.

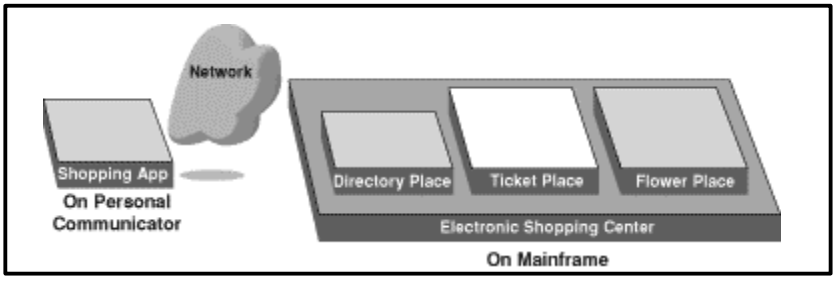

FIGURA 4.2: Computadores como “places”
FONTE: GMI (1998).

Agentes: O paradigma modela uma aplicação como uma coleção de agentes os quais, ocupando particularmente diferentes places, podem se mover de um place para outro. No mercado eletrônico, um típico place é permanentemente ocupado por um agente que representa o place e fornece seu serviço. Por exemplo, o ticketing-agent informa sobre eventos e vende os ingressos, o flower-agent informa sobre arranjos florais e monitora as entregas e o directory-agent informa sobre outros places, e como acessá-los.

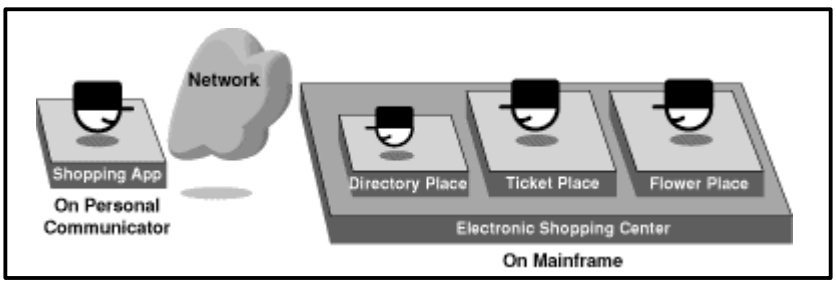

FIGURA 4.3: Agente representando um “place”
FONTE: GMI (1998).

[2] General Magic, Inc.

Travel: O paradigma permite que um agente se mova de um place para outro. Travel é a marca registrada de um sistema em programação remota, através do qual um agente utiliza um serviço remotamente e então retorna para o place de origem. Mover programas entre computadores através de uma rede não é algo novo. Utilizar uma rede local para baixar um programa de um servidor é um exemplo familiar. Mas mover programas enquanto são executados não é usualmente comum. Um programa convencional, escrito por exemplo em C ou C++, não pode ser movido sob estas condições pois nem seus procedimentos nem seu estado são portáveis. Uma linguagem de programação para agentes permite que um computador "empacote" um agente – seu procedimento e estado – e o tranporte para outro computador, permitindo que o agente decida dentro de seu escopo, quando tal movimento seja requerido.

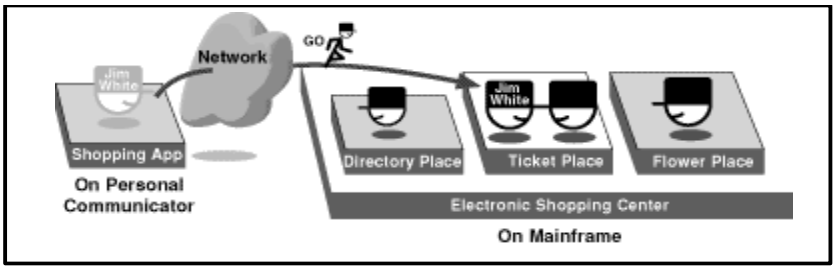

FIGURA 4.4: Agente migrando para "places"
FONTE: GMI (1998).

Meetings: O paradigma permite que dois agentes se encontrem num mesmo place, intercambiando procedimentos entre si. Um agente pode "viajar" para um place num servidor encontrando o agente estacionário que introduz os serviços oferecidos.

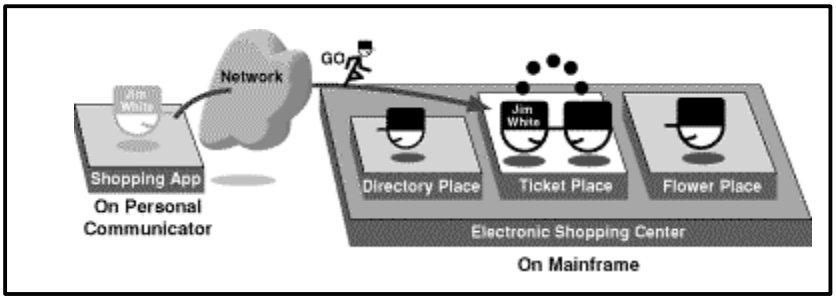

FIGURA 4.5: "Meetings" ou encontros / interações
FONTE: GMI (1998).

Authorities: O paradigma permite que um agente ou place questione, avalie a autorização de um outro agente sempre que este se mova entre regiões de uma rede. A menos que a região de origem possa comprovar a autorização do agente, satisfazendo a região destino, esta rejeita o acesso. Um place pode discernir a autorização de qualquer agente que chega e adotar um esquema em que admita somente agentes com determinadas autorizações. Da mesma forma, pode-se estabelecer tais verificações entre um agente e um place, e ainda entre dois agentes.

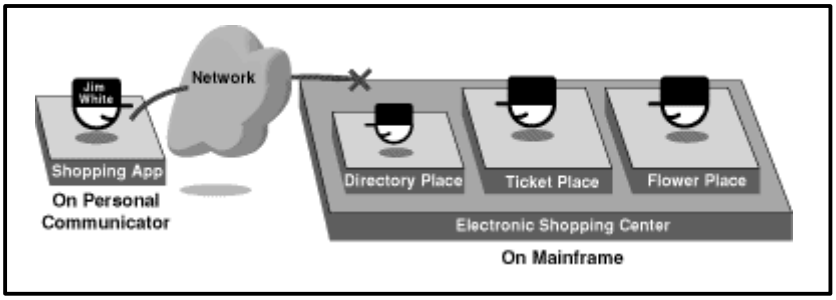

FIGURA 4.6: "Authorities" ou autoridades de um agente
FONTE: GMI (1998).

Connections: O paradigma permite que dois agentes em diferentes places se comuniquem entre si. Se os agentes são uns dos mais recentes paradigmas de comunicação, as conexões contabilizam mais tempo de existência. O paradigma de agentes móveis, portanto, integra os dois.

Permits: O paradigma limita as autorizações dos agentes e places atribuindo-lhes permits – dados que concedem capacidades. Um agente ou place pode discernir suas capacidades mas não incrementá-las. Um permit pode conceder o direito de executar uma determinada instrução (por exemplo, garantindo a um agente o direito de criar outros agentes) ou mesmo utilizar um certo recurso em quantidade pré-fixada (por exemplo, máximo tempo de "vida" em segundos, tamanho máximo em bytes ou quantidade limite de computação). Se ultrapassarem tais limites, o agente ou o place são destruídos.

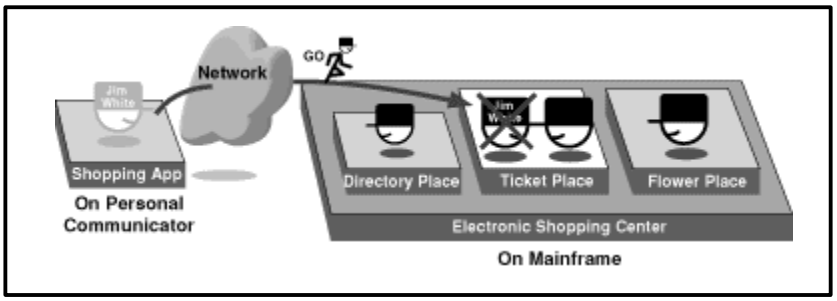

FIGURA 4.7: "Permits" ou permissão para um agente
FONTE: GMI (1998).

Colocando tudo junto: Uma "viagem" de um agente não se restringe a uma rodada única. A força dos agentes móveis surge quando consideramos um agente visitando diversos places sucessivamente. Utilizando os serviços básicos de cada local que visita, tal agente pode desenvolver um serviço composto de alto nível.

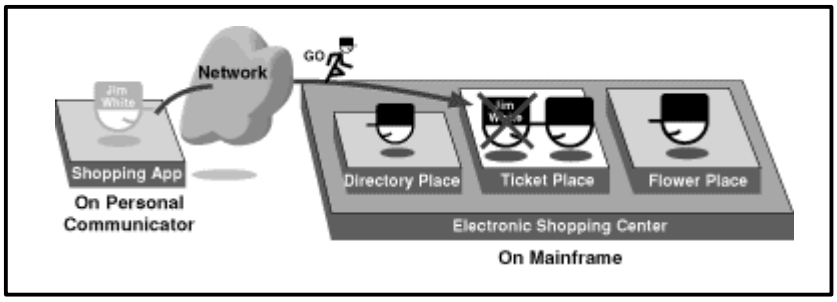

FIGURA 4.8: Agentes com itinerários e/ou objetivos complexos
FONTE: GMI (1998).

4.6 - Sistemas / Plataformas de Agentes Móveis

Em um cenário de computação baseado em agentes móveis, cada "host", ou seja, cada máquina pertencente ao domínio de uma aplicação deve fornecer uma espécie de infra-estrutura ou "docking station" onde os agentes possam atuar desfrutando um ambiente local ou melhor dizendo, uma plataforma de agentes. Tais sistemas ou plataformas são definidas e projetadas com o propósito fundamental de se responsabilizar pela transferência, recepção e disponibilização de um ambiente que forneça serviços, recursos e suporte em tempo de execução para os agentes. As principais características encontradas, em geral, nas plataformas de agentes móveis podem ser resumidas e descritas pelos seguintes serviços:

a) **Suporte à mobilidade:** a migração de agentes é uma operação crucial, dentro do conceito de mobilidade, que deve ser suportada pela biblioteca de serviços da plataforma. Representa a habilidade de um agente suspender sua execução, transferir código e estado para outra máquina (host) e retomar a execução a partir do ponto de interrupção.
b) **Gerenciamento de Recursos:** os agentes devem expressar suas exigências quanto ao consumo de recursos (memória, CPU, largura de banda) e a plataforma deve checar autorizações e as cotas de recursos atuando preventivamente contra a monopolização ou utilização excessiva destes recursos.
c) **Suporte à Execução**: garantir o acesso dos agentes aos serviços e bibliotecas em tempo de execução, garantindo, também, o suporte à criação, suspensão e desativação de agentes.

d) **Suporte à Comunicação:** o suporte a diferentes tipos de mecanismos e protocolos padronizados determinam a quantidade e variedade de "entidades" com as quais os agentes podem interagir.

e) **Serviços de Diretório e Informação:** disponibilizar meios para que os agentes possam verificar e/ou checar a disponibilidade de serviços, identificar a presença de outros agentes e auxílio para a localização de agentes remotos.

f) **Suporte a Segurança:** a plataforma deve garantir a privacidade e integridade dos agentes e de sua própria infra-estrutura. Para tais garantias, torna-se indispensável a aplicação de criptografia para o código do agente e o fornecimento de mecanismos para autenticação, autorização e controle de acesso (ver detalhes quanto à segurança no tópico 4.7.2).

g) **Persistência**: um agente pode sofrer diversas modificações entre sua criação e destruição. O mesmo pode permanente ou temporariamente suspender sua execução e armazenar-se numa estrutura secundária. No caso ideal, deve existir um esquema ou mecanismo permitindo que uma instância de um agente seja serializada para um armazenamento persistente após um tempo indefinido ou quando não existem referências para este agente.

4.7 – O Outro Lado: Algumas "Limitações" dos Agentes Móveis?

Existem muitos tópicos brutos e polêmicos incrustrados nos sistemas de agentes móveis desafiando o andamento e conclusão de pesquisas e projetos de aplicativos-software que visam atingir o patamar das soluções comercialmente viáveis. Em um artigo da publicação IEEE (Internet Computing) (Kiniry 1997), alguns pesquisadores discutindo sobre as limitações da tecnologia de agentes móveis, opinaram: "*Os mais importantes assuntos que devem ser abordados em direção ao tratamento de tais limitações convergem para os tópicos de* ***representação de conhecimento*** *e* ***segurança de rede***.".

Como os agentes representarão seu conhecimento e comunicarão suas necessidades e intenções para os outros agentes? Os mecanismos para interação (meeting) e colaboração disponíveis nos sistemas atuais são extremamente limitados. As interfaces estáticas suportadas pelos mecanismos de comunicação de agentes limitam a flexibilidade e possibilidades para a interação dos mesmos.

Baseando-se nos sistemas de agentes móveis em desenvolvimento e já implementados, verifica-se, ao mesmo tempo, que pouco trabalho é reservado aos tópicos de segurança destes sistemas. Enquanto alguns modelos tradicionais para segurança permitem que os desenvolvedores avancem rumo à solução dos problemas de agentes “mal intencionados”, ou seja, agentes que danificam um sistema ou rede de computadores, pouco esforço tem sido dispensado para outros problemas de mesma categoria. Considerada como uma camada crítica dentro dos sistemas distribuídos, e visto que, por similaridades, impõe questões polêmicas ao desenvolvimento de sistemas de agentes móveis, a segurança exige um tratamento minucioso e imprescindível às tecnologias que lutam pelo reconhecimento e confiabilidade no mercado global.

As seções **4.7.1** e **4.7.2**, a seguir, fornecem um panorama que evidencia a concentração de esforço no tratamento das questões relacionadas à representação do conhecimento e à segurança nos ambientes baseados em agentes móveis.

4.7.1 – KQML: Uma Visão Geral de uma Linguagem de Comunicação para Agentes

Um dos mais promissores trabalhos direcionados à representação do conhecimento é a especificação da linguagem KQML (Knowledge Query and Manipulation Language) (KQML 1993). Definida como formato para mensagens e protocolo para tratamento de mensagens, a KQML foi projetada para suportar o compartilhamento de conhecimento em tempo de execução entre agentes. Mesmo que se assegure o caráter indispensável de tal tecnologia para que os sistemas de agentes móveis possam alcançar os níveis de complexidade de aplicações comerciais (networking, e-commerce, colaboração, ...), não se deve desprezar o fato de que a integração das implementações KQML com os sistemas de agentes configura outra área problemática.

A KQML foi concebida com o intuito principal de prover um formato de mensagens e um protocolo para a manipulação destas mensagens a fim de suportar o compartilhamento - em tempo de execução - de informação entre agentes. Os pontos chave são: **a)** as mensagens KQML não comunicam meramente sentenças em alguma linguagem, mas, ao invés disto, comunicam uma atitude sobre o conteúdo (declaração, solicitação, pesquisa, resposta básica, etc.); **b)** as primitivas da linguagem são chamadas de **performatives**, as quais definem todas as ações permissíveis (operações) que os agentes podem empreender na comunicação entre eles; **c)** a KQML assume que, para os agentes, a comunicação transcorre como transmissão de mensagens "ponto-a-ponto"; **d)** um ambiente KQML pode ser enriquecido com agentes especiais – os **facilitators**, os quais fornecem funções adicionais para que os agentes possam "negociar e/ou lidar" com a rede de comunicação (associação de endereços físicos a nomes simbólicos, registro de agentes e/ou serviços disponibilizados e solicitados por outros agentes, melhorias aos serviços de comunicação como redirecionamento – forwarding, "brokering" e "broadcasting").

A KQML pode ser considerada como uma linguagem de comunicação para a permuta de informação e conhecimento entre agentes, através do uso de um conjunto de tipos de mensagens padronizadas. Segue, abaixo, um exemplo de uma mensagem KQML.

```
( ask-if
      : sender          A
      : receiver  B
      : language  Prolog
      : ontology  foo
      : reply-with      id1
      : content         “bar(a,b)” )
```

Na terminologia KQML, **ask-if** é uma performative. Uma performative “seta” parâmetros introduzidos por palavras chave. Neste exemplo, o agente A (sender) está consultando o agente B (receiver) , utilizando a linguagem Prolog (language), sobre o status de “bar(a,b)” (content). Qualquer resposta a esta mensagem KQML será identificada como id1 (reply-with). A ontologia foo pode fornecer informação complementar em relação à interpretação do “content”.

Supondo que B não esteja “apto” para realizar a ação sugerida por A na mensagem anterior, B retornará uma resposta para A na seguinte forma.

```
( sorry
      : sender          B
      : receiver  A
      : in-reply-to     id1
      : reply-with      id2         )
```

B (: sender) utiliza a performative **sorry** para informar A (receiver) sobre a impossibilidade de avaliar “bar(a,b)”. O agente reconhecerá perfeitamente que tal mensagem se refere a esta avaliação específica pois B utilizou o parâmetro “in-reply-to” com o valor **id1**.

A KQML não é uma linguagem compilada ou interpretada, definindo-se como uma especificação de uma coleção de primitivas para comunicação entre agentes. Por tal razão, convém utilizar a definição “implementação KQML” quando se refere aos mecanismos necessários para que uma aplicação execute envios e recepções de mensagens KQML através da rede.

A primeira implementação de KQML sobre TCP/IP em C foi desenvolvida em UMBC e Lockheed-Martin [FIN 94]. Uma segunda implementação foi construída em

Java (JAT 1996). Uma nova interface e facilidades de distribuição e integração formaram a motivação para se utilizar o padrão CORBA. Os argumentos para tal abordagem se apoiram em dois principais fatores: a) prover uma implementação KQML baseada em outra plataforma de comunicação pode encorajar novos estudos e pesquisas a fim de comparar e avaliar a performance de todas as versões existentes além de contribuir para a expansão da KQML; b) oferecendo serviços tais como "naming services", "trader services", "access control services" e "events services", o padrão CORBA surge como uma solução adequada para facilitar a implementação das performatives de "networking" e dos domínios dos agentes.

A arquitetura (Benech 1997) provê uma implementação KQML baseada no padrão CORBA com o propósito de satisfazer os requisitos de comunicação provenientes da integração da tecnologia de agentes e o gerenciamento de redes e serviços.

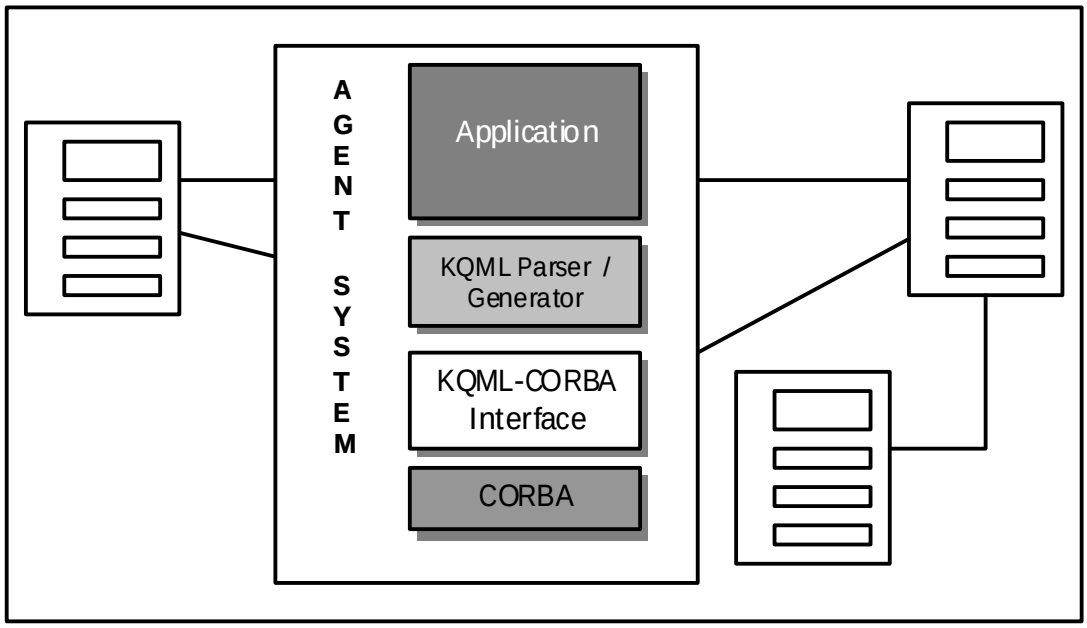

FIGURA 4.9: Arquitetura KQML-CORBA para gerenciamento cooperativo
FONTE: Benech (1997).

A KQML fornece uma linguagem poderosa e uniforme para os agentes atuando em modelos de cooperação, além de encontrar no padrão CORBA a plataforma apropriada para suportar a distribuição transparente dos agentes. A Figura 4.9 mostra um possível cenário em que se destaca uma aplicação de gerenciamento distribuído e cooperativo composta por alguns agentes interagindo com outros agentes utilizando a camada KQML-CORBA.

4.7.2 - Agentes Móveis e a Segurança

A tecnologia de agentes móveis oferece um novo paradigma de computação no qual um programa, na forma de um “agente-software”, pode suspender sua execução em um determinado “host”, transferir-se para outro “host” de uma rede, e retomar a execução neste novo local. A abordagem e a utilização de "código móvel" têm uma longa história nascida através do uso de sistemas "remote job entry" na década de 1960. As "encarnações" atuais de agentes podem ser caracterizadas de diversas formas variando de "simples" objetos distribuídos a software altamente organizado e munido de inteligência. As taxas crescentes da sofisticação do "software móvel" podem ser associadas, em mesma escala, aos perigos e ameaças de segurança.

Esta seção provê uma visão geral sobre as ameaças que desafiam os projetistas das plataformas de agentes e os desenvolvedores de aplicações baseadas em agentes, além de identificar, também, os requisitos ou objetivos de segurança genéricos e uma série de medidas para combater as respectivas ameaças e satisfazer os requisitos de segurança.

4.7.2.1 - As ameaças à Segurança (“Security Threats”)

As ameaças à segurança geralmente se encaixam em três classes principais: revelação de informação, recusa de serviço (Denial of Service - DoS) e corrupção da informação. Existe uma variedade de formas para se examinar tais classes de “perigos” em maiores detalhes enquanto aplicados aos sistemas de agentes. Utiliza-se, portanto, os componentes de uma sistema de agente para categorizar as ameaças obtendo uma maneira de identificar os possíveis alvos e fontes de um ataque. Os agentes móveis oferecem maiores condições para que o abuso e uso não apropriados do conceito alastre os efeitos destes perigos numa escala significante. Existe um número de modelos descrevendo os sistemas de agentes (Fuggetta 1998), (Fipa 1997a) e (OMG 1997); entretanto, na discussão das questões de segurança, torna-se suficiente a utilização de um modelo muito simples compreendendo apenas dois componentes principais: o agente e a plataforma de agentes. Assim, um agente possui código e informação de estado necessária para “carregar” alguma computação.

A mobilidade permite que um agente se mova entre as plataformas de agentes. A plataforma de agentes provê o ambiente computacional no qual os agentes operam. A plataforma a partir da qual o agente inicia a migração é referida como a “home platform”, e normalmente é o ambiente mais “confiável” para um agente. Um ou mais “hosts” podem englobar uma plataforma que pode suportar múltiplos ambientes, ou “places” onde os agentes podem interagir. Visto que alguns destes aspectos detalhados não afetam a discussão dos tópicos de segurança, os mesmos são omitidos do modelo descrito na Figura 4.10.

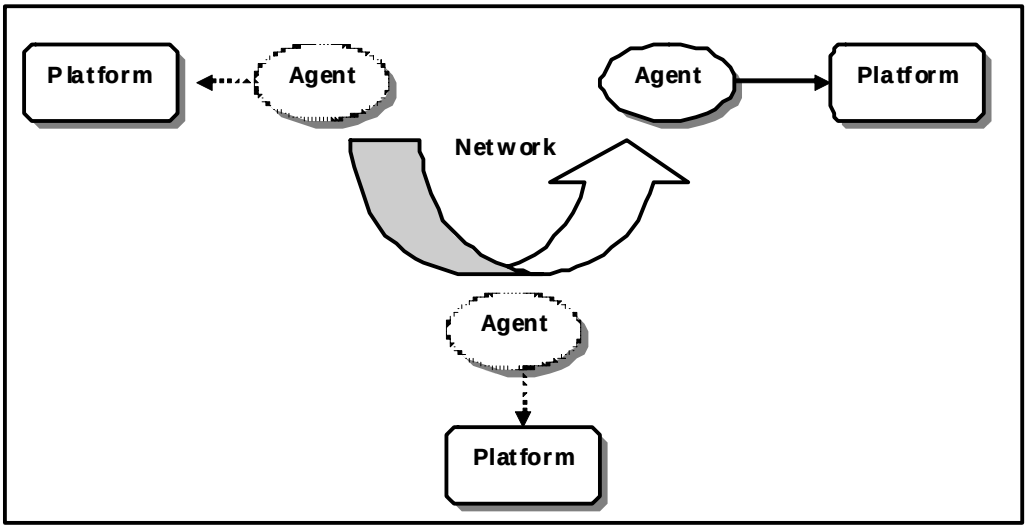

FIGURA 4.10: Modelo de Sistema de Agentes
FONTE: Jansen (1999).

São identificadas quatro categorias de ameaças à segurança:

a) Quando um agente ataca uma plataforma de agentes: esta categoria representa um "jogo" de ameaças no qual os agentes exploram as debilidades de segurança de uma plataforma ou lançam ataques contra a plataforma (ver Tabela 4.2).

TABELA 4.2: Categoria Agente -> Plataforma.

Masquerading (máscaras)	Quando um agente sem autorização declara a identidade de outro agente diz-se tratar de operação de masquerading.
Recusa de Serviço (Denial of Service)	Os ataques dos agentes móveis podem iniciar procedimentos de recusa de serviço (denial of service) ao consumir quantidades excessivas dos recursos computacionais da plataformra. Estes ataques podem ser lançados intencionalmente através de "scripts" de ataque que exploram a vulnerabilidade dos sistemas, ou através de erros de programação não intencionados.
Acesso não Autorização	Mecanismos de controle de acesso são utilizados para impedir que usuários sem autorização ou processos acessem serviços e recursos para os quais não possuem permissão e privilégios especificados política de segurança.

FONTE: Jansen (1999).

b) Quando uma plataforma de agentes ataca um agente: esta categoria representa o conjunto de ameaças através do qual as plataformas comprometem a segurança dos agentes (ver Tabela 4.3).

TABELA 4.3: Categoria Plataforma -> Agente.

Masquerading (máscaras)	Uma plataforma de agente pode "mascarar-se" como outra plataforma a fim de ludibriar os agentes móveis quanto a seus verdadeiros destinos e domínios de segurança correspondentes, extraindo informações dos mesmos. A plataforma "mascarada" pode danificar o agente visitante e a plataforma cuja identidade foi assumida.
Recusa de Serviço (Denial of Service)	Quando um agente chega em uma plataforma, espera que suas solicitações sejam executadas totalmente, que a alocação de recursos seja apropriada e que os contratos de qualidade do serviço sejam cumpridos. Uma plataforma, porém, pode ignorar as solicitações do agente, causar atrasos inaceitáveis para a conclusão de tarefas críticas tais como o registro de ordens de operações num mercado de ações, deixar simplesmente de executar o código do agente ou até mesmo terminar o agente sem qualquer notificação.
Espionagem (Eaves-dropping)	A plataforma pode, além de monitorar as comunicações, interceptar as instruções executadas pelos agentes, os dados públicos e não criptografados e os dados especificamente gerados na plataforma. Visto que a plataforma tem acesso ao seu código, estado e dados, o agente visitante deve ser cauteloso quanto a possibilidade de expor algoritmos proprietários, segredos comerciais, estratégias de negociação ou outras informações importantes. Mesmo que o agente não exponha explicitamente sua informações, a plataforma pode estar apta para deduzir o significado dos tipos de serviços solicitados e da identidade dos agentes com que se comunica.
Alteração	Quando um agente chega em uma plataforma, expõe seu código, estado e dados. A modificação do código de um agente, e por consequência, do comportamento deste agente, pode ser detectada através da inserção de uma assinatura digital ao código do agente. Os riscos de segurança provenientes de quando um agente se move de sua plataforma-origem são considerados como "problemas one-hop", enquanto os riscos associados às múltiplas migrações são tratados como "problemas multi-hop".

FONTE: Jansen (1999).

c) Quando um agente ataca outro agente dentro de uma plataforma de agente: esta categoria representa o conjunto de ameaças no qual os agentes exploram as debilidades de segurança de outros agentes ou lançam ataques contra outros agentes (ver Tabela 4.4).

TABELA 4.4: Categoria Agente -> Agente.

Masquera-ding (máscaras)	Um agente pode tentar disfarçar sua identidade a fim de ludibriar um outro agente com o qual está se comunicando, tentando convencê-lo a fornecer números de cartão de crédito, informações de conta bancária, alguma forma de dinheiro digital ou outra informação privada. Tal artifício (masquerading) prejudica ambos os agentes, o que está sendo "enganado" e aquele cuja identidade foi assumida "corruptamente", em especial quando em sociedades de agentes onde a reputação é estimada e utilizada como meio para se estabelecer confiança (trusting).
Recusa de Serviço (Denial of Service)	Enviando repetidamente mensagens para outro agente, ou utilizando o artifício de spamming[3], as rotinas de manipulação de mensagens dos agentes receptores podem ser sobrecarregadas exageradamente. Agentes que sofrem "ataques-spamming" podem optar pelo bloqueio das mensagens de agentes sem autorização, mas até mesmo esta tarefa requer algum processamento.
Repúdio / Renúncia	O repúdio acontece quando um agente, participando em uma transação ou comunicação, declara posteriormente que tal transação ou comunicação nunca ocorreu. Proveniente de um ato deliberado ou acidental, o repúdio pode gerar "disputas sérias" que dificilmente serão resolvidas a menos que estratégias apropriadas de "contra-ataque" estejam disponíveis.
Acesso não Autorizado	Um agente pode diretamente interferir em outro agente invocando seus métodos públicos (por exemplo, “attempt buffer overflow”, “reset to initial state”, etc.), ou obtendo acesso e modificando os dados ou código do agente. A modificação do código de um agente é uma forma particularmente insidiosa de ataque, visto que pode radicalmente alterar o comportamento do agente. Um agente também pode ganhar informação sobre outras atividades dos agentes usando os serviços de plataforma para "grampear" as suas comunicações.

FONTE: Jansen (1999).

[3] inundação e/ou envio de mensagens sobre um assunto irrelevante a um grande número de receptores ao mesmo tempo.

d) Quando outras "entidades" atacam o sistema de agentes: esta categoria representa o conjunto de ameaças no qual entidades externas, inclusive os agentes e plataformas de agente, ameaçam a segurança de uma plataforma de agente (ver Tabela 4.5).

TABELA 4.5: Outros -> Agente.

Masquerading (máscaras)	Os agentes podem solicitar serviços à plataforma local e remotamente. Um agente em uma plataforma remota pode "mascarar-se" como outro agente e requisitar serviços e recursos para os quais não tem autorização. Tais agentes podem agir em parceria com com uma plataforma mal intencionada. Uma plataforma remota também pode "mascarar-se" assumindo uma identidade que engane tanto outras plataformas como outros agentes.
Recusa de Serviço (Denial of Service)	Os serviços da plataforma podem ser acessados local e remotamente. Os serviços para o agente oferecidos pela plataforma e as comunicações entre plataformas podem ser interrompidos por ataques de "denial of service". As plataformas de agente também estão suscetíveis a todos ataques convencionais (DoS) dirigidos ao sistema operacional e aos protocolos de comunicação.
Acesso não Autorizado (Eavesdropping)	Usuários remotos, processos e agentes podem solicitar recursos para os quais não têm autorização. O acesso remoto a uma plataforma e à própria máquina hospedeira deve ser cuidadosamente protegido, visto que arquivos "script" de ataque convencionais livremente disponíveis na Internet podem ser utilizados para subverter o sistema operacional e obter diretamente o controle de todos recursos.
Copy and Replay	Sempre que um agente móvel migra entre plataformas, a exposição aumenta proporcionalmente os riscos à segurança. Uma entidade que intercepta um agente ou a mensagem de um agente em trânsito pode obter uma cópia dos mesmos, efetuar uma clonagem e retransmití-los. Por exemplo, o interceptor pode capturar uma "ORDEM DE COMPRA" e repetí-la diversas vezes, fazendo com que o agente compre em quantidade superior à programada. O interceptor pode copiar e efetuar "replay" de uma mensagem ou de um agente por completo.

FONTE: Jansen (1999).

4.7.2.2 - Requisitos de Segurança

Os usuários de sistemas de computadores em rede apresentam 5 requisitos principais de segurança: confidencialidade, integridade, disponibilidade, e responsabilidade (account). Os usuários das plataformas de agente móvel também têm estas mesmas exigências de segurança. Esta seção provê uma avaliação breve destes requisitos de segurança e como eles se aplicam às plataformas de agente (ver Tabela 4.6).

TABELA 4.6: Requisitos de Segurança.

Confidencialidade	Qualquer dado privado armazenado em uma plataforma ou transportado por um agente deve permanecer como informação confidencial. As plataformas devem possuir mecanismos que garantam confidencialidade às comunicações internas e inter-plataformas. Alguma entidade "espiã" pode recolher informações das atividades de um agente, não apenas do conteúdo das mensagens trocadas, mas também do fluxo de mensagens de um agente para outros agentes. A monitoração do fluxo de mensagens pode permitir que outros agentes deduzam informações úteis sem ter acesso ao real conteúdo da mensagem. A localização dos agentes móveis também pode ser guardada confidencialmente. Comunicando-se através de um "proxy" o agente pode ocultar sua presença numa plataforma particular. Os agentes podem decidir se a sua localização estará publicamente disponível nos diretórios da plataforma, fazendo com que as plataformas, por consequência, reforcem as políticas de segurança sobre os agentes que optem pelo "anonimato".
Integridade	A plataforma deve proteger os agentes contra modificações não autorizadas de seus dados, código e estado, e garantir que apenas agentes ou processos autorizados realizem alteração nos dados compartilhados. A operação segura dos sistemas de agentes móveis depende, também, da integridade das próprias plataformas locais e remotas. Um "host" mal intencionado pode facilmente comprometer a integridade de um agente móvel enquanto visita uma plataforma remota. A tendência em direção a proliferação de sistemas operacionais e plataformas com código livre (open source) pode facilitar a atuação de administradores ou organizações inescrupulosas modificando tanto a plataforma como o sistema de suporte. Os ataques orientados a metas contra as comunicações dos agentes intentam comprometer a integridade das mensagens alterando o seu conteúdo, substituindo por completo a mensagem, reutilizando uma mensagem antiga, deletando a mensagem ou alterando a origem / destino da mesma.
Accountability (Responsabilidade – "prestação de contas")	Cada processo, usuário ou agente numa dada plataforma deve ser responsabilizado (prestar contas) por suas ações. Logo, devem ser unicamente identificados e autenticados. A "prestação de contas" requer a manutenção de um "audit log" dos eventos ocorridos relevantes à segurança e a listagem de cada evento com o respectivo agente responsável. Mecanismos de autenticação fornecem "prestação de contas" para ações dos usuários. Os agentes devem estar aptos para autenticar suas identidades para as plataformas e para outros agentes. A "prestação de contas" também é essencial para reforçar a confiança entre os agentes e as plataformas visto que um agente autenticado pode obedecer as políticas de segurança da plataforma e ainda exibir comportamento mal intencionado disseminando informações falsas.
Disponibilidade	A plataforma de agente deve possuir capacidades para: a) garantir a disponibilidade de dados e serviços para os agentes locais e remotos, a provisão de concorrência controlada, suporte a acessos simultâneos , o gerenciamento de "deadlock"; b) executar a detecção e recuperação de falhas de software/hardware; c) manipular as requisições de inúmeros "visitantes" e agentes remotos ou ocorrências não intencionadas de recusa de serviço.
Anonimato (Anonymity)	Deve ser cogitada a necessidade de a plataforma assimilar as exigências de privacidade de um agente e as necessidades da plataforma em manter o agente responsável por suas ações. A plataforma pode manter segredo sobre a identidade de um agente não desprezando os modos de reversão do anonimato para determinar a identidade do mesmo quando necessário.

FONTE: Jansen (1999).

4.7.2.3 - “Contramedidas” de Segurança: Protegendo a Plataforma de Agentes

Uma das principais exigências referente à implementação de sistemas de agentes é a garantia de que um agente não possa interferir nos outros agentes e na própria plataforma base. Uma abordagem comum satisfatória se resume no estabelecimento de domínios separados e isolados para o agente e para a plataforma controlando todos os acessos entre domínios. Tradicionalmente, tal conceito é referido como “reference monitor” [CRU 83]. Implementações deste conceito têm surgido desde o início dos anos 80, empregando diversas técnicas de segurança convencionais aplicáveis ao ambiente de agentes tais como:

a) Mecanismos para isolar processos (entre si e isolamento do processo de controle);
b) Mecanismos para controlar o acesso aos recursos computacionais;
c) Métodos de criptografia para cifrar trocas de informação;
d) Métodos de criptografia para identificar e autenticar usuários, agentes e plataformas; e
e) Mecanismos de auditoria aos eventos relevantes à segurança que ocorrem na plataforma.

As técnicas elaboradas para proteger a plataforma de agentes incluem:

a) Software-Based Fault Isolation (Wahbe 1993);
b) Safe Code Interpretation (Ousterhout 1998);
c) Signed Code;
d) Authorization and Attribute Certificates;
e) State Appraisal (Farmer 1996);
f) Path Histories (Chess 1995), (Ordille 1996) , e
g) Proof Carrying Code (Necula 1996).

4.7.2.4 - “Contramedidas” de Segurança: Protegendo os Agentes

Enquanto as medidas tomadas para a segurança da plataforma evoluíram diretamente dos mecanismos empregados para os próprios "hosts", enfatizando assim, medidas ativas de prevenção, as medidas para a proteção dos agentes tendem a ser caracterizadas como medidas de detecção visto que os mesmos estão completamente suscetíveis às plataformas e, não podendo se prevenir contra ocorrências de

comportamento mal intencionados, estão aptos, apenas, para detectá-las (Jansen 1999). Este problema tem como origem a inabilidade para se estender o ambiente "seguro" da “plataforma-home” para as outras plataformas remotas. Algumas técnicas de propósito geral para a proteção dos agentes incluem:

a) Partial Result Encapsulation (Young 1997), (Yee 1997) e (Karjoth 1998];
b) Mutual Itinerary Recording (Roth 1998);
c) Itinerary Recording with Replication and Voting (Schneider 1997);
d) Execution Tracing (Vigna 1997);
e) Environmental Key Generation (Riordan 1998);
f) Computing with Encrypted Functions (Sander 1998); e
g) Obfuscated Code (Time Limited Blackbox) (Hohl 1998).

4.7.2.5 - Algumas Aplicações de Agentes Móveis e Cenários de Segurança

A tecnologia de agentes móveis tem avançado por sobre as fronteiras dos laboratórios de pesquisa em direção a muitas áreas de aplicações comerciais. Destacando algumas entre muitas, tais como “e-commerce”, gerenciamento de redes e “PDAs”, algumas observações podem ser levantadas quanto aos tópicos relevantes de segurança para estes típicos cenários (Jansen 1999):

a) Comércio Eletrônico (“e-commerce”): O nível de segurança exigido para as aplicações “e-commerce” - (Caglayan 1997), (Chess 1995), (Lange 1998) e (GMI 1998) e a sensibilidade (“sensibility”) dos dados e código do agente influenciam diretamente no grau de mobilidade do agente móvel. Como apresentado na Figura 4.11, proporcionalmente ao aumento da sensibilidade de sua tarefa, o projetista pode reduzir o grau de mobilidade do agente. A barra vertical sombreada representa o ponto delimitador para que o desenvolvedor decida quais agentes serão estáticos e quais serão móveis.

Esta decisão se fundamentará nos mecanismos de segurança disponíveis, nos requisitos de performance, na sensibilidade dos dados e código do agente, nos limites de riscos aceitáveis e no nível de funcionalidade exigido.

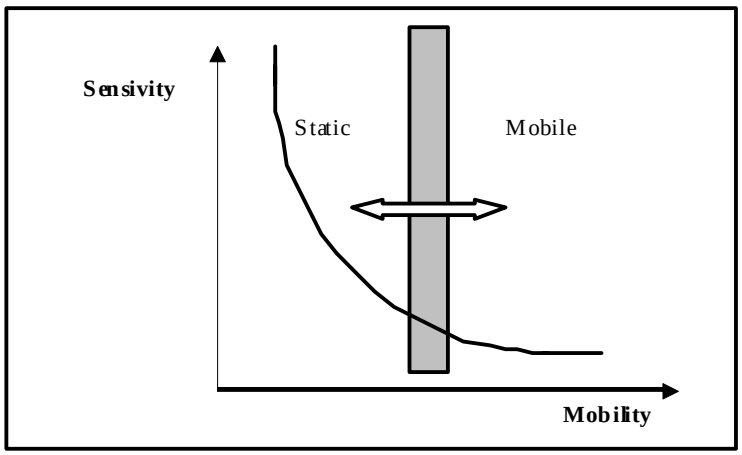

FIGURA 4.11: Graus de Mobilidade X Sensibilidade
FONTE: Jansen (1999).

b) Gerenciamento de Rede: As políticas de gerenciamento de rede permitem apenas o trânsito de código "in-house" (interno ao domínio) ou código de confiança certificada. Somente os administradores de rede autorizados têm permissão para "injetar" agentes na rede, o controle estrito de acesso à plataforma deve ser implantado, o projeto e o desenvolvimento dos agentes móveis devem obedecer aos princípios da engenharia de software e aos métodos de controle de qualidade, mecanismos de "audit log" dos eventos relacionados à segurança devem ser mantidos e analisados (Jump).

c) Assistentes Pessoais ("Personal Digital Assistants" -PDA): No ambiente de atuação destes agentes, os principais tópicos de segurança considerados são confidencialidade, "non-repudiation" e autenticação mútua. Diversos mecanismos convencionais de segurança têm sido aplicados aos sistemas de agentes móveis, destacando-se entre tantos outros, como representante consagrado, o "PKI Privilege Management Extensions" (Jansen 1999).

4.8 - Padrões de Projeto para Agentes Móveis

Após inúmeras considerações quanto à "aplicação" da tecnologia de agentes móveis, torna-se conveniente a abertura de um parênteses para a abordagem de um tópico referente à disciplina de projeto de aplicações baseadas em agentes: "**Padrões de Projeto para Agentes ou "Agent Design Patterns**".

Como em outras tecnologias e pesquisas emergentes, os desenvolvedores e pesquisadores pioneiros repetidamente inventam e re-inventam soluções, às vezes, eficazes, e , outras vezes, nem tanto, para problemas recorrentes. Durante um recente trabalho de desenvolvimento (Lange 1998c) com plataformas de agentes, foram reconhecidos um número de padrões, também recorrentes, no projeto de aplicações com agentes móveis. Vários deles receberam nomes com significado intuitivos tais como *Master-Slave*, *Messenger* e *Notifier*. Tal experiência conta o quão importante é a identificação e formalização de elementos de projeto úteis e re-utilizáveis para aplicativos de agentes móveis. Este é o papel da "disciplina" D*esign Patterns*[1] .

Os padrões descobertos por Danny B. Lange e Yariv Aridor (Lange 1998c) podem ser conceitualmente divididos em três classes: ***Traveling***, ***Task*** e ***Interaction***. Tal esquema facilita a compreensão do domínio e aplicabilidade de cada padrão, a distinção de diferentes padrões e a descoberta de novos padrões. Descreve-se abaixo as três classes referidas como também os padrões em cada classe.

4.8.1 - Padrões da Classe Travelling

A migração é a essência dos agentes móveis. A classe **Travelling** (migração) trata de vários aspectos de gerenciamento dos movimentos dos agentes móveis. Os padrões desta classe, **Itinerary**, **Forwarding** e **Ticket** permitem que se estabeleça o encapsulamento da gerência dos movimentos que, por sua vez, promove o reuso e simplifica o projeto do agente.

[1] conceito originado entre engenheiros de software e pesquisadores da comunidade orientada a objetos (Lange 1998c)

O padrão **Itinerary** (itinerário) é um exemplo da classe **Travelling** que se relaciona com o roteamento entre múltiplos destinos. Um itinerário mantém uma lista de destinos, define um esquema de roteamento, trata casos especiais – tais como decidir qual providência tomar caso um destino não esteja disponível – e sempre conhece o próximo destino. Outro padrão fundamental da classe **Travelling** é o **Forwarding**. Tal padrão permite que um determinado "host" redirecione (encaminhe) automaticamente agentes para outro "host". O padrão **Ticket** foi inicialmente descrito por White [GMI 98]. Conceitualmente, um "ticket" é uma versão "enriquecida" de uma **URL** que engloba requisitos relativos à qualidade do serviço, permissões e outros dados. Por exemplo, pode-se incluir informações "relevantes" ao se despachar um agente para um "host" remoto. Assim, ao invés de insistir, ingenuamente, em migrar para um "host" desconectado, o agente pode, então, tomar decisões racionais enquanto "viaja".

4.8.2 - Padrões da Classe Task

Os padrões da classe **Task** – **Master-Slave** e **Plan** - se relacionam com a quebra de tarefas e como estas serão delegadas para um ou mais agentes. Em geral, as tarefas podem ser diretamente atribuídas a agentes de propósito geral. Além disso, uma dada tarefa pode ser cumprida por um único ou vários agentes, os quais trabalhando em paralelo cooperam para a efetivação do objetivo (no caso de uma busca paralela).

Um padrão fundamental desta classe é o **Master-Slave**, através do qual um agente "master" delega uma tarefa para um agente "slave". Este se move para um "host" destino, realiza determinada tarefa e retorna com um resultado, se a ocasião exigir. O padrão **Plan**, por adotar um conceito de "work-flow" (fluxo de trabalho) através do qual organiza múltiplas tarefas realizadas em seqüência ou em paralelo pelos agentes, é mais complexo. Tal padrão encapsula o fluxo de tarefas, o qual é ocultado do agente. O agente simplesmente fornece a habilidade de mover-se para a realização das tarefas nos destinos específicos. O padrão promove a reusabilidade de tarefas, atribuição dinâmica de tarefas para os agentes além de composição de tarefas.

4.8.3 - Padrões da Classe Interaction

A habilidade de os agentes se comunicarem com outros agentes é vital para a cooperação entre eles. Os padrões da classe **Interaction** - **Meeting**, **Locker**, **Messenger**, **Facilitator** e **Organized Group** - modelam conceitos para facilitar a localização e interação dos agentes.

O padrão **Meeting**, inicialmente descrito por White (GMI 1998), provê meios para que dois ou mais agentes efetuem interação local num dado "host", abstraindo a sincronização de tempo e espaço necessária para interações locais. Os agentes podem se mover para um destino específico, denominado "meeting place", onde são notificados a respeito da chegada dos outros agentes interessados num tópico similar para interação. Os agentes podem explorar o padrão **Locker** para temporariamente armazenar dados privados. Desta maneira, evitam carregar informação desnecessária numa dada ocasião, ao mesmo tempo em que se disponibilizam estes dados para uma futura referência e utilização.

Os agentes podem estabelecer comunicação remota utilizando o padrão **Messenger**, que abstrai o conceito de mensagens na forma de agentes que carregam e despacham mensagens entre agentes. Por exemplo, se um agente "slave" deseja relatar um resultado provavelmente intermediário para o agente "master", ele pode enviar tal resultado através de um agente "messenger" enquanto prossegue na realização de sua tarefa atual. O padrão **Facilitator** descreve um serviço de nomeação e localização para agentes. Convém atribuir, geralmente, um nome simbólico (com significado) ao agente a fim de facilitar sua localização numa ocasião futura. Por exemplo, um agente de coleta de informações movendo-se continuamente pela rede pode ser requisitado em intervalos aleatórios por outros agentes que, em busca de atualizações de dados, não se preocuparam com a localização corrente deste agente "nômade".

Pode-se utilizar o padrão **Organized Group** quando se deseja compor múltiplos agentes em grupos dentro dos quais todos membros viajam juntos (pode-se denominar como padrão "tour"). Tal padrão pode ser considerado um elemento fundamental da colaboração entre múltiplos agentes móveis.

4.8.4 - Exemplos de Alguns Padrões

Foram selecionados um padrão de cada classe apresentada - **Master-Slave** da classe **Task**, **Meeting** da classe **Interaction** e **Itinerary** da classe **Travelling** - para um exame mais profundo. As descrições de padrão seguem um roteiro comum cobrindo os seguintes tópicos: intento, motivação, aplicabilidade, participantes, colaboração e conseqüências. Os diagramas associados e descritos a seguir obedecem à notação orientada a objetos[2] [7].

```
public abstract class Slave extends Aglet {

        Object result = null

        public void onCreation(Object obj) {
                // Called when the slave is created. Gets the
                // remote destination, a reference to the master
                // agent, and other specific parameters.
        }

        public void run () {
                // At the origin:
                initializeJob();
                dispatch(destination); // Goes to destination
                // At the remote destination:
                doJob(); // Starts on the task.
                result=...;
                // Returns to the origin.
                // Back at the origin.
                // Delivers the result to the master and dies.
                dispose();
        }
}
```

FIGURA 4.12: Código exemplo da classe Slave
FONTE: Lange (1998c).

4.8.4.1 – Padrão Master-Slave

a) Intento: o padrão define um esquema através do qual um agente "master" pode delegar uma tarefa para um agente "slave".

b) Motivação: existem várias vantagens que induzem à criação de agentes "master" que por sua vez criam e delegam atividades para agentes "slave". A principal delas é performance. O agente "master" pode dar continuidade a outras tarefas em paralelo com o "slave". A idéia central do padrão é utilizar classes abstratas, **Master** e **Slave**, para verificar as partes

[2] Rumbaugh, J. et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991

invariantes na delegação de uma tarefa entre agentes “master” e “slave”: despacho do “slave” para outros destinos, inicialização da tarefa e tratamento das exceções (“exception”). Os agentes “master” e “slave” são definidos como subclasses de Master e Slave, dos quais somente as partes como “qual tarefa realizar” e “como o master deve processar os resultados” são deixadas a cargo do implementador. Na prática, a classe Master pode ter um método abstrato **getResult** por meio do qual se define como manipular os resultados. A classe Slave tem 2 métodos, **initializeJob** e **doJob**, que respectivamente definem os passos de inicialização antes de o agente migrar e da definição da tarefa propriamente dita. A Figura 4.12 acima apresenta o código exemplo para a classe **Slave** implementada como um “aglet”[3].

c) Aplicabilidade: recomenda-se utilizar o padrão **Master-Slave** nos seguintes casos: 1**)** quando um agente precisa realizar diversas tarefas em paralelo; e 2**)** quando um agente estacionário quer realizar uma tarefa num “host” remoto. As duas situações se referem a tarefas executadas num único destino.

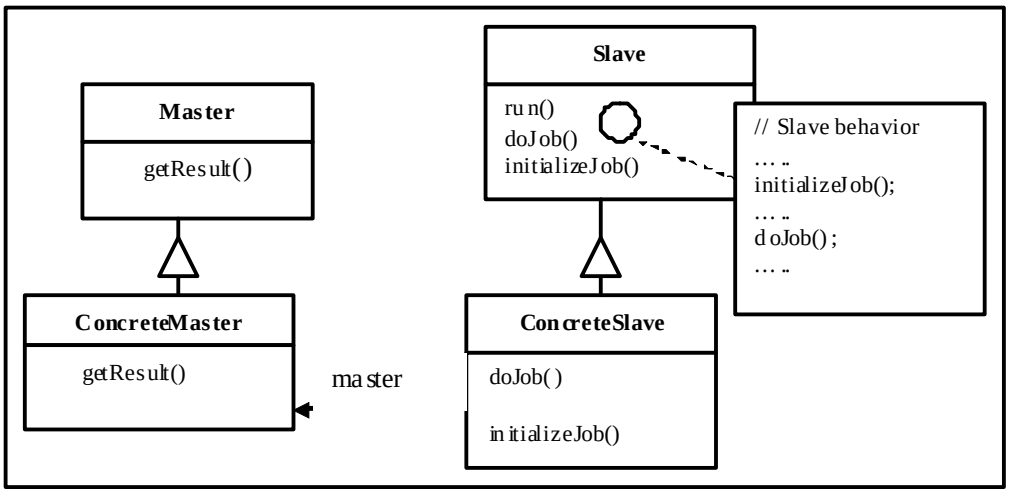

FIGURA 4.13: Participantes no padrão Master-Slave
FONTE: Lange (1998c).

d) Participantes: quatro classes participam do padrão **Master-Slave**. Veja na Figura 4.13 acima os relacionamentos entre as estruturas: **Master** (define um “skeleton” de um agente “master”); **Slave** (define um “skeleton” de um agente “slave”); **ConcreteMaster** (implementa métodos abstratos da classe Master) e **ConcreteSlave** (implementa métodos abstratos da classe Slave).

[3] Agente implementado na plataforma Agles Software Development Kit (ASDK)

e) Colaboração: a colaboração entre os participantes no padrão **Master-Slave** é como segue (ver, também, Figura 4.14): **1)** Um agente "master" instancia um "slave"; **2)** O agente "slave" migra para um host remoto e realiza sua tarefa; **3)** O agente "slave" retorna com os resultados para o "master".

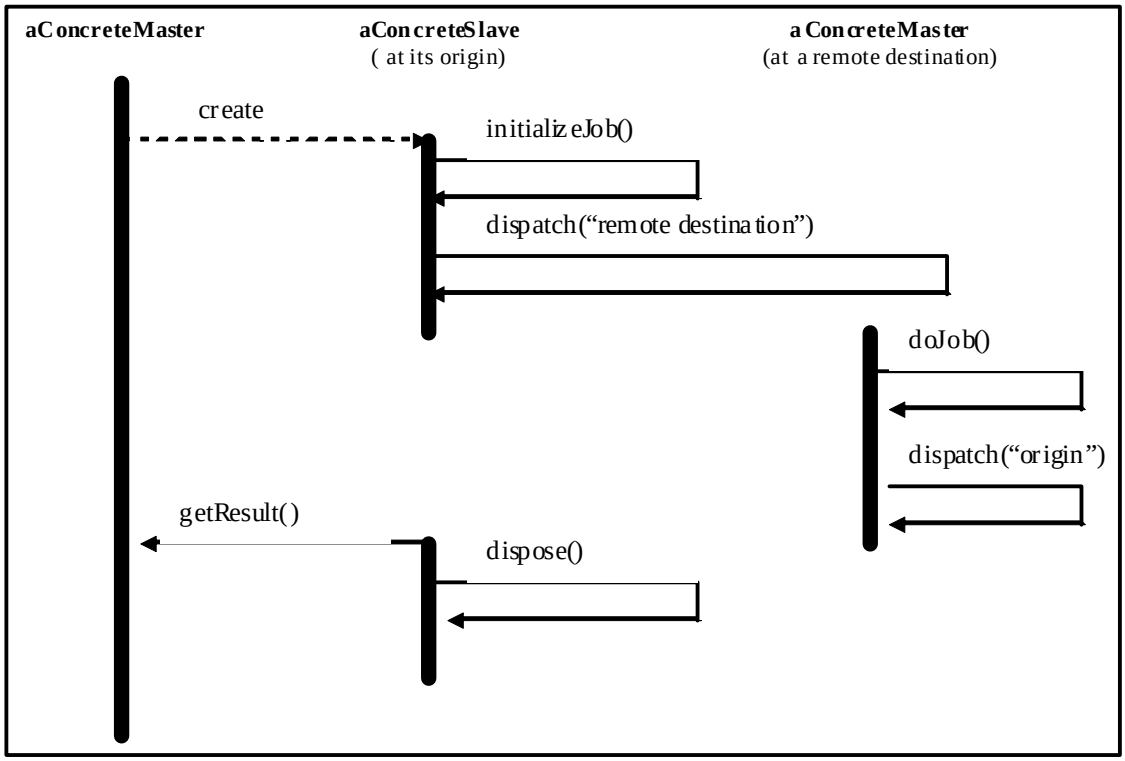

FIGURA 4.14: Colaboração no Padrão Master-Slave
FONTE: Lange (1998 c).

f) Consequências: O padrão provê um meio para o reuso de código entre classes de agente. Na prática, simplifica-se o processo de projeto e implementação pois os desenvolvedores abordam apenas aspectos "variáveis" de agentes pré-definidos – através de bibliotecas padrão ou ferramentas para desenvolvimento de agentes. A desvantagem de um padrão baseado em herança é a determinação em fase de projeto do comportamento de um agente "slave". Por exemplo, um agente não pode ser "transformado" em um agente "slave" em tempo de execução, nem mesmo atribuir "novas missões" para um "slave" pré-definido. Uma versão mais sofisticada deste padrão poderia utilizar um modelo baseado em delegação, no qual a tarefa seria "objectified"[4], possibilitando, assim, a atribuição de qualquer objeto "tarefa" para um agente "slave".

[4] termo, possivelmente traduzido com "abstração / abstrair", utilizado em [Lange 98 c]

4.8.4.2 – Padrão Meeting

a) Intento: O padrão provê meios para que os agentes estabeleçam interações locais em "host" específicos.

b) Motivação: Agentes em diferentes locais podem necessitar de interação local entre si. Considere-se, por exemplo, agentes de comércio instanciados por diferentes usuários (em diferentes "hosts") para busca, compra e venda de mercadorias particulares de seus clientes-usuários. Para tal situação, agentes "buyer" e "seller" precisam se encontrar e interagir dispostos especificamente a "fechar" contratos. A estratégia de se despachar todos agentes de comércio para um local central (denominado "virtual marketplace"), onde possam "negociar", apresenta duas vantagens principais: **1ª)** uma vez que esses agentes deixam suas origens, podem prosseguir em suas negociações mesmo que suas origens (máquina do cliente) sejam desconectadas da rede ou protegidas por um "firewall" (visto que agentes por trás de um "firewall" não recebem mensagens); e **2ª)** suas interações locais proporcionam baixo "overhead" de comunicação quando comparado às taxas das interações remotas. O problema crucial surge na sincronização desses agentes, inicialmente em seus "host-origem", de maneira tal que possam visitar o mercado virtual e estabelecer comunicação entre si. A solução, através do padrão **Meeting**, se constrói pela utilização de uma classe **Meeting** que encapsula um destino específico (local de encontro) e um identificador "**Id"** único. Em geral, agentes que devem interagir localmente estarão "equipados" com um objeto da classe **Meeting**. O agente, ao chegar no local de encontro, utiliza o "**Id"** para localizar o objeto-gerente específico ao local a fim de se registrar (ou seja, adicionar-se na lista de agentes já registrados). O objeto-gerente notificará, então, a chegada do novo agente (tomando uma referência local e vice-versa), de modo que as interações envolvendo o recém-chegado iniciem. Os agentes devem "dar baixa" junto ao gerente antes de deixar o local de encontro. Através dos "**Id"**, múltiplos locais de encontro podem ser instanciados simultaneamente num único "host". Os objetos da classe **Meeting** podem ser distribuídos através de mensagens ou localizados em diretórios centrais.

c) Aplicabilidade: Este padrão é aplicável em todas as situações em que existe a necessidade de que agentes em diferentes "hosts" interajam localmente (no mesmo lugar). Três ocasiões mais comuns nas quais o padrão se adequa são descritas abaixo: **1)** Quando os agentes precisam interagir e o "overhead" relativo à migração para um local central e à interação local é consideravelmente menor do que o "custo" associado à comunicação remota; **2)** Quando os agentes não podem interagir remotamente, visto que estão localizados por trás de um "firewall" ou em "hosts" que apresentam conexões não confiáveis e "low-bandwidth" (por exemplo, "laptops" e computadores portáteis); e **3)** Quando os agentes precisam acessar serviços locais

num “host” determinado. Neste caso, é conveniente que os agentes interajam localmente com o serviço fornecido por um “host” específico.

d) Participantes: Os relacionamentos entre as estruturas dos participantes neste padrão são mostrados na Figura 4.15 abaixo: **Agent** (classe base de um agente móvel), **ConcreteAgent** (uma subclasse da classe **Agent** que mantém objetos “meeting”), **Meeting** (objeto que armazena o endereço do “meeting place”, um identificador único e outras informações relevantes, além de notificar o “MeetingManager” sobre “chegadas”e “partidas” de agentes), **MeetingManager** (este objeto reconhece todos agentes que participam atualmente do “encontro”. Notifica os agentes presentes sobre a chegada de novos agentes e vice-versa).

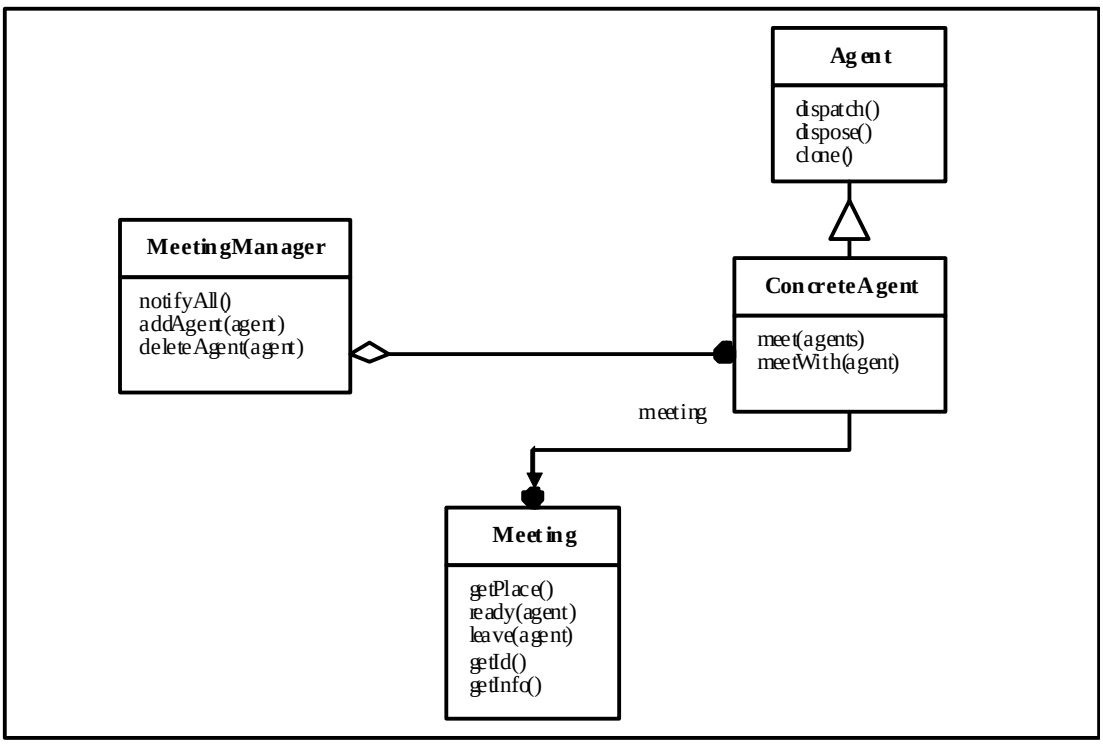

FIGURA 4.15: Participantes no Padrão Meeting
FONTE: Lange (1998c).

e) Colaboração: A colaboração entre os participantes no padrão Meeting está esboçada da seguinte forma (ver Figura 4.16 abaixo): **1)** Um objeto Meeting é criado com um identificador único e um “meeting place”; **2)** Um objeto ConcreteAgent é despachado para o meeting place. Na chegada ao destino, notifica o objeto Meeting apropriado sobre sua chegada através do método ready; **3)** Sempre que um objeto Meeting é informado sobre o recém-chegado ConcreteAgent, localiza o objeto MeetingManager com o qual registra o agente através do método addAgent; **4)** Ao registrar, ConcreteAgent é notificado por meet() a respeito de todos agentes já presentes no local (denotado pelo arrivedAgents no diagrama à frente.) Estes últimos recebem a notícia do agente “novato” através do meetWith(); **5)** Sempre que o objeto Meeting

recebe a informação da partida de um agente, ele localiza o MeetingManager, eliminando o registro através do método deleteAgent().

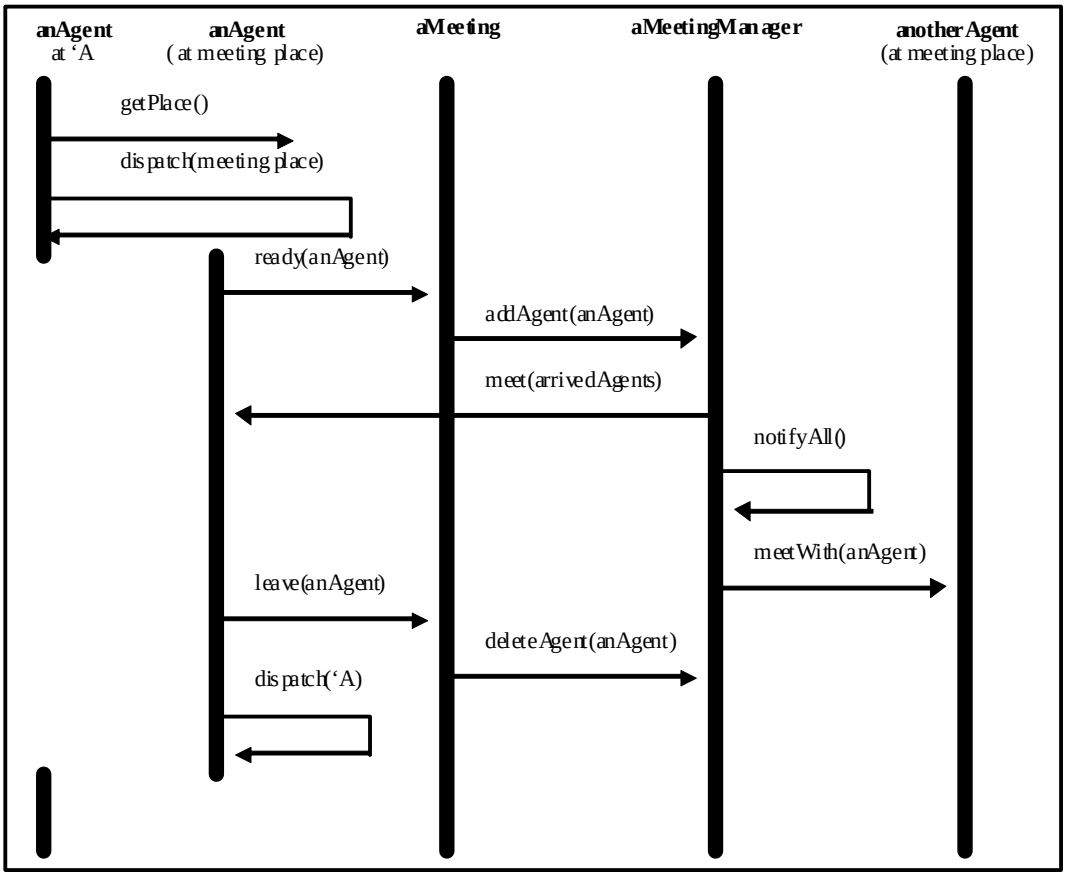

FIGURA 4.16: Colaboração no Padrão Meeting
FONTE: Lange (1998c).

f) Conseqüências: A utilização do padrão **Meeting** tem como conseqüência os seguintes benefícios e desvantagens: **1)** Provê recursos para a comunicação entre agentes indo ao encontro da natureza móvel dos agentes; contrariamente aos objetos estáticos, os agentes móveis não podem manter referências privadas para que automaticamente se localizem entre si e interajam. Tal padrão possibilita a interação local sem considerar a posse de referências dos outros agentes; **2)** Um agente pode interagir com um número ilimitado de agentes; **3)** Simplifica a comunicação entre agentes. O **MeetingManager** pode ser implementado como uma espécie de mediador na transferência ou "multicast" de mensagens entre agentes. Conseqüentemente, as interações "many-to-many" ("todos interagem com todos") são simplificadas (substituídas) por interações "many-to-one" ("todos interagem com um") entre os agentes e o **MeetingManager**. De uma perspectiva diferente, o **MeetingManager** pode manter dados compartilhados por múltiplos agentes e automaticamente notificá-los sobre qualquer mudança nestes dados; e **4)** Podem ocorrer prejuízos entre as interações de baixo "overhead" e agentes "stand-by" (ociosos e/ou inativos) aguardando a chegada de todas as "partes interessadas" (outros agentes) no "meeting place". Logo, em algumas situações, as interações remotas podem ser mais apropriadas.

4.8.4.3 – Padrão Itinerary

a) Intento: Criar objetos para abstração de itinerários e da "navegação" dos agentes entre múltiplos destinos.

b) Motivação: Como entidade móvel e autônoma, um agente é capaz de navegar independentemente por vários "hosts". Especificamente, ele deveria estar apto para manipular "exceptions" tais como "host desconhecido" enquanto se prepara para migrar ou realizar um "tour" completo (por exemplo, retornar a locais já visitados). Poderia, ainda, considerar a necessidade de alteração dinâmica do itinerário. Por exemplo, migrando em busca de informações num "Serviço de Páginas Amarelas", o agente extrai endereços relevantes adicionando-os ao seu itinerário. Consequentemente, é preferível separar o controle da navegação do comportamento e controle de mensagens do agente, promovendo, assim, a modularidade de todas as partes. A idéia chave se resume na transferência da responsabilidade pela navegação do objeto agente para um objeto **Itinerary**. A classe de itinerário fornece uma interface para manutenção e modificação do itinerário do agente e para despacho e/ou disparo a novos destinos. Um objeto agente e um objeto da classe **Itinerary** serão conectados como segue abaixo: O agente criará o objeto **Itinerary** e inicializará o mesmo com: **(1)** lista de destinos a serem visitados seqüencialmente; e **(2)** uma referência para o agente. Em seguida, o agente usará um método **go()** para migrar na direção do próximo destino disponível listado no seu itinerário ou voltar para a origem, respectivamente. Para suportar tais cenários, é necessário que o objeto **Itinerary** seja transferido juntamente com o agente e que suas referências sejam mantidas em todos os destinos.

g) Aplicabilidade: Convém utilizar o padrão **Itinerary** quando se deseja: **1)** Separar os detalhes entre os detalhes da viagem ("tour") e o comportamento do agente promovendo a modularidade de ambas as partes; **2)** Prover uma interface uniforme para viagens seqüenciais de agentes para múltiplos "hosts"; **3)** Definir roteiros que podem ser compartilhados pelos agentes.

h) Participantes: A colaboração entre os participantes no padrão **Itinerary** está esboçada na Figura 4.17 abaixo: **Itinerary** (define uma interface que "navega" junto com um agente); **ConcreteItinerary** (implementa a interface **Itinerary** e rastreia as informações sobre o atual destino do agente); **Agent** (a classe base de um agente móvel); e **ConcreteAgent** (uma subclasse da classe do agente que mantém uma referência para um objeto **ConcreteItinerary**).

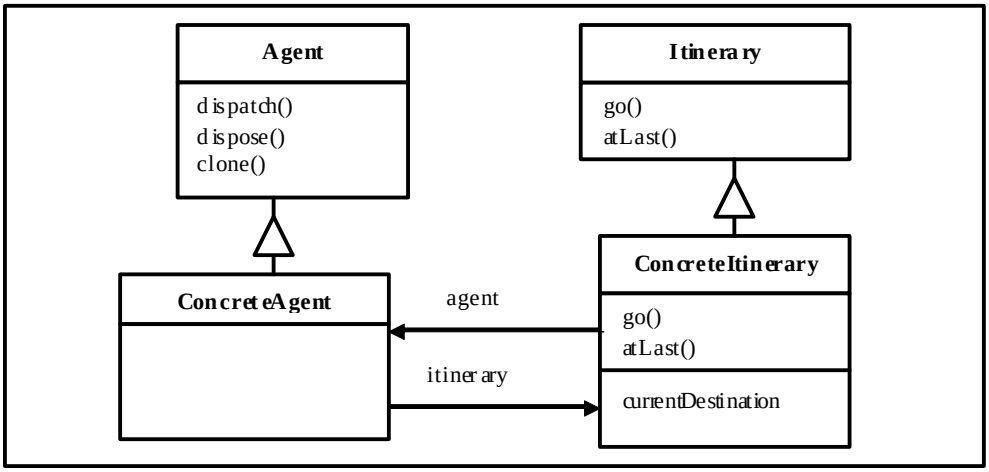

FIGURA 4.17: Participantes no padrão Itinerary
FONTE: Lange (1998c).

i) Colaboração: A Figura 4.18 abaixo representa a colaboração entre os participantes de acordo com o padrão **Itinerary: 1)** O objeto **ConcreteItinerary** mantém-se informado (rastreia) a respeito do destino corrente do **ConcreteAgent** e pode despachá-lo para o próximo local; **2)** Sempre que o **ConcreteAgent** é despachado para um novo destino, o **ConcreteItinerary** também é modificado e suas referências são re-armazenadas no destino-alvo.

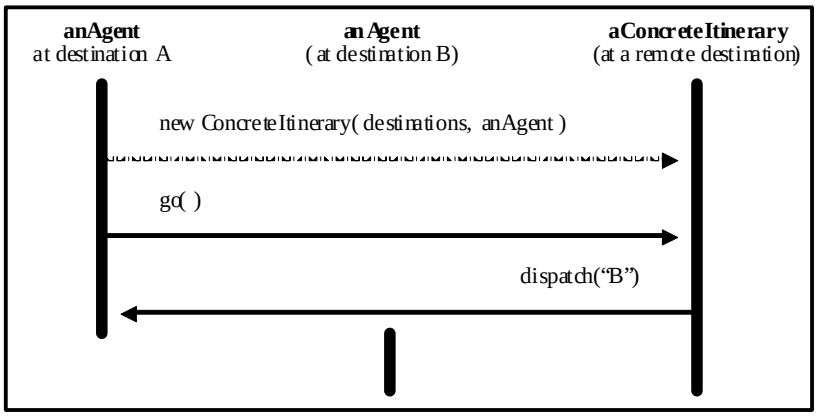

FIGURA 4.18: Colaboração no Padrão Itinerary
FONTE: Lange (1998c).

j) Conseqüências: O padrão **Itinerary** destaca três conseqüências principais: **1)** Suporte à variação da navegação. Por exemplo, uma rotina diferenciada de tratamento de "exceptions" pode ser definida se um agente falhar na migração de um "host" para outro: "***cancelar a viagem e voltar para a origem, tentar acessar outro destino retomando novas tentativas mais tarde***". O padrão facilita tais variações simplesmente através da substituição de um objeto **Itinerary** por outro ou através da definição de sub-classes **Itinerary**. A classe **Agent**, portanto, não sofre modificações. **2)** Facilita compartilhamento de "viagens" por diferentes agentes. Por exemplo, dois agentes podem utilizar o mesmo "tour" para múltiplos usuários; um para agendar encontros entre todos os agentes e outro para despachar mensagens de notificação. **3)** Simplifica a implementação de tarefas seqüenciais. Tarefas variadas podem ser encapsuladas em objetos **Task** enquanto uma classe **Itinerary** é estendida com uma interface para associar objetos **Task** com os destinos. O objeto itinerário mantém informações sobre quais tarefas deve realizar. Sempre que o agente é despachado, ele simplesmente dispara a execução da tarefa atual salva pelo objeto itinerário. Nas plataformas de agentes móveis baseadas em Java, nas quais os agentes são transportados apenas com seu código e dados (sem o estado integral de execução), o padrão **Itinerary** dispensa a necessidade de se manter manualmente informações do estado de execução de um agente, isto é, "o que o agente deve fazer" quando migra (viaja).

CAPÍTULO 5
COMPARAÇÃO ENTRE ALGUMAS PLATAFORMAS DE AGENTES

5.1 – Introdução

A "enxurrada" de resultados devolvidos por uma máquina de busca como Yahhoo, Altavista ou Infoseek para o termo "Mobile Agent Platforms" ou "Mobile Agent Frameworks" pode servir como um termômetro, nada tradicional e sem validade científica comprovada, indicando o interesse explosivo e generalizado pelo assunto, tanto por parte de universidades e centros de pesquisa como também por interesse de companhias líderes em Tecnologia de Informação.

A diversidade de sistemas de agentes móveis já era identificada há alguns anos atrás. A tabela abaixo foi apresentada num artigo (Kiniry 1997) publicado em agosto de 1997 cujo foco limitava-se apenas a sistemas baseados na linguagem Java

TABELA 5.1: Sistemas de Agentes Móveis Java.

Empresas	Homepage	Informações técnicas
Agentsoft	www.agentsoft.com	Assistente para navegação e busca na WWW
Oracle	www.oracle.com/products/networking/mobile_agents/html	Sistema de messaging e banco de dados distribuídos
ObjectSpace	www.objectspace.com/Voyager	Desenvolvimento de aplicativos para sistemas distribuídos utilizando técnicas tradicionais e de agentes móveis. Tecnologia unindo ORB + agentes móveis
Mitsubishi	www.meitca.com/HSL/Projects/Concordia	Aplicações de rede
Microsoft	www.microsoft.com/intdev/agent	Serviços dentro da interface Windows
IBM	www.trl.ibm.co.jp/aglets	Aplicações independentes de plataforma e que rodam em hosts que suportam J-AAPI
General Magic	www.genmagic.com/agents	Aplicações com suporte para acesso a objetos CORBA e BD relacionais via JDBC
KYMA Software	www.info.unicaen.fr/~serge/kyma.html	Solução de problemas através de lógica fuzzy, redes neurais e máquinas de inferência
MCC Co.	www.info.unicaen.fr/~serger/kyma.html	Agentes distribuídos de comunicação e baseado em conhecimento
FTP Software	www.ftp.com/product/whitepapers/4agent.htm	Suporte a funções de gerenciamento de rede para dispositivos IP-connected
Firefly Network	www.firefly.com www.firefly.net	Conteúdo personalizado para sites Web
Crystaliz	www.crystaliz.com/logicware/logicware.html	Suporte a aplicativos Web (interação e coloboração)
Autonomy	www.agentware.com/index.htm	Recuperação de informação com redes neurais

FONTE: Kiniry (1997).

Mergulhado neste “leque de opções”, o objetivo deste capítulo concentra-se fundamentalmente na apresentação de uma análise comparativa entre algumas plataformas de agentes móveis.

A avaliação considera uma amostra, adequada aos propósitos da pesquisa , composta por 6 (seis) plataformas:

1) **Voyager** da ObjectSpace (EUA) (VOYAGER)
2) **Grasshopper** da IKV^{++} (Alemanha) (GRASSHO)
3) **Odyssey** da General Magic (EUA) (ODYSSEY)
4) **Aglets Software Development Kit** da IBM (Japão) (AGLETS)
5) **April** do Imperial College (Reino Unido) (APRIL)
6) **D’Agents** do Darmouth College (EUA) (DAGENTS)

5.2 – Tópicos para Comparação

Os tópicos selecionados para orientar a comparação entre as plataformas eleitas constituem-se de:

a) Características Gerais
b) Life Cycle (ciclo de vida) do Agente
c) Mobilidade do Agente
d) Comunicação do Agente
e) Mecanismos de Segurança
f) Compatibilidade aos Padrões, Capacidade de Inter-Operação e Portabilidade
g) Aplicabilidade e Documentação
h) Informações do Produto

À medida em que se apresenta, em maiores detalhes cada um dos tópicos relacionados acima, poderá ser verificado a composição de alguns comentários relacionados a subtópicos complementares introduzidos oportunamente.

5.3 – Comparação

5.3.1 – Características Gerais

A tabela abaixo compara os seguintes sub-tópicos concernentes às características gerais das plataformas:

Linguagem de Implementação: A linguagem utilizada na implementação da estrutura básica da plataforma e dos agentes.

Arquitetura / Componentes: Os principais componentes da arquitetura.

Tecnologia das Camadas de Transporte e Comunicação: A tecnologia abordada na transferência e comunicação dos agentes.

TABELA 5.2: Características Gerais das Plataformas Selecionadas.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Implementation Language	Platform in C, agents in April	Java	Platform in C, agents in TCL	Java	Java	Java
Architecture/ Components	Agents as lightweight "processes" running within an April invocation, any process can also act as agent station	Aglets as mobile agents, aglet Context as host/ interface to runtime environment	Agents move from agent daemon to another, one per computer, each agent runs in its own interpreter	MASIF compliant	MASIF similarity	Distributed mobile objects as agents connected by an ORB, Space/ Subspace / Super-space concept for broadcast messaging
Transport & Communication Technology	Proprietary protocol based on TCP/IP between special communication daemons, transport uses communication mechanism, limited IIOP support	Proprietary Agent Transfer Protocol (ATP) based on TCP/IP accessible via customisable Agent Transfer Communication Interface (ATCI)	Proprietary protocol based on TCP/IP or on e-mail	TCP/IP, RMI, CORBA IIOP, TCP/IP + SSL, RMI + SSL supported by internal ORB	Via transport API with implementation of RMI (default), IIOP, DCOM	Proprietary protocol based on TCP/IP and CORBA supported by internal ORB

FONTE: Miami (1998).

Comentários

A linguagem Java está presente na grande parte das plataformas de agente. Tal fato, resumindo-se, tem base na adoção crescente e vasta circulação da tecnologia. A linguagem garante real independência de plataforma e provê conceitos de segurança apropriados que vêm ao encontro dos requisitos das plataformas de agente. Os produtos April e D'Agents introduzem linguagens proprietárias incomuns ao "padrão" Java.

Visto que os "candidatos" April e Voyager destinam o foco de prioridades para os processos e/ou objetos móveis numa proporção maior em vista à atenção dispensada aos paradigmas de agentes, não assumem, por natureza, a totalidade do conceito de uma plataforma de agente.

Os "candidatos" April, ASDK e D'Agent suportam apenas protocolos de proprietários. Grasshopper, Odyssey e Voyager fornecem maior flexibilidade suportando diversos protocolos padronizados.

5.3.2 - Ciclo de Vida do Agente

As características relacionadas ao tópico e descritas na tabela a seguir são:

Identificador Global Único (GUID): a organização do "namespace" (Naming Service) dos agentes do sistema. A ocorrência de conflitos deve ser prevista caso o GUID não seja único no âmbito global.

Criação Remota de Agentes: Aptidão para criar agentes "externamente" ao sistema de agentes. Geralmente, tal capacidade também implica na consideração de algumas funcionalidades de gerenciamento através de uma interface remota tais como suspensão, reestabelecimento e deleção.

Funções Especializadas de Life Span (tempo de "vida" / duração): Capacidade de se especificar estratégias de "life span" baseadas, por exemplo, num intervalo de tempo ou na quantidade de "host" visitados.

Persistência para os Agentes: Compreende mecanismos para desativação do agente e "swapping" (troca / permuta / transferência) para um armazenamento secundário permitindo reativações futuras, esquemas de backup e recuperação do mesmo após ocorrências de erros.

Armazenamento em Base de Dados: Intimamente ligado à persistência denotando a habilidade de armazenar os agentes em uma base de dados

TABELA 5.3: Ciclo de Vida (Lifecycle) do Agente.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
GUID	Yes	Yes	Yes	Yes	No	Yes
Remote Agent Creation	No	No	Yes	Yes	No	Yes
Special Life Span Functions	No	No	Yes	No	No	Yes
Agent Persistence	Yes	Not explicitly	No	Yes	No	Yes
Database Storage	Yes	No	No	Yes	No	Yes

FONTE: Miami (1998).

Comentários

Quase todas plataformas fornecem identificadores únicos (**id**) utilizados para referências ao agente (identificação). Apenas o produto Odyssey não apresenta nenhum mecanismo garantindo o caráter único dos **id's**. Os GUID's da plataforma D'Agents têm a limitação de se alterar após cada migração, dificultando o rastreamento de um agente em movimento.

A habilidade de criação remota está presente nas plataformas D'Agents, Grasshopper e Voyager. Tal característica simplifica o processo de instalação de agentes remotos além de permitir a criação de agentes a partir de objetos externos à plataforma presente, um HTTP-browser, por exemplo.

Apenas Voyager e D'Agents provêem funcionalidades para limitar o "life span" dos agentes, as quais podem se tornar imprescindíveis em sistemas muito complexos. Voyager fornece um "garbage collector" (coletor de lixo) distribuído que permite 5 (cinco) modos diferentes para determinar o tempo de vida de um agente, controlando as condições de tempo e "destruindo" este último se necessário. D'Agents integrou as funções de life span no gerenciador de segurança o qual controla, entre outras coisas, as condições de tempo.

Visto que os objetos de aplicações distribuídas frequentemente apresentam "life-span" duradouros, a persistência se firma como uma importante característica sendo utilizada para propósitos de recuperação de falhas e gerenciamento de recursos.
Com April, é possível armazenar agentes no sistema de arquivos ou numa base de dados externa. O gerenciamento da persistência deve ser implementado pelo programador de agentes.

Aglets disponibiliza um serviço de persistência escasso visto que : a) os agentes podem ser desativados por um tempo determinado; b) os agentes desativados são transferidos para um armazenamento secundário; c) não é possível obter uma cópia persistente de um agente em execução.

Grasshopper fornece um serviço mais elaborado. Pode-se atribuir persistência a um sub-domínio completo, incluindo os agentes, através da persistência implícita. Adicionalmente, os serviços de persitência explícita podem ser utilizados para transferir agentes para uma base de dados ou obter uma cópia persistente do agente permitindo ativação automática nos casos de falha do sistema.

Voyager também inclui suporte abrangente. É possível persistir objetos individuais como também agentes "por inteiro" com um único comando. Agentes podem ser reservados à parte (para uma base de dados) liberando-se recursos da plataforma. Os objetos podem ser automaticamente reativados. O dado persistente pode acompanhar (seguir) automaticamente o movimento do agente. Voyager independe da base de dados.

5.3.3 - Mobilidade do Agente

A tabela abaixo compara as seguintes características com respeito à migração do agente:

TABELA 5.4: Mobilidade do Agente.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Serialisation of the Execution Stack	No	No	Yes	No	No	No
Number of Entry Points	One	One	-	Multiple	One	Multiple
Validity of Object References (e.g. via Proxies)	References become invalid	Proxies of migrating objects became invalid, migratin proxies keep valid	References become invalid	Proxies keep valid	References become invalid (no remote communication	Proxies keep valid
Control of Migration	(Agent itself)	(Agent itself), other agents, agent system, user via GUI	(Agent itself)	(Agent itself), other agents, agent system, applications, user via GUI	(Agent itself)	(Agent itself), other agentes, applications
Special Itinerary	No	Yes	No	No	Yes	Yes
Concept of Stationary Agents	No	No	No	Yes	No	No

FONTE: Miami (1998).

Serialização do Estado de Execução: O estado atual de execução pode ser capturado e transferido?

Número de Entry Points: Quantos pontos de entrada existem após a migração?

Validade da Referência aos Objetos: As referências aos objetos permanecem válidas após a migração?

Controle da Migração: Quem está no controle da migração do agente? Pode ser o sistema de agentes ou o próprio agente?

Itinerários Especiais: A habilidade de especificação de itinerários para cenários de múltiplas migrações (multi-hop)

Comentários

A serialização da pilha de execução não é possível nas plataformas baseadas em Java e April. Portanto, pontos de entrada (entry points) fixos são definidos através dos quais o agente retoma suas atividades após a migração. Grasshopper e Voyager permitem a especificação destes pontos de entrada pelo próprio agente.

Apenas a plataforma D'Agents está apta para transportar agentes com toda informação da pilha de execução, o que pode simplificar a programação dos agentes visto que seu "estado" não precisa ser armazenado em estruturas de dados antes da migração.

A ausência da capacidade de serialização é uma desvantagem que pode ser superada por outras estratégias de programação alternativas.

Um problema maior dos agentes móveis é a validade das referências ao objeto. Apenas Voyager e Grasshopper provêem mecanismos que automaticamente atualizam as referências aos agentes em movimento. Em todas as outras plataformas, tal "missão" é reservada aos programadores resultando em overheads para cada agente.

A migração é, de acordo com os conceitos de agentes móveis, controlada pelos próprios agentes na maior parte dos casos. Adicionalmente, ASDK, Grasshopper e Voyager permitem que outros agentes ou as aplicações controlem a migração. ASDK e Grasshopper permitem a movimentação de agentes através de uma GUI.

Em contraste a todas as outras plataformas, Grasshopper distingue agentes estacionários e agentes móveis segundo as especificações MASIF.

5.3.4 - Comunicação do Agente

A tabela abaixo lista os seguintes atributos da comunicação do agente:

Tipo da Comunicação: Descreve o mecanismo básico de comunicação.

Exigência de Geração de Classes Proxy: Existe a exigência de pré-processamento de classes para a geração de "proxys"?

Transparência de Localização: Denota o fato de que na comunicação a localização dos "participantes" (partes envolvidas) é transparente e internamente manipulada por mecanismos básicos de comunicação.

Comunicação Síncrona: A forma de se comunicar sincronamente, ou seja, o transmissor (remete a mensagem) bloqueia a execução até que o receptor (recebe a mensagem) sinalize um retorno.

Comunicação Assíncrona: A forma de se comunicar assincronamente, ou seja, o transmissor dá prosseguimento à sua execução, desconsiderando a sinalização do receptor. O resultado da invocação do método pode retornar num tempo posterior.

Comunicação One-Way: Um caso especial de comunicação assíncrona, onde o transmissor apenas envia uma mensagem e não aguarda um resultado.

Comunicação Multicast: Habilita o cenário onde um transmissor se comunica com múltiplos receptores.

Serviços de Eventos / Call Back: A característica de envio e manipulação de eventos assincronamente.

Redirecionamento de Mensagens: Habilidade de redirecionamento de uma mensagem para um receptor que se encontra em uma nova localização.

Interface para Invocação Dinâmica: Criado pelo padrão CORBA, este termo denota a habilidade de se invocar um método de outro agente quando se constata a ausência do "proxy".

Escopo do Serviço de Diretórios: A visibilidade do serviço de diretórios utilizado.

Comentários

Todas plataformas de agente baseadas em Java, exceto a ASDK, empregam a invocação de métodos para a comunicação entre agentes aproveitando-se da maior flexibilidade na utilização de transmissão de mensagens.

Em contraste a todas as outras plataformas, Odyssey não suporta comunicação remota. Isto se deve ao fato de que não se constatou nenhuma melhoria no último ano (referir-se à época de elaboração do relatório).

TABELA 5.5: Comunicação de Agentes.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Communication Type	Message passing	Message passing	Message passing	Method invocation	Method invocation	Method invocation
Remote Communication	Yes	Yes	Yes	Yes	No	Yes
Proxy-class Generation Required	-	-	-	Yes	-	No
Location Transparency	Yes	Almost transparent	Almost transparent	yes	-	Yes
Synchronous Communication	Yes	Yes	No	Yes	Yes	Yes
Asynchronous Communication	Yes	Yes	Yes	Yes	No	Yes
One-Way Communication	Yes	No	Yes	No	No	Yes
Multicast Communication	Yes	Yes	No	Yes	No	Yes
Event Services / Call Back	No	Yes (simple)	No	Yes (simple)	No	Yes
Message Forwarding	No	No	No	Yes	No	Yes
Dynamic Invocation Interface	-	No	-	Yes	No	Yes
Scope of Directory Service	-	Host	Host	Region	Host	Global

FONTE: Miami (1998).

Grasshopper é a única plataforma que requer a geração de classes "proxy". No Voyager, a geração de "proxy" é efetuada em tempo de execução.

Nas plataformas April, ASDK e D'Agents, a comunicação é transparente quanto à localização apenas enquanto os agentes não se movem, exigindo que a partir do momento em que a migração ocorre, uma nova comunicação seja estabelecida. Grasshopper e Voyager possibilitam a transparência de localização total e completa nas comunicações. As referências aos agentes em movimento são automaticamente atualizadas pelo sistema.

April, ASDK, Grasshopper e Voyager suportam comunicação multicast. Voyager utiliza uma arquitetura especializada com "Spaces" e "Sub-spaces" para o "despacho" eficiente de mensagens.

ASDK e Grasshopper provêem um serviço mínimo (simples) de “callback” para eventos importantes. Voyager implementa uma abordagem arrojada e complexa para serviços de eventos distribuídos.

April, ASDK, D’Agents e Odyssey fornecem serviços de diretório simples que se limitam a encontrar agentes apenas no “host” local. Apenas Grasshopper e Voyager implementam serviços de diretório que disponibilizam suporte abrangente para localização remota de objetos.

Nenhuma plataforma avaliada apresenta atualmente especificações do conteúdo e semântica das mensagens ou implementações de uma linguagem de comunicação de agentes (ACL).

5.3.5 - Mecanismos de Segurança

As características, relacionadas aos mecanismos de segurança, presentes na tabela abaixo são:

TABELA 5.6: Mecanismos de Segurança.

Criterion	April	ASDK	D’Agents	Grasshopper	Odyssey	Voyager
Internal Security Mechanism	None	Aglet Security Manager based on Java Security Manager	Security policies with resource management	Grasshopper Security Manager based on Java Security Manager	Java Security Manager	Java Security Manager
Authentication of Agents toward platform	-	No	Yes	Yes	No	No
Levels of Trust	-	2	2	User grained	(not documented)	2
High Level Access Control	No	Yes, configurable	Yes	Yes, configurable	No	No
External Security Mechanism	No	No	PGP (RSA)	X.509 certificates, SSL (RSA/DES)	No	No
Authentication of platforms toward platform	No	No	Yes	Yes	No	No
Secure Agent Transport	No	No	Yes (integrity and confidentiality)	Yes (integrity and confidentiality)	No	No
Secure Agent Communication	No	No	Yes (integrity and confidentiality)	Yes (integrity and confidentiality)	No	No

FONTE: Miami (1998).

Mecanismo de Segurança Interna: Descreve os mecanismos de segurança da plataforma visando a prevenção contra ataques internos de agentes “mal intencionados”.

Autenticação Agente / Plataforma: O agente, por conta própria, efetua autenticação dirigida à plataforma?

Níveis de Confiança : A granularidade da confiança.

Controle de Acesso: A existência de políticas de controle de acesso configurável pelo usuário.

Mecanismos de Segurança Externa: Descreve os mecanismos de segurança da plataforma visando a prevenção contra ataques externos à plataforma, por exemplo, adoção de protocolos criptografados.

Autenticação Plataforma / Plataforma: A plataforma fornece autenticação entre sistemas de agentes?

Segurança no Transporte do Agente: O transporte do agente é apoiado por segurança criptografada?

Segurança na Comunicação do Agente: A comunicação remota é apoiada por segurança criptografada?

Comentários

A segurança é um pré-requisito para a implantação da tecnologia de agentes em ambientes distribuídos e heterogêneos. April, atualmente, não inclui mecanismos de segurança. Odyssey e Voyager se beneficiam do Java Security Manager mas não provêem funcionalidade elevada. Existem apenas os agentes totalmente “confiáveis” ou os agentes sem confiabilidade alguma.

ASDK suporta controle de acesso de alto nível também apoiado no Java Security Manager, entretanto não inclui mecanismos de autenticação. Apenas as plataformas Grasshopper e D’Agents fornecem segurança interna e externa. Ambas suportam comunicação segura com integridade e confidencialidade, além de permitir que os agentes decidam a respeito da necessidade da segurança externa ou não. D’Agents utiliza o protocolo para codificação não muito apropriado para comunicações de alta velocidade conhecido como RSA.

Grasshopper emprega o protocolo SSL (RSA) para autenticação e permuta de “session key”, e o mais rápido algoritmo DES para codificação de dados. Os agentes são autenticados por conjuntos de “string” próprios. Apenas a plataforma se autentica com métodos criptografados durante o processo SSL. A plataforma provê uma GUI que pode ser utilizada para se configurar políticas de segurança especificamente para cada usuário e suporta o conceito dos certificados X.509.

D'Agents permite a autenticação dos agentes e plataformas utilizando assinaturas digitais. Baseados nesta informação, o agente e a plataforma podem tomar decisões a respeito das requisições de um agente "externo ao contexto". Para se alterar as políticas de segurança, a implementação do gerenciador (segurança) deve ser modificado. D'Agents deve ter acesso às chaves públicas de todos usuários e servidores.

5.3.6 - Compatibilidade aos Padrões, Capacidade de Interoperação e Portabilidade

A tabela abaixo compara os aspectos do presente tópico, entre eles:

TABELA 5.7: Compatibilidade aos Padrões, "Inter-Working" e Portabilidade.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
MASIF compliant?	No	No	No	Yes	No	No
Intention to become MASIF compliant?	Yes	Yes	No	-	Yes	No
FIPA compliant?	No	No	No	No	No	No
Intention to become FIPA compliant	No	No	No	Yes	No	No
Access from and to external objects	External Java and C APIs, limited CORBA2	Java mechanisms	No	Java mechanisms, CORBA2	Java mechanisms	Java mechanisms, CORBA2
WWW Connectivity	No	Additional package (Fiji)	No	Via external interfaces	Not documented	Full applet-to-agent connectivity
Supported Operating Systems	Solaris SunOS Linux	Solaris, Windows95/NT, Linux with JDK 1.1 or higher	Solaris, Linux, AIX, FreeBSD, Digital OSF, IRIX	Solaris, Windows95/NT, Linux with JDK 1.1 or higher	Solaris, Windows95/NT, Linux with JDK 1.1 or higher	Solaris, Windows95/NT, Linux with JDK 1.1 or higher

FONTE: Miami (1998).

Compatibilidade com MASIF: A plataforma é compatível aos padrões MASIF?

Intenção em adotar MASIF: A plataforma pretende se tornar compatível aos padrões MASIF num futuro próximo?

Compatibilidade com FIPA: A plataforma é compatível aos padrões FIPA?

Intenção em adotar FIPA: A plataforma pretende se tornar compatível aos padrões FIPA num futuro próximo?

Acesso a partir de / para Objetos Externos: Como a plataforma interopera com objetos externos ao sistema de agentes?

Conectividade WWW: Habilidade de interagir com sistemas de agentes através da World Wide Web.

Suporte a Sistemas Operacionais e JVM: Quais sistemas operacionais ou Java Virtual Machine são suportados pela plataforma?

Comentários

Atualmente, Grasshopper é a única plataforma de acordo com as especificações MASIF. Nenhuma plataforma avaliadas provê uma linguagem para a comunicação de agentes. Nenhuma plataforma é compatível com o padrão FIPA, destacando-se o anúncio da implementação de uma ACL (padrão FIPA) previsto pela plataforma Grasshopper (efetivada em 1999). Todas plataformas baseadas em Java podem se comunicar com um objeto externo através de mecanismo regulares Java. Se um ORB padrão CORBA estiver instalado, Voyager e Grasshopper adicionalmente permitem o acesso a partir de e para objetos externos via CORBA2. Grasshopper permite conectividade com a WWW e Voyager fornece total conectividade applet-agente.

5.3.7 - Usabilidade e Documentação

A tabela abaixo compara os seguintes critérios concernentes à usabilidade e à documentação das plataformas:

TABELA 5.8: Usabilidade e Documentação.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Configuration	Nothing to customise	GUI	configuration files	GUI or configuration files	Complex scripts necessary	no GUI
Management / Monitoring Utility	No	Yes with GUI	No	Yes with advanced GUI	No	No
Centralised Management Ability	No	No	No	No	No	No
Documentation	Extensive	Incomplete and not very comprehensive	Incomplete	Comprehensive	Incomplete and insufficient	Very comprehensive

FONTE: Miami (1998).

Configuração: Como o sistema de agentes é configurado?

Utilitários para Gerenciamento e Monitoração: Existem ferramentas de gerenciamento e monitoração para o controle do sistema de agentes?

Gerenciamento Centralizado (?): Existem funções que suportam o gerenciamento centralizado de um domínio de sistemas de agentes?

Documentação: A documentação disponível é completa, abrangente e de boa qualidade?

Comentários

A usabilidade das plataformas avaliadas é variada. Enquanto ASDK e Grasshopper fornecem ferramentas gráficas (GUI) úteis para configuração, gerencimento e monitoração, April, D'Agent, Odyssey e Voyager oferecem apenas o ambiente básico sem qualquer ferramenta para monitorar a plataforma de agente. Grasshopper e Voyager disponibilizam documentação "aceitável".

5.3.8 - Informações do Produto

A tabela abaixo compara os seguintes critérios relativos às informações gerais do produto:

Status: o status atual ou versão da plataforma.
Produto Comercial: A plataforma é oferecida comercialmente?
Aplicações: Em quais aplicações a plataforma é utilizada atualmente?
Licença: Quais licenças estão disponíveis?
Pacotes Incluídos: Quais pacotes complementam / estão incluídos na plataforma?
Necessidade de Pacotes Externos: Quais pacotes adicionais são necessários para rodar o sistema de agentes?
Suporte Técnico / Produto: Que tipo de suporte é oferecido para a plataforma?
Futuros Desenvolvimentos: Quais melhorias são programadas para um futuro próximo?

Comentários

Todas as plataformas, exceto Odyssey, estão em constante desenvolvimento. Grasshopper e Voyager são disponibilizadas comercialmente. Voyager é utilizado em inúmeras aplicações corporativas.Grasshopper e D'Agents são utilizados em projetos de pesquisa.

Apenas as companhias IKV (Grasshopper) e ObjectSpace (Voyager) oferecem suporte ao produto.

TABELA 5.9: Informações do Produto.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Version and Status	Version 3.0, Version 4.0 beta, on-going academic project	Version 1.0, first release as formal product, further development not certain	Version 2.0, on-going academic project	Version 1.1, second release as formal product, still in development	Version 1.0 beta2, further development not certain	Version 1.01, first release as product, Version 2.0 beta2 is available
Commercial Product	No	No	No	Yes	No	Yes
Applications	Few research applications	Few example applications, one commercial	Public research	Research projects	Small example applications	A number of applications in industry
Licencing	Free for non commercial use, no source code, Commercial use by arrangement	Free for non commercial use, including source code (not for Tahiti)	Free, including C source code of the core and the adapted TCL interpreter	Free for non commercial use, commercial use and source code with license only	Free for non commercial use, including source code	Free for commercial use without source code, licensing of source code possible
Included Software Packages	Debugger	Tahiti visual agent manager	TCL adapted interpreter	Java Foundation Classes (Swing), Database	None	None
Additional Software Packages	DialoX X(Windows interface), AdB database	Fiji (aglets context on a HTTP-browser), Java-Based Moderator Templates	None	None	None	None
Required External Software	-	JDK 1.1 or higher	None	JDK 1.1 or higher, optional: CORBA 2.0, IAIK security package	JDK 1.1 or higher	JDK 1.1 or higher
Product / Technical Support	Contact via e-mail	No official support, Mailing list (not hosted by IBM), contact via e-mail	Contact via e-mail	Mailing list, full support if licensed, training courses	No support	Mailing list, contact via e-mail, full support if licensed, Partner programs, training courses
Future Development Issues	Refinement of object-orientation and type inferencing	Adding of security related API's, ATCI support for CORBA / IIOP	Persistence agent store, support of other programming languages	FIPA compliant, transaction support	No announcement	DCOM integration, object auditing, security, QoS of messages, agent collaboration

FONTE: Miami (1998).

5.4 – Resultados da Comparação entre Plataformas

Segundo a avaliação do relatório (MIAMI 1998) em que o presente capítulo se baseia, foi traçada uma visão geral compacta dos principais pontos fortes (vantagens) ou pontos fracos (desvantagens) de cada plataforma. As plataformas candidatas que alcançaram os melhores resultados, de acordo com os critérios da comparação, foram respectivamente Grasshopper e Voyager. As características, avaliadas como vantagens, da plataforma Grasshopper compreendem:

a) Compatibilidade à especificação MASIF
b) Interface gráfica (GUI) para configuração e gerenciamento da plataforma
c) Dispositivos para segurança interna e externa
d) Suporte abrangente para persistência
e) Serviço de diretórios geral
f) Atualização automática das referências a objeto dos agentes em movimento
g) Implementação de uma ACL[1] (Agent Communication Language)

A plataforma Voyager, por sua vez, enumera as seguintes vantagens:

a) Suporte abrangente para persistência
b) Atualização automática das referências a objeto dos agentes em movimento
c) Conceito eficiente para comunicação “broadcast”
d) Serviço de Diretórios Geral
e) CORBA ORB embutido
f) Funções para “Life Cycle” (ciclo de vida) abrangentes
g) Eventos distribuídos

Em virtude da apuração de resultados obtida, será apresentada oportunamente, a seguir, uma visão geral das duas plataformas que se destacaram, ressaltando-se desde já que, como conseqüência dos contínuos lançamentos de ambas as empresas de desenvolvimento para seus produtos, as versões consideradas para a presente pesquisa

[1] futuramente, pois ainda não está disponível na versão analisada.

carregam alguns avanços em relação aos exemplares utilizados na confecção do relatório de comparação.

5.5 – Visão Geral das Plataformas em Destaque

O panorama geral sobre as plataformas que obtiveram melhores resultados foi extraído de documentos disponibilizados nos respectivos "sites" da IKV^{++} (GRASSHO) e ObjectSpace (VOYAGER).

5.5.1 – Grasshopper

5.5.1.1 – A Plataforma de Agentes Grasshopper

A plataforma Grasshopper é o primeiro ambiente para desenvolvimento e execução de agentes móveis compatível com o padrão OMG MASIF. A compatibilidade com as especificações FIPA foi anunciada para a versão 2.0 (disponível em 11/1999). As características fundamentais do Grasshopper, quando comparadas às outras plataformas, são: sofisticados mecanismos de segurança (proteção da plataforma contra agentes mal intencionados e proteção agente <-> agente em relação ao controle de acesso, certificados e criptografia), e sua funcionalidade de gerenciamento. São planejadas interfaces para os sistemas de gerenciamento existentes.

Outras habilidades essenciais da plataforma são: integração com tecnologias "*state of the art*" tais como Java e CORBA e a sua extensibilidade. Grasshopper é completamente implementado em Java. Portanto, a plataforma pode ser instalada em qualquer computador que forneça um ambiente de execução Java. Grasshopper pode interagir com outros aplicativos Java e este aplicativos podem ser integrados à plataforma. Não obstante tais considerações, as partes de soluções-software (tais como interfaces, base de dados, programas, etc.) podem ser integradas em soluções baseadas no Grasshopper.

Além disso, a arquitetura "open system" do Grasshopper suporta rápida e fácil adaptação para ambientes específicos e requisitos de aplicações (por exemplo, Windows CE, novos protocolos de comunicação, mecanismos de segurança, etc.) A comunicação

agente-usuário (humano) é suportada integrando-se diferentes interfaces de comunicação tais como “speech” (via telefone), fax, e-mail, SMS, WWW e conversores de mídia (“media converters”). Ou seja, uma aplicação baseada no Grasshopper suporta todos os tipos convencionais de comunicação. Resumindo, existem numerosas possibilidades para a re-engenharia e melhoramentos dos aplicativos existentes nas áreas de: e-commerce, recuperação dinâmica de informações, serviços inovadores de telecomunicações e computação móvel.

5.5.1.2 – Aplicabilidade do Grasshopper

Em geral, Grasshopper é uma plataforma “middleware” universal que manifesta a evolução de software baseado em objetos distribuídos rumo à tecnologia de agentes móveis. Desta forma, Grasshopper concretiza o avanço de ambientes “orientados a objetos distribuídos” (Java/CORBA) em direção a uma plataforma de desenvolvimento flexível para as soluções em software distribuído. Logo, não existem limitações concernentes às aplicações para a plataforma. Figuram como exemplos: serviços avançados (móveis) de telecomunicações, serviços de investigação móveis, serviços automáticos para informação de acidentes, e-commerce / serviços Internet (monitoração de mercado de ações, leilões “on-line’) e tele-medicina (serviços de monitoração de pacientes). Além disso, descreve-se abaixo outros possíveis cenários:

a) **Concretização de Sistemas de Software Distribuído:** Os agentes móveis possuem aptidão para migrar com autonomia para diferentes locais de uma rede a fim de acessar localmente os serviços. Em contraste, o paradigma tradicional “cliente-servidor” depende de interações através da rede.. A eficiência das duas abordagens depende da respectiva aplicação: enquanto pequenas requisições dos servidores são realizadas através da abordagem tradicional (RPC, por exemplo), os múltiplos cenários com interações complexas e “data-intensive” podem se beneficiar dos agentes móveis reduzindo a utilização da rede (“network bandwidth” / disponibilidade) e incrementando a tolerância a falhas da aplicação.

b) **Automatização do Gerenciamento de Software:** O gerenciamento de software, isto é, o controle e a monitoração de componentes-software e suas interações, é um pré-requisito fundamental para se garantir a perfeita execução de sistemas distribuídos. A concepção de aplicativos de gerenciamento utilizando agentes móveis oferece diversas vantagens: a) por meio da mobilidade, os agentes de gerenciamento

podem ser facilmente instalados (ou, ainda, executar auto-instalação) em diferentes nós da rede que devem ser administrados; b) por meio da autonomia, as tarefas de controle e monitoração podem ser delegadas aos agentes de gerenciamento (gerenciamento por delegação).

c) **Execução Assíncrona:** Algumas vantagens essenciais das aplicações baseadas em agentes podem ser alcançadas em muitos cenários onde o acesso à rede é limitado por tempo (por exemplo, via notebook), "bandwidth" (por exemplo, via modem) ou consumo de custos (por exemplo, via telefone móvel). Desejando efetuar processos de rede de longa duração, um usuário pode delegar a tarefa para um agente móvel. O agente é inicializado no dispositivo do usuário e "injetado" na rede, permitindo que tal dispositivo seja desconectado. O agente realiza sua tarefa e aguarda por uma nova conexão do usuário a fim de migrar de volta para o dispositivo e devolver os resultados.

5.5.1.3 – A "Difusão" da Plataforma

O desenvolvimento do Grasshopper foi iniciado em conjunto com a padronização internacional da tecnologia de agentes em 1997. Desde agosto de 1998, disponível comercialmente, a plataforma tem despertado interesse considerável de institutos de pesquisa, bem como de companhias, pelo mundo todo. Um grande incentivo à disseminação da plataforma pela Europa foi gerado a partir de pesquisas de "frameworks" tais como em ACTS (Advanced Communications and Telecommunications Technologies) (ACTS 1998). Os projetos destes "frameworks" se concentram principalmente no desenvolvimento de infra-estruturas de comunicação avançadas para Tecnologia da Informação, baseadas em tecnologias "state of the art" de software. Neste escopo, Grasshopper representa a base tecnológica para concepção de protótipos de sistemas de gerenciamento de redes distribuídas, novas arquiteturas para redes inteligentes (IN – Intelligent Networks) e sistemas de comunicação móveis de 3ª geração. Informações adicionais relacionadas a esses projetos podem ser encontradas na homepage do CLIMATE[2]. A "homepage"[3] da plataforma disponibiliza diversas informações e detalhes sobre o Grasshopper.

[2] http://www.fokus.gmd.de/research/cc/ima/climate/climate.html
[3] http://www.ikv.de/products/grasshopper

5.5.2 – Voyager

5.5.2.1 – A Primeira Plataforma de Agentes para Computação Distribuída em Java

A companhia ObjectSpace, uma líder na tecnologia de software avançada, anunciou seu produto Voyager (04/1997) como a primeira plataforma, no mundo, para computação distribuída em Java aperfeiçoada pela tecnologia de agentes. Voyager é a plataforma mais simples e poderosa para o desenvolvimento de sistemas distribuídos Java™ atualmente no mercado. É a única plataforma Java que integra num único "módulo" o suporte para as técnicas, tradicionais e baseadas em agentes, de programação distribuída possibilitando a criação de sofisticadas aplicações de rede (Voyager 05)

Voyager foi a primeira plataforma comercial que permitia o desenvolvimento de sistemas utilizando mecanismos para comunicação remota e a tecnologia de agentes. Algumas soluções de software se esforçam para a concepção de uma infra-estrutura que utilize a linguagem Java, a tecnologia de agentes e os "ORBs" (Object Request Brokers) superpondo, em camadas, cada uma destas tecnologias. Tais tentativas, entretanto, sofrem um impacto significante em relação à performance, à usabilidade (aplicação) e aos benefícios alcançados (Voyager 05).

Voyager, por outro lado, "captura" os conceitos de tais tecnologias distintas integrando as mesmas numa plataforma elegante e compacta para o desenvolvimento de sistemas distribuídos Java. Segundo John Nordstrom da Sabre Desision Technologies, "*A importância da tecnologia de agentes para a Internet será incrementada na mesma intensidade representada pela Internet para a computação pessoal*". O mesmo avaliou que "*Voyager é a mais poderosa e descomplicada solução para computação distribuída baseada em agentes*."

5.5.2.2 – A força da Plataforma Voyager

Voyager provê uma implementação unificada e compacta de um ORB, disponibilizando os conceitos da tecnologia de agentes ao mesmo tempo em que obtém vantagens direta da linguagem Java. Foi a primeira implementação (absolutamente orientada a objetos) de uma plataforma de agentes onde objetos-agente confiáveis podem se mover e operar com autonomia enquanto se comunicam com aplicativos-cliente através de interfaces, também, orientadas a objetos. Voyager se diferencia de outras plataformas para computação distribuída pelas seguintes razões:

Voyager inclui suporte integrado para a tecnologia de agentes. Voyager permite que se crie – em poucos minutos – agentes que podem migrar por uma rede dando continuidade à sua execução enquanto se movem. Visto que, pela perspectiva da plataforma, um agente é simplesmente um tipo especial de objeto, os agentes em movimento e outros objetos de uma determinada aplicação podem trocar mensagens remotas através da sintaxe formal das mensagens Java.

Voyager facilita a concepção de sistemas distribuídos. As tecnologias tradicionais requerem o cumprimento de processos massivos e propensos a erros para que se "prepare" uma classe para a programação remota. Um único comando Voyager substitue todas estas restrições habilitando automaticamente qualquer classe para a computação distribuída em poucos segundos.

Voyager não requer modificações das classes Java. Voyager pode, remotamente, construir e se comunicar com qualquer classe Java, mesmo que pertencentes à bibliotecas externas ("third-party"), sem código algum. Outras tecnologias, em geral, requerem a utilização de arquivos "**IDL"**, definições de interfaces e modificações nas classes originais que, em conjunto, consomem tempo de desenvolvimento e dificultam o aparelhamento das classes de um domínio específico a um "ORB" particular.

Voyager possui ótima integração com a linguagem Java. Os objetos podem ser construídos remotamente através da sintaxe formal de construção Java, os métodos estáticos podem ser executados remotamente, as "exceptions" remotas são automaticamente "redirecionadas" (rethrown) para o "caller" e os objetos serializáveis podem ser passados e "retornarem" por valor. Todas estas características simplificam e aceleram o desenvolvimento dos programadores Java.

Voyager inclui suporte integrado para a mobilidade de objetos. Uma vez criados, os objetos serializáveis podem ser movidos para uma nova localização, mesmo enquanto estejam recebendo

mensagens. As mensagens enviadas para a "antiga" localização são automaticamente redirecionadas para a localização atual.

Voyager é "rápido". As mensagens remotas são tão rápidas quanto os "ORB's" CORBA. Além disso, as mensagens despachadas através de agentes móveis são, em média, 100.000 vezes mais rápidas do que os outros "ORB's" Java.

Voyager é "pequeno". O sistema Voyager, completo e em formato de arquivo "**.JAR**", ocupa 70 KB (descompactado: 150 KB).

Voyager é "geral". A versão 2.0.2 inclui suporte para mensagens "one-way", "future" e "sync" utilizando comunicação TCP; sistema para eventos distribuídos (arquitetura Space), comunicação em grupos, serviço de diretórios distribuído, comunicações UDP, facilidades para rastreio de mobilidade, integração CORBA e Agregação Dinâmica.

Voyager tem a melhor conectividade. Alguns "ORBs" impedem que os objetos (no browser) se comuniquem com objetos que não estejam localizados no servidor de "browsers". Voyager não impõe tal restrição e inclui um roteador integrado que permite a comunicação com objetos localizados em qualquer posição do sistema (autorizados pelo "firewall").

Voyager é 100% Java. As aplicações Voyager carregam a "marca": "*write once and run anywhere*".

CAPÍTULO 6
DESTAQUES DA PLATAFORMA VOYAGER

6.1 – Introdução e Considerações Especiais

Serão apresentados através de conceitos teóricos e de exemplos, no decorrer deste capítulo, tópicos de destaque que se caracterizaram como fundamentais e promissores tanto para o escopo ou abrangência dos objetivos traçados para a idealização do protótipo do Serviço de Agentes, como para futuras implementações do serviço que integrem a tecnologia de objetos distribuídos e a tecnologia de agentes móveis.

Dentro de um ambiente de computação distribuído, em especial naqueles fundamentados na tecnologia de objetos distribuídos, os "ORBs" (Object Request Broker – tais como CORBA, DCOM e RMI) atuam como arquiteturas de suporte, permitindo que desenvolvedores criem objetos remotos para os quais enviam mensagens como se todos "residissem" na mesma localização. Em geral, tais arquiteturas incluem serviços característicos como "distributed garbage collector", diferentes modos de "messaging" (transferência de mensagens) e "naming service". Entretanto, nenhuma destas arquiteturas suporta mobilidade de objetos ou a tecnologia de agentes móveis e autônomos.

As plataformas de agentes, como algumas das apresentadas no capítulo anterior (comparação de plataformas), permitem a seqüência de operações triviais à tecnologia de agentes tais como a criação de um agente, a programação do mesmo com um conjunto de tarefas e o "despacho" do agente dentro da rede para que realize sua missão. Entretanto, a grande maioria destas plataformas possui o suporte mínimo para tópicos básicos de computação distribuída além de diferenciar o tratamento entre objetos e agentes. Por exemplo, a plataforma ASDK (Aglets) utiliza "sockets" e a plataforma Odyssey utiliza RMI na migração de agentes entre máquinas. Mas, nenhuma destas plataformas permite o envio de mensagens Java para um agente estacionário ou para um agente em movimento. Desta forma a comunicação de objetos com um agente - após a migração do mesmo - e a comunicação "direta' entre os agentes representam um requisito, quando não totalmente inviável, de solução muito complexa.

O diferencial de destaque da plataforma Voyager advém da integração da tecnologia de computação distribuída (sistemas distribuídos, objetos distribuídos, etc.) com a tecnologia de agentes. E o aspecto mais importante a se evidenciar desta "integração" é que a tecnologia de agentes participa como complemento evolutivo dentro de uma estratégia de desenvolvimento da empresa ObjectSpace que, segundo o relatório da IDC (International Data Corporation) (Garone 1998), se concretiza como: " ... ***Na prática, ao embutir 'habilidades' de agentes dentro do produto Voyager, a ObjectSpace realiza avanços significativos na tentativa de evidenciar a tecnologia de agentes. Entretanto, os agentes compreendem uma pequena porção da estratégia de desenvolvimento de produtos da empresa. O objetivo final da ObjectSpace se consuma em um produto que permita a criação de um ambiente no qual objetos, agentes, e em geral, todos clientes e servidores possam interoperar sem se "preocupar" com os modelos básicos de componentização e/ou de infra-estrutura (DCOM, CORBA, e Java RMI).***" . Portanto, a filosofia do Voyager defende que um agente é, de fato, um tipo especial de objeto que pode se mover com autonomia, retomar sua execução enquanto se move e, exceto tais peculiaridades, se comporta exatamente como um outro objeto qualquer.

6.2 – A Utilização de Interfaces para a Computação Distribuída

A linguagem Java suporta uma característica chamada ***interface***. Não contendo código, uma interface relaciona o conjunto de "representações e/ou assinaturas" de métodos que devem ser definidos por qualquer classe que implemente esta interface. Uma variável cujo tipo é uma interface pode se referir a qualquer objeto cuja classe implemente a referida interface. Por convenção, sem se determinar que o código de uma aplicação deve seguir impreterivelmente a regra, os nomes da interfaces Voyager começam com "I". Descreve-se, abaixo, um exemplo de uma interface (Voyager 04):

```
public interface IStockmarket
{
int quote ( String symbol );
int buy ( int shares, String symbol );
int sell (int shares, String symbol);
void news ( String announcement);
}
```

Se uma classe, por exemplo, **Stockmarket**, implementa a interface **IStockmarket**, pode-se escrever:

```
IStockmarket market = new Stockmarket(); // a variável market se refere ao objeto local
```

O Voyager apresenta tal característica de linguagem para simplificar a computação distribuída. Um objeto remoto é representado por um objeto "proxy" especial que implementa as mesmas interfaces de sua "duplicata". Uma variável cujo tipo é uma interface pode se referir a um objeto remoto via "proxy", visto que ambos, o objeto remoto e o "proxy", implementam as mesmas interfaces.

6.3 – A Criação de Objetos Remotos

Para criar um objeto numa localização específica, deve se utilizar o método **Factory.create()** o qual retorna um "proxy" para o objeto recentemente criado e gera dinamicamente a classe "proxy", caso não exista. Existem diversas variações do método **create()** dependendo de situações tais como quando um objeto é criado localmente e o construtor de uma classe recebe argumentos. Por exemplo, para se criar uma instância "default" da classe **Stockmarket** em um programa local e outra instância em um programa "rodando" na porta 8000 da máquina "dallas", a sintaxe seria a seguinte:

```
IStockmarket market1 = (IStockmarket) Factory.create( "Stockmarket" );
IStockmarket market2 = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000" );
```

A porção de código abaixo descrita é utilizada para se criar uma instância da classe **Stockmarket** utilizando o construtor que recebe tipos **String** e **int**:

```
Object[] args = new Object[] { "NASDAQ", new Integer( 42 ) };
IStockmarket market3 = (IStockmarket) Factory.create( "Stockmarket", args, "//dallas:8000" );
```

6.4 - Advanced Messaging: a transferência de mensagens no Voyager

Uma das visões mais convincentes para o futuro da tecnologia de agentes é um "mundo" no qual agentes especializados colaboram entre si a fim de atingir um objetivo que não pode ser alcançado por um agente individual. Tais agentes deveriam se mover, quando apropriado, para realizar interações ("conversas high-bandwidth") que possivelmente não seriam executadas em redes com baixa "largura de banda". Uma plataforma que permita a construção de tais ambientes, portanto, deveria disponibilizar o suporte a um sistema poderoso e robusto de "messaging[1]" através do qual os agentes móveis e os objetos se comunicassem diretamente, mesmo quando em movimento (agentes móveis)

O Voyager permite que um agente envie mensagens regulares Java para outros agentes (mesmo para aqueles em movimento) sem se preocupar com a localização dos mesmos na rede. Além disso, permite que se atribua "aptidão remota" (classes "remote-enabled") a uma classe Java sem qualquer modificação. Uma vez habilitada, esta classe pode ser instanciada facilmente em qualquer lugar da rede e receber mensagens Java.

Cada vez que um objeto se move, deixa um objeto "forwarder" que re-direciona as mensagens para a nova localização do objeto. Quando um objeto "morre", seus "repetidores" também morrem. Visto que o Voyager trata um agente como um objeto especial, os agentes podem ser criados remotamente e mensagens podem ser enviadas a eles enquanto se movem.

Mensagens síncronas podem ser enviadas no Voyager através da sintaxe normal do Java. Entretanto, visto que muitas aplicações necessitam de maior flexibilidade, o Voyager fornece uma camada de abstração de "messaging" que suporta características mais sofisticadas de transferência de mensagens

[1] Transferência de mensagens

6.4.1 - Invocação Dinâmica de Mensagens: Mensagens Sync

Por "default", as mensagens no Voyager são síncronas. Quando um "caller" envia uma mensagem síncrona, o "caller" bloqueia (interrompe o processamento) até que a mensagem se complete e o valor de retorno, se existir, chegue . Por exemplo, a linha seguinte de código envia uma mensagem síncrona **quote()** para uma instância de **Stockmarket**:

```
int price = market.buy (42, " SUN ");
```

Pode-se enviar uma mensagem síncrona dinamicamente utilizando o método estático **Sync.invoke()**, que retorna um objeto de **Result** quando a mensagem se completar. Examinando, então, o objeto **Result**, obtém-se o resultado: um valor ou uma exceção. Para enviar uma mensagem síncrona, invoque **Sync.invoke()**, passando os seguintes parâmetros: "*objeto destino*", "*nome do método a ser invocado no objeto destino*" e "*parâmetros para o método invocado dinamicamente em um objeto array*".

Por exemplo, a seguinte linha de código usa **Sync** para invocar dinamicamente uma mensagem **quote()**[2] em uma instância de **Stockmarket**:

```
Result result = Sync.invoke (market, " buy ", new Object [] {new Integer (42), " SUN "});
int price = result.readInt ();
```

Argumentos primitivos devem ser enviados como seus **Object** equivalentes. Na maioria dos casos, o simples nome do método será suficiente. Porém, se existe mais que um método com o mesmo nome no objeto destino, o nome do método deve ser especificado com os tipos de argumento através da sintaxe "***método(tipo1, tipo2)***".

Os seguintes métodos são utilizados para a consulta dos objetos **Result**. Note que no caso de métodos síncronos, o valor de resposta (reply) está sempre disponível assim que tais métodos são invocados. As mensagens **Future**, discutidas mais tarde, permitem que métodos sejam invocados antes que o valor de resposta cheguem.

[2] o código completo da classe **StockMarket.java** se encontra no Capítulo 8

- **IsAvailable()**

  Retorna **true** se **Result** recebeu seu valor de retorno.

- **readXXX()**, onde **XXX = Boolean, Byte, Char, Short, Int, Long, Float, Double, Object**

  Retorna o valor de **Result**, bloqueando até que o valor chegue ou o período de tempo de **Result** transcorra. Se o valor não é recebido dentro de um período de tempo determinado, uma exceção **TimeoutException** é disparada. A "contagem regressiva" do período de tempo inicia assim que **readXXX()** é invocado e não quando a mensagem é realmente enviada.

- **IsException()**

  Aguarda por uma resposta ("reply") e então retorna **true** se **Result** contém uma exceção.

- **GetException()**

  Aguarda por uma resposta ("reply") e então retorna a exceção em **Result** ou **null** se nenhuma exceção ocorreu.

6.4.2 - Invocação Dinâmica de Mensagens: Mensagens One-Way

Uma mensagem "**one-way**" não retorna um resultado. Quando um "caller" envia uma mensagem deste tipo, o "caller" não bloqueia enquanto a mensagem se completa, acelerando, desta forma, a conclusão do processo. Para se enviar uma mensagem "**one-way**" dinamicamente, utiliza-se o método **OneWay.invoke()**, passando os seguintes parâmetros: "*objeto destino*", "*nome do método que deve ser invocado no objeto destino*" e "*parâmetros para o método num objeto array*". Por exemplo, a linha de código abaixo apresenta o uso de uma mensagem "**one-way**":

```
Result result = OneWay.invoke( market, "buy", newObject[] { new Integer( 42 ), "SUN" } );
```

O objeto representado pela variável "**result**" nunca conterá um valor, e apenas apresentará uma exceção caso ocorra um erro, por parte do cliente, durante a transmissão da mensagem. Por exemplo, se o cliente não puder contatar o servidor remoto durante o envio da mensagem, o resultado será uma exceção **"IOException"**.

6.4.3 - Invocação Dinâmica de Mensagens: Mensagens Future

Uma mensagem "**future**" retorna imediatamente um objeto **Result** ("placeholder" para o valor de retorno). Quando um "caller" envia uma mensagem "**future**", ele não bloqueia enquanto a mensagem se completa. Pode-se utilizar **Result**

para recuperação do valor de retorno a qualquer momento. Para se enviar uma mensagem "**future**", invoca-se o método **Future.invoke()** com os seguinte parâmetros: "*objeto destino*", "*nome do método a ser invocado no objeto destino*" e "*parâmetros para o método em um objeto array*".

O código abaixo exemplifica o uso de mensagens "**future**" invocando uma mensagem **quote()** num objeto **Stockmarket** e lendo, mais tarde, o valor de retorno.

```
Result result = Future.invoke( market, "quote", new Object[] { "SUN" });
// realização de outras operações
int price = result.readInt(); // block for price, if necessary
```

Através do mecanismo padrão "**event/listener**" do Java, é possível obter a notificação sobre a chegada do valor de retorno. Quando um valor de retorno chega, **Result** envia **resultReceived()** com um objeto **ResultEvent** para todos **ResultListener** especificados no escopo do **Future.invoke()** ou adicionados ao objeto **Result** após o envio da mensagem.

Por "default", as mensagens Voyager são assíncronas e não se definem intervalos para que se completem. Tal comportamento, em geral adequado, permite que os programas aguardem o tempo necessário para a conclusão no envio de mensagens. Entretanto, se a rede não "inspira" muita confiança, é razoável atentar para a necessidade de se prever períodos de "tolerância" para que as mensagens se completem. A linha de código abaixo cria um objeto **Result** com um período de "timeout"[3] de 10.000 ms :

```
Result result = Future.invoke( market, "quote", new Object[] { "SUN" } , false, 10000, null );
```

O Voyager também permite que se altere o valor de "timeout" para um **Result** gerado por uma mensagem futura através dos seguintes métodos:

[3] intervalo, tempo

- **setTimeout(long** *timeout* **)**

  Altera o valor de "timeout" para um objeto **Result**. Quando **Result** é lido, a cronometragem do "timeout" inicia.

- **getTimeout()**

  Retorna o valor atual de "timeout" para um objeto **Result**. O valor "default", zero, indica a inexistência de um "timeout".

6.5 - Agregação Dinâmica

A Agregação Dinâmica permite que se anexe e/ou agregue objetos secundários, denominados **facetas**, a um objeto primário em tempo de execução. Um objeto primário e suas facetas formam um **agregado** que é tipicamente "armazenado" (persistência), movido e recolhido pelo "garbage collector" como uma única unidade. O diagrama seguinte (Figura 6.1) ilustra um objeto primário e suas facetas.

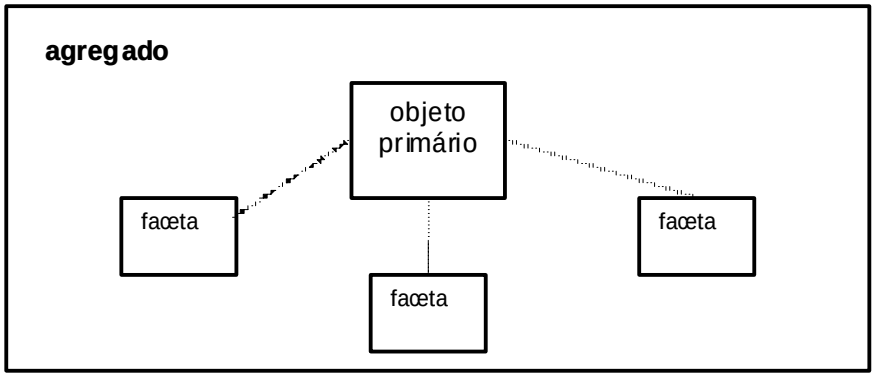

FIGURA 6.1 : Diagrama de representação da Agregação Dinâmica
FONTE: Voyager (1998).

Há várias regras principais associadas às facetas:

a) Não se exige modificação alguma no código de uma classe para que suas instâncias possam representar o papel de uma faceta e/ou de um objeto primário.

b) Não existe a restrição de que a classe de uma faceta esteja relacionada de qualquer forma à classe de um objeto primário. Uma instância de uma classe pode ser adicionada como faceta a qualquer tipo de objeto primário.

c) As facetas não podem ser aninhadas. Em outras palavras, uma faceta não pode ter uma faceta.

d) As facetas não podem ser removidas. Uma vez adicionada, mantém-se relacionada ao agregado durante todo "ciclo de vida" dele.

e) Um objeto primário e suas facetas têm a mesma duração de "vida", sendo "requisitado" e/ou recolhido pelo "garbage collector" quando já não há referência ao objeto primário ou à qualquer uma de suas facetas.

Em especial, ressalta-se a importância dos mecanismos de agregação dinâmica visto que os mesmos são considerados na própria definição do conceito de agente na plataforma Voyager. A faceta **Agent**[4] é agregada dinamicamente a um objeto que passa a utilizar todos os métodos definidos nesta classe. Não menos importante, se destaca a utilização da agregação para se atribuir mobilidade aos objetos definidos no ambiente Voyager.

Além destes destaques, existem muitas outras possibilidades de uso para a agregação dinâmica. Por exemplo, pode se adicionar dinamicamente uma faceta "plano de gratificação" para um objeto ***empregado***, uma faceta "histórico de conserto" para um objeto ***carro***, uma faceta "registro de pagamento" para um objeto ***cliente***, ou uma faceta "hyperlinks" para um objeto genérico.

Será descrito, a seguir, como adicionar e acessar facetas. Para detalhes complementares, o algoritmo para selecionar a implementação de uma faceta e a abordagem para se definir classes "facet-aware" podem ser consultados na referência (Voyager 04).

[4] Classe dentro de **com.objectspace.voyager.agent** que contém métodos para acesso à faceta de um agente

6.5.1 - Acessando e Adicionando Facetas

As facetas de um objeto primário são representadas por uma instância da classe **Facets** que é inicialmente fixada (setada) como **null**. Para acessar as **Facets** de um objeto, deve-se utilizar um dos seguintes métodos estáticos de **Facets**:

- **get(Object** *object* **)**
  Retorna as **Facets** do objeto, podendo ser **null**.
- **of(Object** *object* **)**
  Retorna as **Facets** do objeto, fixando-a a uma instância de **Facets** se atualmente estiver **null** .

Visto que uma faceta é parte de uma agregação, ao se invocar **Facets.get()** ou **Facets.of()** em uma faceta, se obtém como retorno a instância **Facets** do objeto primário. Para manipular as facetas de um objeto, se utiliza os seguintes métodos de instância definidos em **Facets**:

- **get(String** *interfacename* **)**
  Retorna um "proxy" para a faceta que implementa a interface específica, ou **null** se não for encontrada.
- **of(String** *interfacename* **)**
  Retorna um "proxy" para a faceta que implementa a interface específica, adicionando automaticamente outra caso uma interface compatível não seja encontrada. Se o nome da interface começa com "I", a classe sem o prefixo "I" é utilizada como implementação "default".
- **getPrimary( )**
  Retorna um "proxy" do objeto primário.
- **getFacets( )**
  Retorna um "array" de "proxy" das facetas do objeto primário.

Há dois métodos "helper" adicionais em **Facets** que simplificam a manipulação de facetas:

- **get(Object** *object,* **Class** *type* **)**
  Se o objeto especificado é uma instância da interface determinada, retorna o objeto. Se o objeto especificado tem uma faceta que é uma instância da interface especificada, retorna um "proxy" para aquela faceta. Caso contrário, retorna **null** .

- **of(Object** *object,* **Class** *type* **)**

  Se o objeto especificado é uma instância da interface determinada, retorna o objeto. Se o objeto especificado tem uma faceta que é uma instância da interface especificada, retorna um "proxy" para aquela faceta. Caso contrário, adiciona e retorna um "proxy" para uma faceta que implementa o tipo especificado.

Por conveniência, é comum se definir métodos estáticos **get()** e **of()** que realmente simplificam o acesso às facetas. Por exemplo, assumindo-se que o nome da interface da faceta é **IRepairHistory**, é comum se adicionar métodos helper **get()** e **of()** à classe **RepairHistory** que ficaria assim:

```
static public IRepairHistory get( Object object )
 {
  return (IRepairHistory) com.objectspace.voyager.Facets.get( object, IRepairHistory.class );
 }
static public IRepairHistory of( Object object ) throws ClassCastException
 {
  return (IRepairHistory) com.objectspace.voyager.Facets.of( object, IRepairHistory.class );
 }
```

Estes métodos, então, permitiriam que se escrevesse código desta forma:

```
// return the car's repair history facet or null if it doesn't have one
IRepairHistory history1 = RepairHistory.get( car1 );
// return the car's repair history facet, adding one if it doesn't already exist
IRepairHistory history2 = RepairHistory.of( car2 );
```

6.6 – Mobilidade

Dentro da estratégia de desenvolvimento da plataforma Voyager, a mobilidade se caracteriza como destaque de utilidade por, pelo menos, duas razões:

i) Permitir que dois objetos, trocando grande número de mensagens entre si, sejam movidos para uma localização que os aproxime o máximo possível, reduzindo o tráfego de rede e aumentando o "throughput[5]". Uma mensagem local é, no mínimo, 1000 vezes mais rápida se

[5] ritmo de transferência, quantidade de dados que é possível transferir no barramento de dados ou através de um dispositivo por um segundo. Fonte: Babylon Translator On-Line

comparada a uma mensagem, equivalente, remota. Tal técnica é conhecida como "*locality optimization*".

ii) Um programa pode mover objetos para um dispositivo móvel de modo que este programa permaneça com o dispositivo mesmo após sua desconexão da rede.

6.6.1 – Movendo um objeto para uma nova localização

Para mover um objeto para uma nova localização, deve ser usado o método **Mobility.of()** a fim de se obter a faceta de mobilidade do objeto para que, então, se utilize os métodos definidos em **IMobility**:

- **moveTo(String** *url* **)**
  Move para o programa com o URL específica.
- **moveTo(Object** *object* **)**
  Move para o programa que contém o objeto especificado. O objeto normalmente é especificado como uma "proxy".

Por exemplo, o código a seguir cria uma instância da classe **StockMarket** no "host" *//dallas:8000* e então a move para *//tokyo:9000*:

```
IStockMarket market = (IStockMarket) Factory.create ("StockMarket", "//dallas:8000");
market.news ("at first location"); // send message to initial location
Imobility mobility = Mobility.of ( market ); // obtain mobility facet
mobility.moveTo ("//tokyo:9000"); // move the object to a new location
// the last two lines could be written as Mobility.of (market).moveTo ("//tokyo:9000")
market.news ("at second location"); // message is delivered to new location
```

O método **moveTo()** motiva a seguinte sequência de ocorrências:

a) Quaisquer mensagens que o objeto está atualmente processando são completadas e as novas mensagens que chegam para o objeto são suspensas. O código que faz isto só pode detectar chamadas de métodos **syncronized**, assim não se recomenda mover um objeto que poderia estar executando métodos **non-syncronized**.

b) O objeto e todas suas partes não-transientes são copiados para a nova localização através da serialização em Java, ignorando caracteres "pass-by-reference" como **java.rmi.Remote** e **com.objectspace.voyager.IRemote**. Uma exceção é "lançada" caso qualquer parte do objeto não seja serializável ou um erro de rede ocorra. Ao invés de copiar uma parte particular como um objeto, reserva um "proxy" para esta parte.

c) Os novos endereços do objeto e de todas suas partes não-transientes são "conservadas" (cached) na velha localização

d) O objeto "antigo" é destruído.

e) As mensagens suspensas, enviadas ao antigo objeto, são retomadas.

f) Quando uma mensagem enviada via "proxy" chega ao antigo endereço de um objeto movido, uma exceção ("exception") especial contendo o novo endereço do objeto é repassada para o "proxy antigo". O "proxy" apanha esta exceção, efetua um "rebind" para o novo endereço, e então reenvia a mensagem para o endereço atualizado. Se o programa na antiga localização "cai" (crash) antes que o "proxy antigo" seja atualizado, o proxy "ultrapassado" não terá a capacidade de, com sucesso, efetuar o "rebind" e uma mensagem enviada pelo "proxy" gerará uma exceção **ObjectNotFoundException**.

g) O método **moveTo()** retorna depois que o objeto é movido com sucesso ou quando uma exceção de mobilidade ocorre. Se uma exceção acontece, o antigo objeto é restabelecido à sua condição original, as mensagens suspensas são reconsideradas, e a exceção é repassada (rethrown) dentro de uma exceção **MobilityException**.

As regras do "garbabe collector" não são afetadas pela "mobilidade" - um objeto movido é "requisitado" quando não há nenhuma referência local ou remota para ele. Repara-se que os novos endereços "conservados" (cached) na antiga localização não são tratados como referências pelo sistema de recolhimento de "lixo" (garbage collector). Não é seguro mover um objeto quando existem referências Java locais apontadas para o mesmo a partir de um contexto fora do Voyager ou se o objeto tem um ou mais "threads" não associados a uma mensagem remota.

6.6.2 - Obtendo Notificação de Movimento

Às vezes um objeto "precisa saber" que está prestes a se mover ou se foi movido há pouco tempo. Por exemplo, pode se considerar a necessidade de um objeto móvel persistente remover-se de um "repositório-origem" para armazenar-se em um "repositório-destino" em outro "host". O Voyager provê tal capacidade através da

interface **IMobile**. Se um objeto ou qualquer uma de suas partes implementam a interface **IMobile**, receberá "callbacks" durante um movimento na seguinte ordem:

a) **preDeparture(String** *source***, String** *destination* **)**

Este método é executado no objeto original antes de partir. Se o método dispara uma exceção **MobilityException**, o movimento é abortado e nenhum outro "callback" de **IMobile** ocorre.

b) **preArrival()**

Este método é executado na cópia do objeto quando chega ao destino. Se o método dispara alguma exceção **MobilityException**, o movimento é abortado e nenhum outro "callback" de **IMobile** ocorre.

c) **postArrival()**

Neste momento, a cópia do objeto se tornou em objeto "real / atual", o objeto no "host" origem se tornou o objeto "stale" (antigo), e considera-se que a fase de movimento obteve sucesso e não pode ser abortado. O método **postArrival()** é executado imediatamente na cópia de objeto no destino imediatamente anterior ao "callback" do usuário, e é definido para executar atividades típicas tais como adicionar o novo objeto em um armazenamento persistente.

d) **postDeparture()**

Este método é executado no objeto "antigo" no "host" origem, e é tipicamente definido para executar atividades como remover o objeto de persistência. As mensagens enviadas ao objeto "antigo" serão redirecionadas via "proxy" ao novo objeto, de modo que o método **postDeparture()** não deveria utilizar "proxy's" para o objeto original ou para qualquer uma de suas facetas. Note que o fato de o "callback" de um usuário para o novo objeto ser executado com um novo "thread", é possível que este método (**postDeparture()**) seja executado concorrentemente.

6.7 - Agentes Móveis

6.7.1 – Utilização de Agentes Móveis

Um agente móvel e autônomo[6] é um objeto que se move através de uma rede a fim de alcançar suas metas, e, por assim dizer, útil em variadas situações:

1) Se uma tarefa deve ser executada independentemente do computador que dispara uma certa tarefa, um agente móvel pode ser criado para executar esta "missão". Uma vez

[6] segundo a definição da plataforma Voyager (Voyager 04)

construído, o agente pode se mover pela rede e completar a tarefa em um programa remoto.

2) Se um programa lida com o envio de um número grande de mensagens a objetos em programas remotos, um agente pode ser criado para visitar cada programa enviando as mensagens localmente. Mensagens locais são, freqüentemente, entre 1.000 e 100.000 vezes mais rápidas em relação às mensagens remotas.

3) Se existe a intenção de se partilhar os programas / tarefas a fim de executá-los em paralelo, pode-se distribuir o processamento entre vários agentes que migram para programas remotos e se comunicam para alcançar uma meta em comum.

4) Se a monitoração periódica de um objeto remoto é requerida, cria-se um agente que se move para o objeto remoto realizando a monitoração localmente.

5) Se uma série de operações deve ser executada em um dispositivo ocasionalmente conectado a uma rede como, por exemplo, um telefone Java ou pager Java, delega-se um agente para acessar tal dispositivo, executar sua tarefa, e retornar à rede quando necessário.

Deve-se evitar o "ajuste à força" da tecnologia de agente em um programa. As mensagens remotas do Voyager são adequadas para muitas aplicações, e a "simples" mobilidade de um objeto é, com razoável freqüência, suficiente para se transpor o "abismo" entre dois objetos que comunicam através de uma rede. Porém, à medida em que ocorre a "familiarização" com o "poder de fogo" dos agentes, pode-se encontrar muitas maneiras apropriadas para incrementar os programas atuais e futuros com a tecnologia em questão

6.7.2 - Criando Agentes Móveis

Para que um objeto se "transforme" em um agente móvel e autônomo, deve-se utilizar o método **Agent.of()** através do qual se obtém a faceta de agente propiciando, assim, o uso dos métodos definidos em **IAgent**:

a) moveTo(String *url* , String *callback* [, Object [] *args*])

Mover-se para programa com o URL especificado e então reiniciar executando uma chamada de método "oneway" (callback) com argumentos opcionais. Uma exceção **MobilityException** é disparada se o método definido na "invocação" não é encontrado ou não é público.

b) moveTo(Object *object*, String *callback* [, Object [] *args*])

Mover-se para o programa contendo o objeto especificado e então reiniciar executando uma chamada de método "oneway" (callback) com um "proxy" para o objeto como primeiro argumento e argumentos opcionais como argumentos restantes. Uma exceção **MobilityException** é lançada se o método da "invocação" não é encontrado ou não é público.

c) setAutonomous(boolean *flag*)

Se o flag é verdadeiro, torna-se autônomo. Um agente autônomo não é requisitado pelo "garbage collector" ainda que não existam quaisquer referências, locais ou remotas, para ele. Um agente, por default, é inicialmente definido como autônomo, e geralmente executa-se o método **setAutonomous (false)** quando o mesmo alcançou sua meta e pretende colocar-se à disposição do "garbage collector".

d) isAutonomous()

Retorna verdadeiro se este agente é autônomo.

e) getHome()

Volta para a "casa" deste agente que é definida como a URL do agente quando a "faceta de agente" foi inicialmente acessada.

Por exemplo, um objeto pode se mover para o host **//dallas:8000** e reiniciar usando o método **atDallas ()** da seguinte forma:

```
Agent.of( this).moveTo ( "//dallas:8000", "atDallas" );
```

Uma chamada com sucesso ao método **moveTo()** conceitualmente determina que o "thread" de controle no agente seja interrompido no momento que se antecipa ao movimento para ser retomado a partir da chamada do método de "callback" no agente depois da efetivação do movimento. Desta forma, deve-se atentar para que se inclua após o método **moveTo()** somente código para tratamento de exceções.

O exemplo **Agents1 (**Voyager 04) demonstra o uso de agentes móveis e autônomos. Um objeto “agrega dinamicamente” a faceta de agente. Isto permite que o objeto se mova através da rede. Quando o agente completa sua “missão”, sua autonomia é desativada, permitindo que o agente seja recolhido pelo “garbage collector”.

Pôde ser comprovado, através de testes com o exemplo fornecido, que a velocidade na troca de mensagens melhora consideravelmente a partir do momento em que o agente interage localmente com um objeto. O código das classes pertinentes ao exemplo - **ITrader.java, Trader.java, Agents1.Java, IStockmarket.java e Stockmarket.java -** é submetido, no Capítulo 8 referente à implementação do protótipo, a uma análise minuciosa para melhor compreensão do cenário acima descrito pela Figura 6.2.

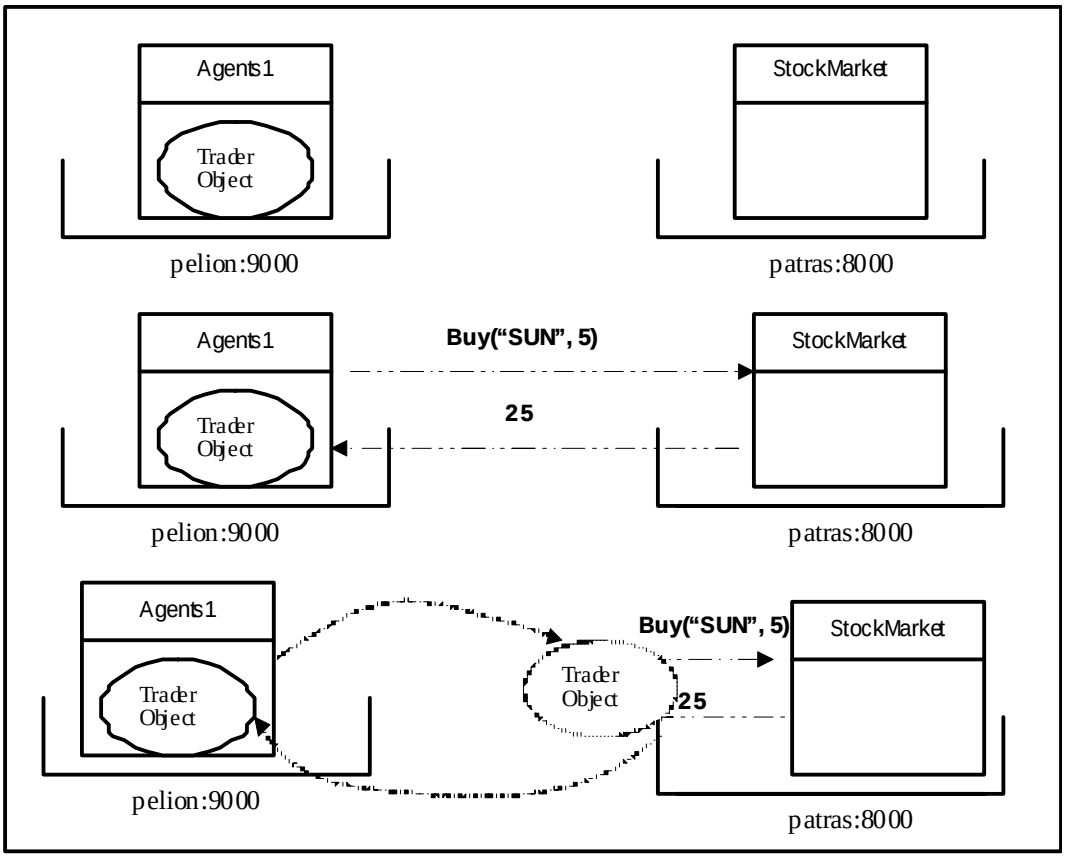

FIGURA 6.2: Representação do exemplo Agents1
FONTE: Vovager (1998).

6.8 - Naming Service

O serviço de nomes permite que se associe um nome a um objeto para uma futura referência a este objeto. Os seguintes tópicos são abordados:

6.8.1 - Utilizando o Namespace

Existem diferentes implementações de “naming service” entre os quais se destacam: Voyager Federated Directory Service, CORBA Naming Service, JNDI , Microsoft Active Directory e RMI registry.

Quando se utiliza um serviço de nomes para associar um nome a um objeto, adiciona-se um prefixo único de modo que o o tipo de serviço possa, mais tarde, ser diretamente determinado a partir do nome. Por exemplo, o serviço de diretório confederados do Voyager utiliza o prefixo **vdir:** e o CORBA adota o prefixo **IOR:**.

A classe **Namespace** do Voyager aproveita as vantagens destes prefixos fornecendo uma interface simples que unifica o acesso para um ou mais destes serviços. Novos serviços podem ser “plugados” e/ou inseridos em **Namespace**.

Cada um dos seguintes métodos utiliza o prefixo do nome para determinar qual “infraestrutura de naming service” acessar.

a) **lookup(String** *name***)**

   Retorna um “proxy” para o objeto associado ao nome, ou **null** no caso de nenhum objeto ser encontrado.

b) **bind(String** *name***, Objetc** *object***)**

   Associa o nome especificado com o objeto, ou lança uma exceção se o nome já possui uma associação.

c) **rebind(String** *name***, Object** *object***)**

   Associa o nome especificado com o objeto substituindo qualquer associação prévia existente.

d) **unbind(String** *name***)**

   Desfaz a associação para o nome especificado.

6.8.2 - Trabalhando com o Serviço de Diretórios Federados

O "Voyager Federated Directory Service" permite o registro de um objeto em uma estrutura de diretório hierarquicamente distribuída. Pode-se associar objetos a nomes de "paths" compostos por "strings" separadas por barras ("/") tal como **fruit/citrus/lemon** ou **animal/mammal/cat**. As "peças de montagem" do serviço de diretório baseiam-se na classe **Directory** que apresenta a seguinte interface:

a) put(String *key*, Object *value*)

Associa uma chave a um valor. Se a chave é uma string, associa a mesma com o valor especificado no diretório local. Se a chave é um path, procura o **Directory** associado com a "raiz" do path e então envia uma mensagem **put()** com o restante do path. Retorna o valor associado previamente com a chave ou **null** se não existir.

b) get(String *key*)

Retorna o valor associado à chave. Se a chave é uma string, retorna o valor associado no diretório local ou null se não existir. Se a chave é um path, procura o **Directory** associado com a "raiz" do path e então envia uma mensagem **get()** com o restante do path.

c) remove(String *key*)

Remove a "entrada" de diretório com a chave especificada. Se chave é uma string, remova sua "entrada" do diretório local. Se chave é um path, procure o **Directory** associado com a "raiz" do nome do path e então envia uma mensagem **remove()** para o restante do path. Retorna o valor associado à chave, ou nulo se não existir.

d) getValues( )

Retorna um array de valores no diretório local.

e) getKeys( )

Retorna um array das chaves no diretório local.

f) clear( )

Remove todas "entradas" do diretório local.

g) size( )

Retorna o número de chaves no **Directory** local.

Para criar um simples diretório de objetos locais, defina um objeto **Directory** e invoque a mensagem **put()** com uma chave e um objeto local.

```
Directory symbols = new Directory();
symbols.put( "CA", "calcium" );
```

```
symbols.put( “AU”, “gold” );
// symbols.get( “CA” ) would return “calcium
```

Para criar uma estrutura “encadeada”de diretórios – um **Directory** que se refere a outro **Directory** – invoque a mensagem **put()** para um objeto **Directory** com outro diretório ou um proxy para um **Directory** remoto como segundo parâmetro.

```
Directory root = new Directory();
root.puts( “symbols”, symbols ); //associate “symbols” with the symbols directory
// root.get( “symbols/CA” ) would return “calcium”
```

Visto que a classe **Directory** implementa a interface **IRemote**, permite-se a passagem de um diretório local como parâmetro para um diretório remoto.

6.8.3 – Utilizando o Naming Service “default”

O serviço de diretórios federados do Voyager é o serviço de nomes “default” explorado pela classe **Namespace**. Pode-se utilizar **Namespace** para operações, com objetos remotos, de “*bind*”, “*rebind*”, e “*unbind*” sem mesmo acessar um objeto **Directory**.

Quando um programa Voyager inicializa, automaticamente exporta um objeto **Directory** para uso do **Namespace**. Ao executar uma operação **lookup()** no **Namespace,** caso não exista um prefixo para o nome, **Namespace** interpreta o nome como uma **URL**, obtém um “proxy” para o **Directory Namespace** no programa correspondente, e então executa um **remote()** no **Directory** com o restante da **URL** como chave. Por exemplo:

Namespace Format	Directory Equivalent
Namespace.lookup(“Fred”)	<Directory @ local program>.get(“Fred”)
Namespace.lookup(“8000/Fred”)	<Directory @ //localhost:8000>.get(“Fred”)
Namespace.lookup(“//dallas:8000/Fred”)	<Directory @ //dallas:8000>.get(“Fred”)
Namespace.lookup(“//dallas:8000/Fred/Bloggs”)	<Directory @ //dallas:8000>.get(“Fred/Bloggs”)

Os métodos **bind()**, **rebind()** e **unbind()** são processados de maneira similar:

Namespace Format	Directory Equivalent
Namespace.bind(“Fred”, object)	<Directory @ local program>.put(“Fred”, object)
Namespace.lookup(“8000/Fred”)	<Directory @ //localhost:8000>.remove(“Fred/Bloggs”)

Por conveniência, o método **Factory.create()** encontra-se integrado com **Namespace**. Se a localização de um “host” é seguida por um nome, este é utilizado automaticamente na execução de um **bind()** no **Namespace** no “host” considerado. Por exemplo, ao invés de utilizar a sintaxe:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000" );
Namespace.bind( "//dallas:8000/NASDAQ", market );
```

pode-se obter os mesmos resultados, com a sintaxe:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000/NASDAQ"
);
```

6.9 - Integração Voyager-CORBA

Uma das maiores “forças” da tecnologia Voyager é sua natural ligação (binding) com a linguagem Java. Qualquer classe Java pode ser “remote-enabled” sem qualquer modificação e os objetos remotos se comportam de maneira intuitiva para os desenvolvedores da linguagem.

A ligação CORBA-Java (binding - ligação entre o protocolo de comunicação e o adaptador de rede) definida pela OMG não apresenta as mesmas propriedades. O objetivo da ObjectSpace não é simplesmente tornar o Voyager um ORB CORBA, visto que o Voyager é mais “poderoso” e cômodo para utilização no desenvolvimento distribuído Java quando comparado às tecnologias CORBA. É crucialmente importante que os desenvolvedores Voyager estejam aptos para acessar objetos CORBA e objetos DCOM de forma descomplicada e simples preservando a superioridade, elegância e facilidade de uso do “binding” Java natural encontrado no Voyager. Combinado à poderosa arquitetura base Java, o Voyager fornece, em diferentes formas, uma melhor

solução para “CORBA-enabling” um programa Java quando comparada às outras opções disponíveis atualmente.

Os destaques da integração Voyager-CORBA são descritos abaixo:

a) Voyager pode criar a interface Java e classes para referência virtual Voyager equivalentes a partir de qualquer arquivo IDL. Pode, também “tomar” qualquer interface Java ou arquivo “***.class”*** e criar automaticamente o arquivo IDL equivalente, permitindo que a operação “CORBA-enabling” ocorra sem modificações.

b) Voyager pode obter um referência virtual para um objeto CORBA, e o CORBA pode obter uma referência remota para um objeto Voyager.

c) Um programador Voyager pode enviar mensagens para um objeto através de referências virtuais mesmo desconhecendo se o mesmo é um objeto CORBA ou um objeto Voyager. Se objeto está sobre um ORB CORBA, o Voyager automaticamente utiliza o protocolo IIOP para se comunicar com o objeto CORBA.

d) As referências virtuais para os objetos Voyager são convertidas automaticamente para referências CORBA quando enviadas para um ORB CORBA, e as referências CORBA são convertidas automaticamente para referências virtuais quando enviadas para o Voyager.

e) Voyager suporta parâmetros in, inout e out através do mecanismo proprietário padrão.

f) Visto que as referências virtuais ocultam os detalhes do ORB, os serviços avançados (tais como Space™, multicast messaging, invocação futura e dinâmica) podem ser utilizados com objetos CORBA, objetos Voyager ou com objetos “combinados” (CORBA + Voyager).

g) O “módulo” para integração com o CORBA adiciona aproximadamente 100KB ao tamanho total do Voyager.

Para uma descrição detalhada sobre os tópicos da integração das tecnologias CORBA-Voyager, incluindo vários exemplos práticos (com código fonte) para utilização, ver (Voyager 03) e (Voyager 04).

6.10 - Multicast e Publish/Subscribe na Arquitetura Space

Sistemas distribuídos requerem características para comunicação entre grupos de objetos. Os exemplos abaixo se encaixam a tais considerações:

a) Os sistemas de cotação de ações utilizam eventos distribuídos para enviar eventos "preço de ações" para os clientes espalhados pelo mundo.

b) Os sistemas de votação utilizam a vantagem da transferência distribuída de mensagens (multicast) para consultar as intenções dos eleitores ou votantes localizados em diferentes pontos de um determinado domínio.

c) Agências de notícias utilizam mecanismos distribuídos publish/subscribe no envio de eventos-notícia apenas para os leitores (assinantes) interessados no tópico transmitido.

A maioria dos sistemas tradicionais adota um único objeto repetidor – repeater – para replicar uma mensagem ou evento para cada objeto que compõe um grupo específico. Tal abordagem, satisfatória e suficiente quando o número de objetos dentro de um sistema é relativamente pequeno, não pode produzir resultados interessantes quando este número aumenta consideravelmente. O Voyager faz uso de uma arquitetura inovativa e "escalável", denominada **Space**, para a replicação de mensagens / eventos. A arquitetura **Space** define-se como um "conglomerado" distribuído de objetos e pode abranger / amarrar múltiplos programas. Um domínio **Space** é criado conectando-se um ou mais **Subspace** (um recipiente local de objetos).

Uma mensagem e/ou evento , enviado(a) via "proxy multicast" dentro de um **Subspace**, é "clonado(a)" para cada **Subspace** vizinho antes de ser despachado(a) para todos objetos no **Subspace** local, resultando numa propagação rápida e paralela de mensagens para todos objetos no **Space**. Enquanto são disseminadas, as mensagens deixam "marcas" únicas que, por um período de cinco minutos, são recordadas pelo **Subspace**. Se o "clone" de uma mensagem entra, novamente, em um **Subspace**, este primeiro detecta a "marca" e se auto destrói. Tal sinalização por marcas permite que **Subspaces** sejam conectados formando topologias arbitrárias sem a possibilidade de múltiplo despacho de mensagem. A tolerância a falhas (devido falhas individuais de rede) aumenta em proporção direta ao "grau" de inter-conexão dos **Subspaces**.

O diagrama abaixo (Figura 6.3) ilustra o envio de uma mensagem para um **Subspace** que faz parte de um **Space**.

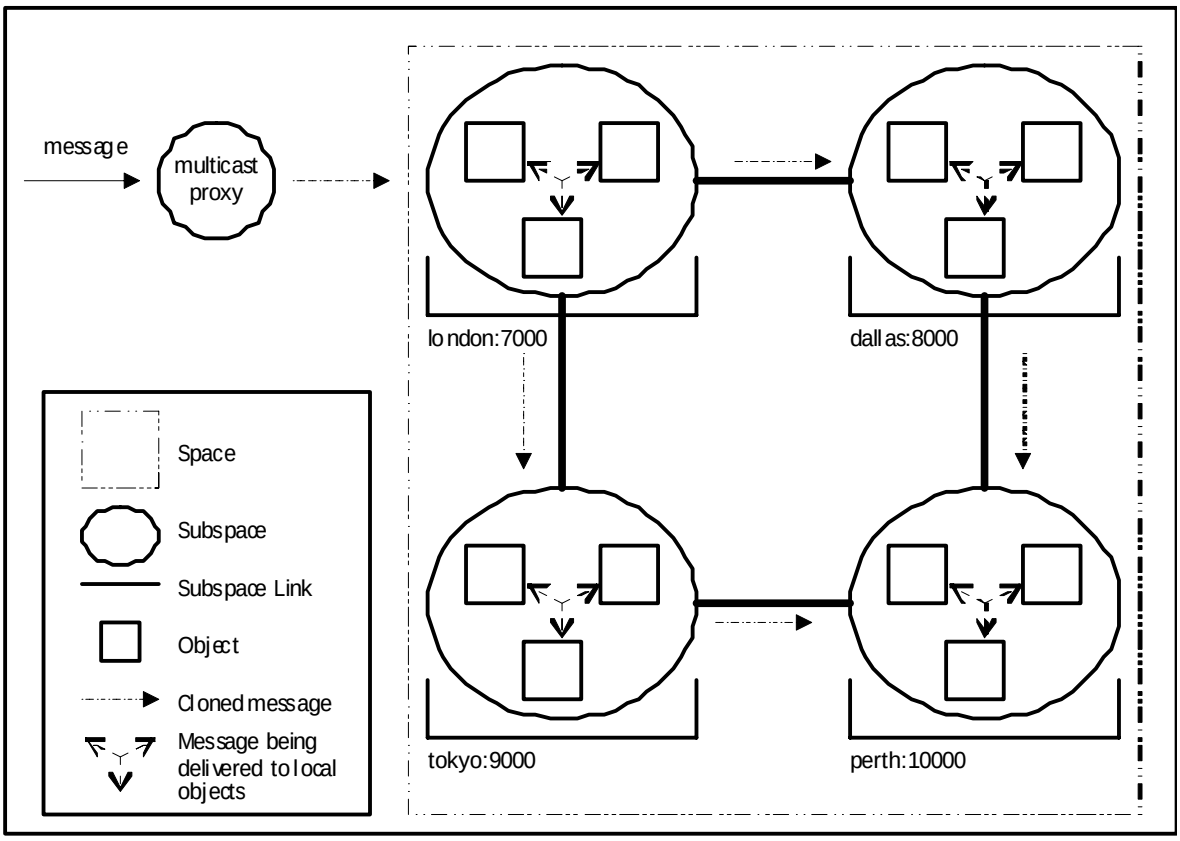

FIGURA 6.3: "Disparo" de uma mensagem dentro da Arquitetura Space
FONTE: Voyager (1998).

Para melhores detalhes da arquitetura **Space**, ver referência (Voyager 1998) – Capítulo 5.

CAPÍTULO 7
O SERVIÇO DE AGENTES

7.1 - Introdução

A linha mestra deste capítulo, o principal numa escala hierárquica de relevância para o trabalho, se orienta rumo à compilação de toda a carga teórica, arrolada nos capítulos anteriores, convergindo para a concepção e/ou composição do Serviço de Agentes.

Como apresentado no Capítulo 2, o software de controle de satélites passa por uma "reestruturação tecnológica" apoiada na proposição da arquitetura do ambiente SICSD. A ênfase maior se encontra na transição de um modelo cliente-servidor tradicional (estático) para uma arquitetura distribuída e dinâmica na qual as aplicações são organizadas como objetos móveis[1].

Entre outros apresentados, o serviço para monitoração destes objetos - Serviço de Agentes - não surge como proposta isolada e sem propósito específico e, como participante de uma pesquisa mais abrangente, compartilha os benefícios e contribuições almejados pela proposta SICSD além de apresentar um perfil sintonizado e apropriado à metodologia, difundida mundialmente, das soluções para software distribuído e dinâmico baseado em agentes móveis.

O principal alerta a se destacar se relaciona ao fato de que a contribuição e o caráter inédito do trabalho não se resumem a uma escolha aleatória da tecnologia de agentes móveis como nova ou melhor metodologia para a "remodelagem" proposta para o software de controle de satélites. Muito além desta falsa impressão, o retorno mensurado através destas duas variáveis assinaladas anteriormente deve contabilizar os aspectos e benefícios já explicitados na apresentação da arquitetura SICSD.

[1]objetos que poderão migrar dinamicamente de uma máquina para outra (ver cap. 2)

Ou seja, a concepção do serviço de monitoração de objetos contribui, de imediato, para a efetivação do sistema caracterizado por vantagens[2] como: concorrência, aumento da confiabilidade, flexibilidade, balanceamento de carga e disponibilidade de serviços. Além disso, a concepção do Serviço de Agentes presume: a) análise e integração de infra-estrutura de comunicação para objetos distribuídos baseada no padrão CORBA; b) assimilação da funcionalidade das plataformas de agentes móveis Voyager e Grasshopper; c) compatibilidade à linguagem Java e aos padrões MASIF e FIPA; d) definição de modelo integrando os conceitos de um ambiente distribuído de agentes (ADA) e de um ambiente de processamento distribuído (APD); e) definição e detalhes do modelo para o Serviço de Agentes; f) aplicação prática do Serviço de Agentes através da disponibilização de um ambiente de teste.

Antecipando-se aos maiores detalhes quanto à sua definição, o Serviço de Agentes compreende a integração de três "***módulos básicos***": os Agentes, a Plataforma de Agentes (ambiente de execução para os agentes) e o Serviço-aplicação (monitoração de objetos). Na verdade, portanto, a grande contribuição inédita do Serviço de Agentes pode ser resumida na composição de um "ambiente" distribuído de agentes, os quais, atuando e/ou executando em plataformas apropriadas, conferem recursos diferenciados de mobilidade e autonomia à implementação do serviço de monitoração de objetos dentro do sistema SICSD com o qual interagem. À bordo da proposta do Serviço de Agentes, a peculiaridade do fator inédito é reforçada pelo novo modelo SICSD para o software de controle de satélites, pelo serviço de monitoração original adaptado a este contexto e pela integração e utilização da metodologia baseada em agentes.

7.2 – Requisitos

Segundo as afirmações de Joseph P. Bigus, em um recente livro de sua autoria, o primeiro passo em qualquer projeto de pesquisa e/ou desenvolvimento de software é a coleção de requisitos da comunidade de usuários à qual se destina (Bigus 1998). Tal comunidade, para o contexto do atual trabalho, compreende prioritariamente os limites e

[2] vantagens previstas para o tipo de sistema baseado na arquitetura SICSD

objetivos definidos no capítulo de motivação dentro do qual foi apresentado a proposta SICSD.

Ao mesmo tempo, porém, deve ser evidenciado que a elaboração da lista de requisitos para a "modelagem" do Serviço de Agentes, delineada abaixo, submete-se em concordância com os diversos tópicos destacados, quanto aos "perigos no desenvolvimento orientado a agentes", no capítulo referente à tecnologia de agentes.

Logo, entre os requisitos para o Serviço de Agentes, firmam-se:

a) A solução proposta para o modelo do Serviço de Agentes nasce e se estabelece dentro das definições de um protótipo e, como tal, não sugere a conclusão de um sistema acabado, robusto e confiável para utilização imediata no departamento.

b) A solução proposta para o modelo do Serviço de Agentes deve considerar a exploração e integração dos padrões e tecnologias correlatas (padrões MASIF e FIPA, tecnologia de objetos distribuídos - CORBA, plataformas de agentes móveis, Java, etc).

c) O modelo deve ser flexível o bastante para suportar um considerável domínio de aplicações ou serviços, restringindo-se sua utilização, no momento, ao estudo conFigurado pela proposta SICSD.

d) Não se considera, a priori, a definição de uma interface gráfica entre o usuário e o Serviço de Agentes, visto que interação básica para o usuário fica sobre a responsabilidade de um módulo "superior" definido na proposta SICSD. Em suma, o Serviço de Agentes deve abstrair o conceito de uma "caixa preta".

e) A aplicação básica disponibilizada pelo Serviço de Agentes deve: 1) trabalhar com grupos de objetos em diversas máquinas, obter informações a respeito de cada objeto e/ou máquina e enviar mensagens para cada objeto e/ou máquina; 2)

trabalhar com um grupo de objetos em apenas uma máquina, obter informações de cada objeto, criar e/ou deletar processos; 3) permitir uma visão global quanto às informações sobre os objetos a partir de qualquer máquina.

f) Além da aplicação básica apresentada, o Serviço de Agentes deve considerar a possibilidade de se disponibilizar aplicações de gerenciamento de serviços, objetos e/ou rede, apoio à decisão de algoritmos de migração e mobilidade.

7.3 - A Integração de Tecnologias para a definição do Serviço de Agentes

Não emergindo genericamente como solução isolada para inovação no desenvolvimento de software, a tecnologia de agentes prevê a aplicação integrada de múltiplas tecnologias (OMG 2000).

Sob a consideração de tal fato, a fundamentação da arquitetura do Serviço de Agentes não deve desprezar um estudo prévio quanto à combinação e/ou integração de áreas correlatas, cujos conceitos e padrões foram investigados e reunidos nos próximos quatro sub-tópicos.

Em primeiro lugar, define-se um modelo inicial para uma aplicação baseada em agentes a partir da arquitetura Mobile Agent Facility (Bieszczad 1998), introduzindo-se, a seguir, um modelo e considerações quanto à integração da tecnologia de objetos distribuídos (através de um Ambiente de Processamento Distribuído - APD) e da tecnologia de agentes móveis (através de um Ambiente Distribuído de Agentes - ADA). O terceiro tópico traça a especificação OMG-MASIF e do padrão CORBA (middleware). Por último, reitera-se alguns argumentos quanto ao Java através dos quais se justifica a opção por uma plataforma de agentes baseada nesta linguagem.

7.3.1 - A arquitetura OMG-MASIF para o Serviço de Agentes

Este primeiro tópico sugere um modelo baseado no padrão MASIF, originalmente introduzido como “Mobile Agent Facility” (MAF - especificado pela OMG) (MASIF), a partir do qual se define a estrutura inicial para o Serviço de Agentes como apresentado na Figura 7.1 abaixo.

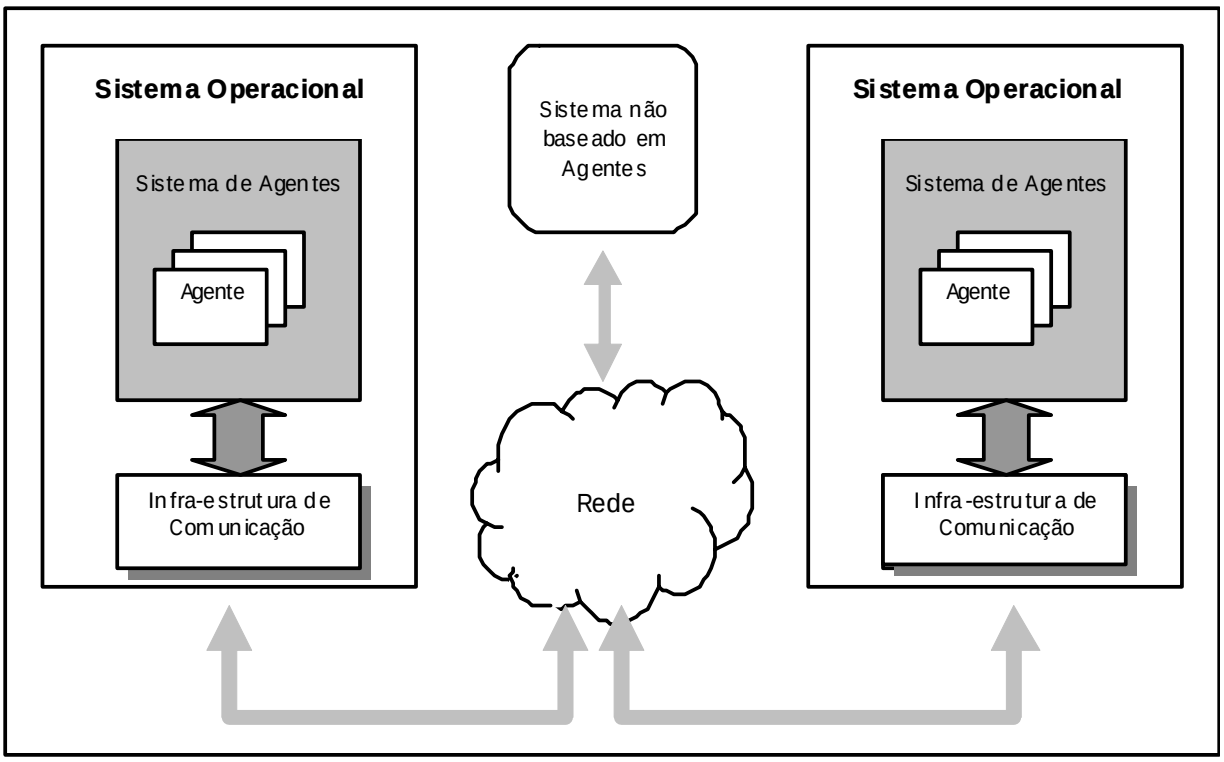

FIGURA 7.1: Arquitetura “Mobile Agent Facility” para o Serviço de Agentes
FONTE: Bieszczad (1998) .

A arquitetura representada acima introduz um primeiro referencial sobre os componentes básicos e o conceito de uma “região de agentes”. Esta região estabelece um domínio onde os agentes, “criados” e “gerenciados” sob a tutela de sistemas de agentes, dispõem de uma infra-estrutura de comunicação por meio da qual migram pela rede possibilitando a realização autônoma de tarefas através da interação com sistemas não baseados em agentes.

7.3.2 - Integração Objetos Distribuídos / Tecnologia de Agentes Móveis

A proposta do sistema SICSD lança seus fundamentos sobre a tecnologia de objetos distribuídos (DOT) e, não fugindo à intenção de se definir resumidamente tais conceitos, indica-se uma referência à recente dissertação de pesquisa (Samira 2000) a qual, avaliando o panorama "mundial" da tecnologia de distribuição de objetos, contribuiu para o estabelecimento de diretrizes. Atualmente o padrão CORBA vem contabilizando alta aceitação no mundo da computação distribuída. Várias especificações de serviços têm sido desenvolvidas, provendo um protocolo comum e interfaces padronizadas independentes de plataforma rumo à interoperabilidade entre aplicações de diferentes fabricantes (Magedanz 1998). Em suma, o padrão CORBA, definindo uma arquitetura que suporta a interoperabilidade de objetos num ambiente distribuído e heterogêneo, permite o acesso transparente aos objetos localizados em diferentes pontos da rede.

O Serviço de Agentes, a bordo dos benefícios já previstos para a tecnologia de agentes, busca complementar e incrementar a funcionalidade da tecnologia de objetos distribuídos resultando em um "middleware" que possa: a) reduzir os requisitos relacionados à carga de tráfego e à disponibilidade da rede de computadores através de operações autônomas e assíncronas dos agentes; b) diminuir o tempo e esforço dispensados na operação de serviços para controle e monitoração de recursos e objetos distribuídos; c) estar apto para provisão "on demand" de serviços customizados através da delegação dinâmica de agentes um sistema-fornecedor para um sistema-cliente ou diretamente para os recursos e/ou objetos do sistema; d) permitir a implementação descentralizada de aplicações para gerenciamento e/ou controle de serviços, designando agentes específicos que atuem o mais próximo possível dos recursos e/ou objetos do sistema.

A Figura 7.2 abaixo representa a abordagem integrada entre as tecnologias DOT e MAT. O maior resultado desta união prevê a combinação das vantagens do CORBA e dos agentes móveis nos moldes de um ambiente distribuído de agentes (definido como ADA), construído no topo de um ambiente de processamento distribuído (definido como APD)

utilizado como canal de comunicação entre as diversas agências (definidas oportunamente) que compõem o ADA.

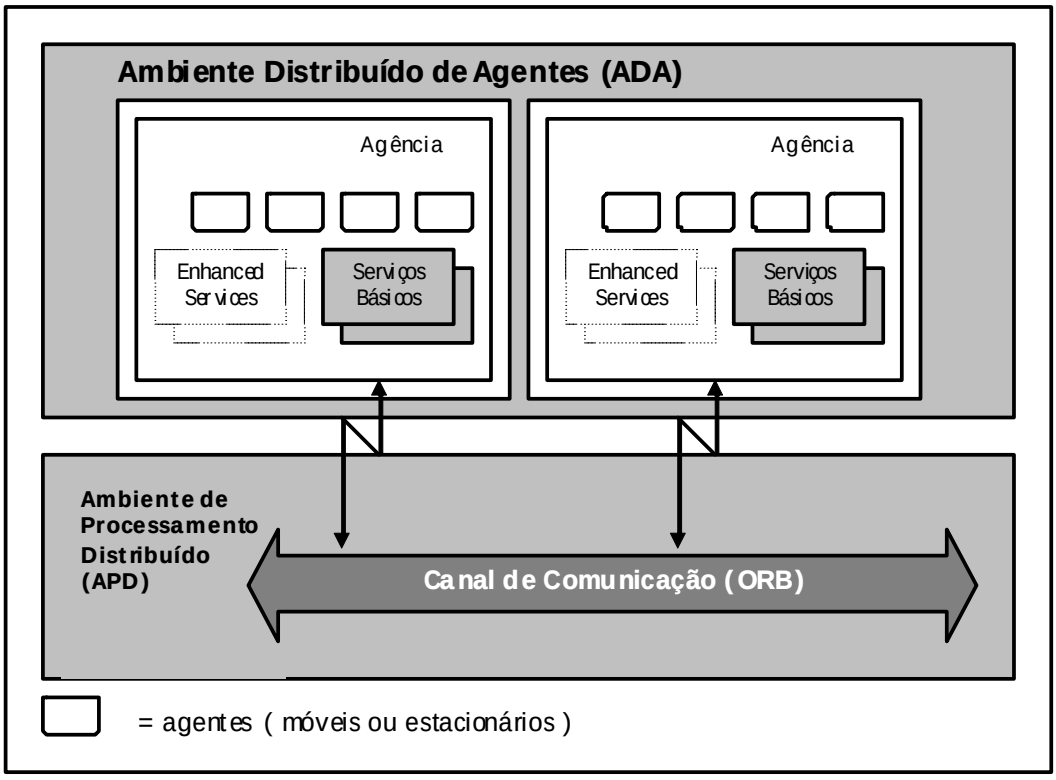

FIGURA 7.2: Arquitetura integrada DOT/MAT para o Serviço de Agentes
FONTE: Magedanz (1998) .

7.3.3 - A Especificação OMG-MASIF para o Serviço de Agentes

Em contraste ao grau de interoperabilidade focalizado pelo padrão CORBA, a tecnologia de agentes móveis "se perde" em meio a imensa variedade de diferentes e incompatíveis abordagens quanto à linguagens de implementação, protocolos, funcionalidades e arquiteturas de plataformas de agentes (Magedanz 1998). A fim de alcançar a integração suficiente com o CORBA, o padrão para as plataformas de agentes móveis foi especificado pela OMG a partir da Mobile Agent Facility (MAF) resultando na especificação OMG-MASIF (MASIF). A idéia central por trás do padrão MASIF é a obtenção de um certo grau de interoperabilidade para as plataformas de agentes móveis de diferentes fabricantes sem acarretar em mudanças radicais da plataforma. A intenção da

OMG-MASIF não se traduz na construção de fundamentos para novas plataformas de agentes móveis. Pelo contrário, as especificações fornecidas são utilizadas como “módulos extensores” para os sistemas existentes.

A Figura 7.3 abaixo apresenta um esboço da arquitetura OMG-MASIF, a partir da qual se definirão, entre outros, os conceitos de região, agência e “place”, oportunamente detalhados à frente na definição do Serviço de Agentes. Em suma, um “place” agrupará a funcionalidade dentro de uma agência, encapsulando certas capacidades e restrições para os agentes que ali atuam. Uma região facilita o gerenciamento da plataforma através da especificação de conjuntos de agências que pertencem a um único domínio.

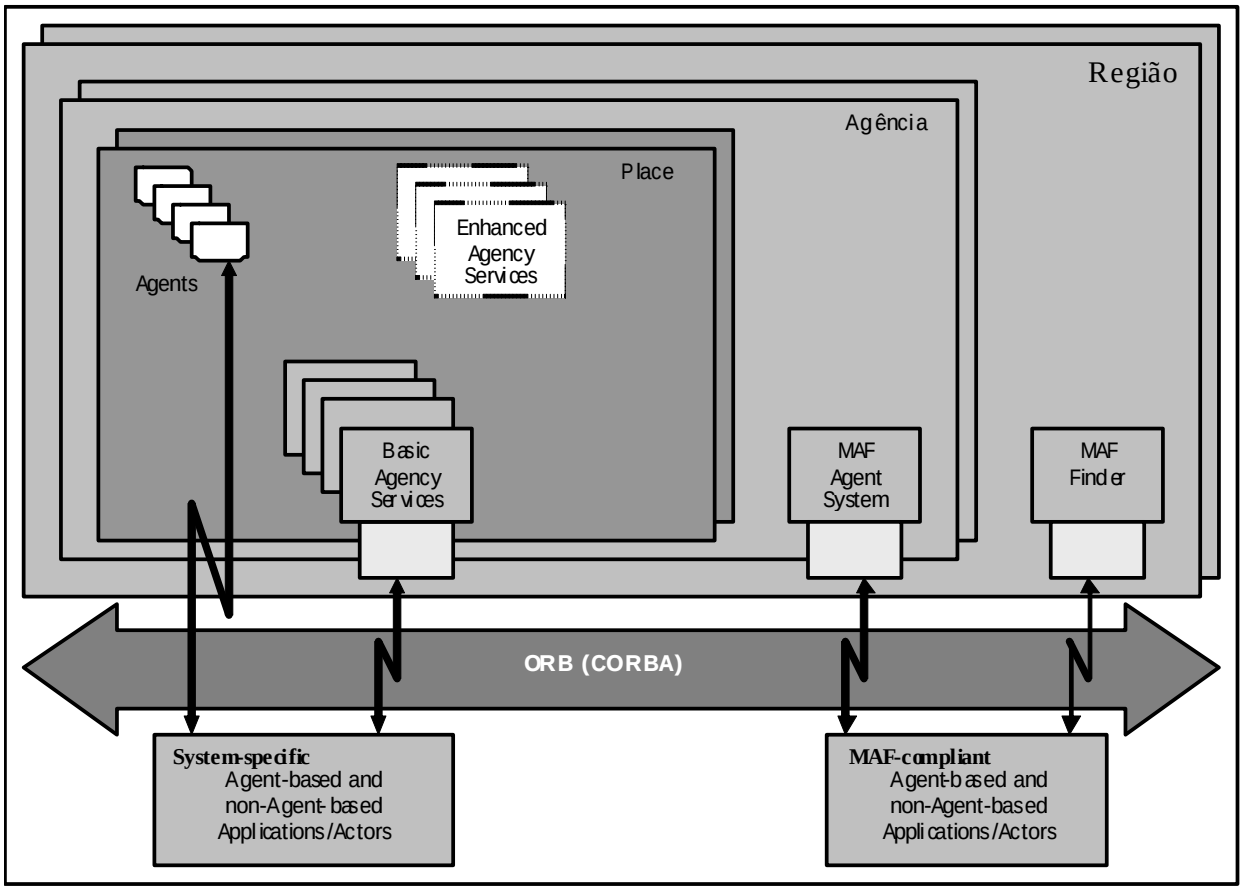

FIGURA 7.3: Arquitetura OMG-MASIF para o Serviço de Agentes
FONTE: OMG (1997).

7.3.4 - O Java como linguagem padrão para o Serviço de Agentes

A linguagem Java apresenta inúmeras vantagens que a tornam particularmente apropriada para a tecnologia de agentes móveis e, em especial, para a implementação do Serviço de Agentes. Mesmo diante de outras “candidatas”, não sendo consagrada como a única linguagem empregada em sistemas baseados em agentes, o Java carrega argumentos poderosos que o elegem como a melhor opção. As razões para tal afirmação são muitas (ITA 1998):

O principal atrativo da linguagem para a tecnologia de agentes é a portabilidade. A utilização dos "bytecodes" e o ambiente de execução interpretado permitem cenários em que qualquer sistema com recursos suficientes possam "acomodar" (host) programas Java. Atualmente, existem até mesmo máquinas que executam Java "natively". Mais uma vantagem pode ser contabilizada visto que existe a possibilidade de um maior número de plataformas capazes de executar o código dos agentes.

Uma segunda vantagem advém como consequência natural da "onipresença" do Java na Internet. Devido ao fato de estar "embutido" em muitos navegadores "browsers" Web, como também em servidores de aplicação, existem muitas plataformas já desenvolvidas. Ferramentas como AWT (Advanced Windowing Toolkit), JFC (Java Foundation Classes) e JDBC (Java Data Base Connectivity) contribuem, ainda mais, para a disseminação da linguagem.

Outra vantagem principal é a proliferação de ferramentas que apóiam os programadores Java. Muitos programadores já estão familiarizados com C++, ao qual o Java se assemelha em muitos aspectos. Somado a isso está a migração de ferramentas existentes para o Java e a criação de muitas outras. O resultado “líquido” é a fartura de recursos altamente qualificados e de fácil manuseio para desenvolvimento e depuração de programas.

Finalmente, pode-se perceber o movimento dos principais segmentos da indústria de software para a linguagem Java. Em relação particular à tecnologia de agentes móveis, tal constatação pode vir fundamentada pela seguinte prova: a empresa Reticular Systems Inc. realizou, recentemente, um levantamento (Reticular 2000) sobre os sistemas de agentes móveis desenvolvidos tanto comercialmente como em projetos acadêmicos e de pesquisas. Dentro do "universo" apurado de 54 plataformas, a linguagem Java está presente em 29 produtos!

7.4 - O Serviço de Agentes: Definição

Toda fundamentação apresentada até o presente momento quanto às tecnologias, modelos e conceitos reitera a condição de que o Serviço de Agentes, não aceitando a definição de uma aplicação isolada, engloba: 1) A instalação, conFiguração e execução de uma plataforma de agentes móveis; 2) Definição e implantação de um ambiente distribuído de agentes (ADA) sobre um ambiente de processamento distribuído (APD) a partir dos serviços fornecidos pela plataforma de agentes; 3) A utilização de agentes móveis no estabelecimento do serviço de monitoração de objetos condicionado, por natureza e motivação, ao ambiente SICSD. A Figura 7.4 retrata a arquitetura geral do Serviço de Agentes.

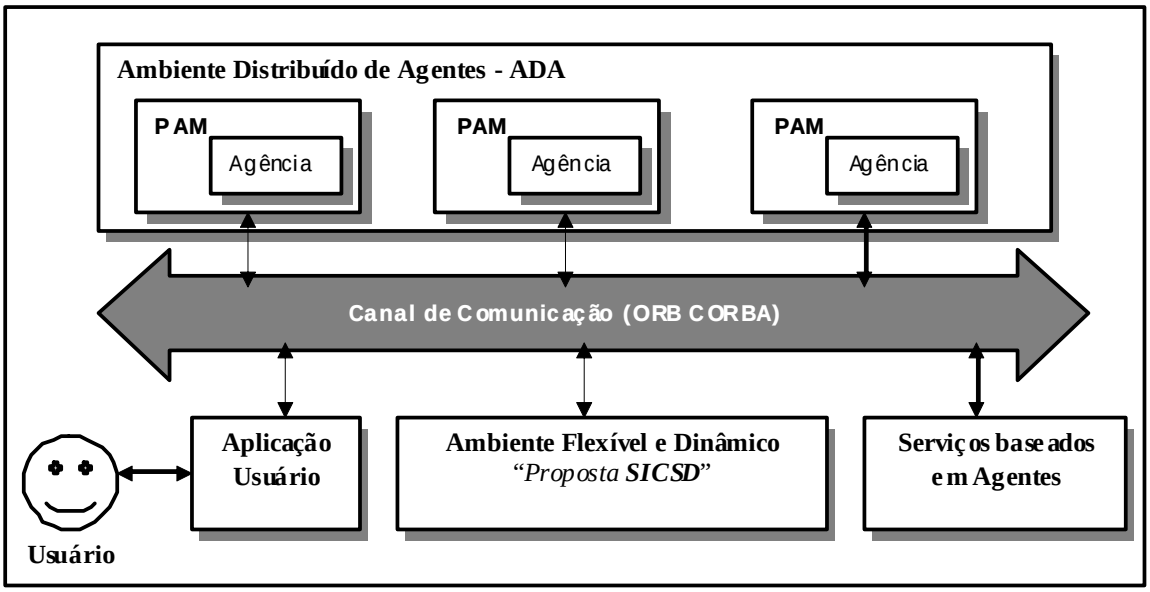

FIGURA 7.4: O Serviço de Agentes inserido na arquitetura do SICSD.

Como configurado na Figura acima, a arquitetura do Serviço de Agentes não abrange unicamente a concepção representada no módulo “Serviços baseados em Agentes” (ver Figura). Na verdade, como já definido no início do capítulo, este serviço abrange e retrata três módulos interdependentes: os agentes, a plataforma de agentes e o serviço-aplicação.

Apresenta-se partir da próxima seção, inclusive, os detalhes concernentes a cada um destes módulos participantes na arquitetura do Serviço de Agentes. Inicialmente serão definidos os pormenores relativos ao ambiente distribuído de agentes (ADA) composto pelas plataformas de agentes e pelos agentes. Em seguida, introduz-se a modelagem do serviço-aplicação. Ou seja, define-se as peculiaridades e a metodologia para a aplicabilidade dos agentes móveis, através do ADA, no serviço de monitoração de objetos na arquitetura do sistema SICSD.

Salienta-se ainda dentro do escopo de definição que toda carga teórica referente ao Serviço de Agentes encontrará, no próximo capítulo, reforço prático na exposição de um ambiente experimental implementado para o protótipo do serviço.

7.5 - O Serviço de Agentes: o Ambiente Distribuído de Agentes (ADA)

De acordo com a descrição da arquitetura introduzida anteriormente, o ambiente distribuído de agentes (ADA) é composto, em suma, pela plataforma de agentes e pelos respectivos agentes. A estrutura disposta na Figura 7.5 conFigura os "participantes" deste ambiente.

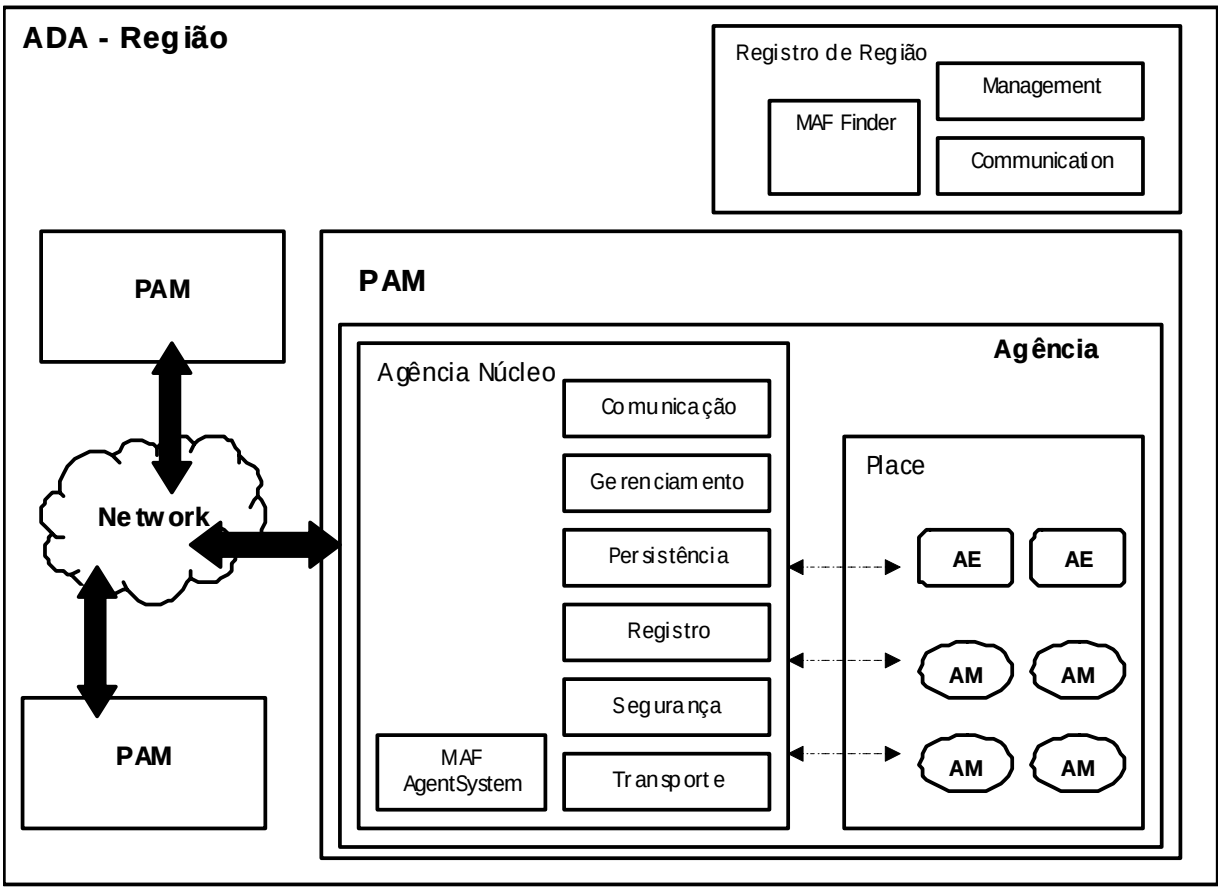

FIGURA 7.5: O ADA : Plataformas, agências e agentes.

7.5.1 – A Plataforma de Agentes (PAM)

Parte da conceituação envolvendo a plataforma de agente foi apresentada em capítulos anteriores relacionados a este tópico. Visto que o presente trabalho não intenta "reinventar a roda" através da implementação de uma plataforma "proprietária", constata-se

a consideração da infra-estrutura fornecida pelos produtos Voyager e Grasshopper destacados no capítulo de comparação de plataformas. A ênfase em torno destas plataformas buscou levantar as características primordiais quanto ao fornecimento da infra-estrutura necessária para o desenvolvimento e gerenciamento da aplicação baseada em agentes móveis.

7.5.2 – As Agências

As agências correspondem à definição MASIF (MASIF) para um sistema de agentes. O ambiente indispensável para a execução destes agentes é uma agência. Em cada máquina ("host") na qual se instala uma plataforma de agentes móveis (PAM), existe pelo menos uma agência "rodando" para o suporte à interpretação, execução, transferência (migração) e destruição dos agentes. Em uma aproximação mais detalhada referente à concepção da plataforma considerada, as agências "rodam" em suas próprias máquinas virtuais (Java Virtual Machine) consistindo em uma agência núcleo e um conjunto de "places" onde "residem" os agentes.

7.5.2.1 – A Agência Núcleo

A agência núcleo representa a funcionalidade mínima requerida por uma agência a fim de suportar a execução dos agentes. Tal funcionalidade é estruturada em serviços (Grasshopper 1999b] tais como Serviço de Comunicação, Serviço de Registro, Serviço de Gerenciamento, Serviço de Segurança, Serviço de Transporte e Serviço de Persistência:

Serviço de Comunicação: Este serviço é responsável por todas as interações remotas estabelecidas entre os componentes da plataforma, tais como a comunicação entre agentes com transparência de localização, migração e localização de agentes através de registros e/ou identificadores únicos. Todas as interações podem ser realizadas via conexões CORBA IIOP, Java RMI ou plain sockets (ver Figura 7.6 abaixo).

Opcionalmente, as conexões RMI e “plain socket” podem ser protegidas através do protocolo padronizado de segurança para Internet conhecido como “Secure Socket Layer” (SSL). O serviço suporta comunicações síncrona, assíncrona e “multicast”, além de permitir invocação dinâmica de métodos.

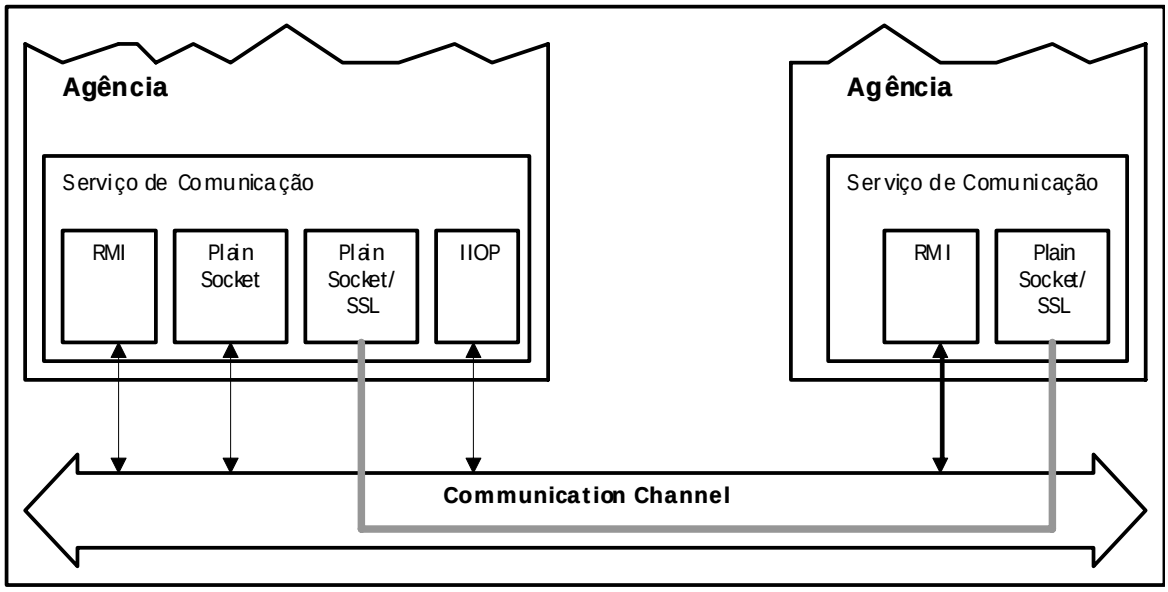

FIGURA 7.6: O Serviço de Comunicação.
FONTE: Grasshopper (1998a).

Serviço de Registro. Cada agência deve estar apta para reconhecer todos agentes e “places” presentes localmente satisfazendo propósitos de gerenciamento externo e disponibilização de informações a respeito das “entidades” registradas no “host” local para futura referência. Além disto, o serviço de registro de cada agência está conectado ao registro de região que mantém informações dos agentes, agências e “places” no escopo do ambiente considerado.

Serviço de Gerenciamento. Os serviços de gerenciamento são desenvolvidos para permitir a monitoração e controle dos agentes e “places” de uma agência por parte dos usuários (humanos). Entre outras, as seguintes funcionalidades merecem citação: a) criar, remover, suspender e reativar agentes, serviços e “places”; b) obter informação sobre agentes e serviços específicos; c) listar todos agentes residentes em um “place” específico; d) listar

todos “places” de uma agência. À parte disto, o gerenciamento de conFiguração habilita a especificação de propriedades de comunicação, de segurança, de “trace” e do sistema.

Serviço de Segurança. Dois mecanismos fundamentais podem ser considerados: segurança externa e interna. A segurança externa protege as interações dos componentes da plataforma, isto é, as interações entre agências e regiões. Para tal propósito, os certificados X.509 e Secure Socket Layer (SSL) são convenientes. SSL é um protocolo padrão que faz uso substancial de criptografia simétrica e assimétrica, permitindo que as partes envolvidas na comunicação desfrutem de confidencialidade, integridade de dados e autenticação mútua. A segurança interna protege os recursos da agência frente ao acesso não autorizado de agentes, além de garantir a segurança para a situação de “ataques” entre agentes. Exige-se, dentro de tais mecanismos, a autenticação e autorização dos agentes que respondem aos interesses de outras partes. Ao lado dos resultados dos procedimentos de autenticação e/ou autorização, ativam-se políticas de controle de acesso. Geralmente os serviços de segurança interna fundamentam-se, em grande parte, nos mecanismos de segurança Java.

Serviço de Persistência. O serviço possibilita o armazenamento dos agentes e “places” (as informações internas mantidas no interior destes componentes) numa estrutura persistente. Desta forma pode-se recuperar os agentes e “places” quando necessário, por exemplo, na reinicialização de uma agência após uma “queda” ou falha do sistema. Destacam-se dois tipos de persistência: implícita e explícita.

Persistência Implícita: Quando o serviço de persistência é ativado, os “places” são automaticamente persistentes após a sua criação. Isto significa que que um “place” existe mesmo após o “shut down” da agência correspondente, permanecendo disponível assim que ocorra a reinicialização da agência. Pode-se habilitar o “salvamento” automático de todos agentes quando a agência sofre um “shut down”.

Persistência Explícita: Distinguem-se três mecanismos existentes: a) Um agente é periodicamente armazenado (“persistentemente”) depois de um certo intervalo de tempo

sem suspender a execução de suas tarefas. O intervalo é especificado pelo próprio agente o qual não se preocupa com a manutenção de suas informações visto que o serviço de persistência se encarrega automaticamente de tais procedimentos que permitem a recuperação dos agentes quando uma agência é reinicializada, por exemplo, após uma "quebra" (crash) do sistema; b) Os agentes podem ordenar ao serviço de persistência que termine aqueles após um tempo determinado de ociosidade no processamento, isto é, quando não recebem solicitações de serviço de outras "entidades". Entretanto, os agentes permanecem registrados dentro da agência e do registro de região, de modo que possam ser reinicializados assim que se detecte tentativas de acesso aos seus serviços; c) Os agentes podem ser terminados explicitamente a qualquer momento em benefício próprio ou a partir de uma solicitação de um administrador da agência. Tal possibilidade se torna adequada quando uma agência deve ser temporariamente terminada.

7.5.3 – Os "Places"

De acordo com as especificações do MASIF (MASIF) (Grasshopper 1999b), os "places" são definidos como contextos dentro de uma plataforma nos quais os agentes executam suas tarefas. Exemplificando, um "Mail Place" pode compreender os serviços de envio e recepção de e-mails e um "Config Place" pode disponibilizar serviços para monitoração de objetos da rede. Ao mesmo tempo, o gerenciamento de segurança é facilitado associando-se uma política de segurança a um "place" específico (e, por conseqüência, a todos os serviços compreendidos). Os "places" podem "hospedar" agentes estacionários e agentes móveis. Toda a funcionalidade presente em um "place" é implementada pela participação e colaboração de agentes que atuam neste "place".

7.5.4 - A Região

O conceito de região (MASIF) facilita o gerenciamento dos componentes distribuídos no ambiente da plataforma, isto é, das agências, "places", serviços e agentes. As agências, bem como seus "places", podem ser associados a uma região específica, ou seja, são registrados dentro de um registro de região.

Cada registro automaticamente armazena cada agente atualmente “hospedado” (hosted) em uma agência associada a uma região. Se um agente migra para outra “localidade”, a informação no registro correspondente é automaticamente atualizada. Uma região pode compreender todas as agências pertencentes a uma companhia, organização ou domínio específicos, facilitando o gerenciamento.

7.5.5 – O Registro de Região

No contexto herdado da MASIF, as regiões representam agregações de sistemas de agentes (isto é, agências) não necessariamente do mesmo tipo mas com autoridades semelhantes. O registro de região armazena informações a respeito de todos associados à uma região específica. Quando um novo componente (agência, “place” ou agente) é criado, tal informação é registrada dentro do registro de região correspondente.

Enquanto as agências e “places” são associados à uma única região durante todo o tempo de atividade, os agentes móveis podem migrar entre diferentes regiões. O registro dos agentes é automaticamente realizado pela agência “hospedeira” de modo transparente para o agente. Se um agente migra, a informação sobre a sua localização é automaticamente atualizada dentro do registro. Desta maneira, um administrador ou agente está sempre apto para obter informações atualizadas sobre a região ou para localizar agentes específicos.

Do mesmo modo, o registro de região facilita o estabelecimento de conexões entre agências e/ou agentes. Por exemplo, um agente A que solicita comunicação com um agente B possui a habilidade de estabelecer tal conexão apenas conhecendo o identificador do agente B. O serviço de comunicação da plataforma automaticamente determina a localização atual do agente B contactando o registro de região e, então, estabelece a conexão. O mesmo se aplica à migração do agente: Um agente pode migrar ao possuir o nome da agência-destino desejada. O nome do “host”, número da porta e protocolo de transporte suportado da agência-destino é automaticamente “colhido” pela agência-origem contactando o registro de região.

O registro de região provê os seguintes serviços/componentes:

Management: O serviço de gerenciamento é responsável pela localização dos agentes dentro de uma região.
Communication: Equivalente ao serviço oferecido pela agência (abaixo descrito), possibilita interações entre o registro de região e "entidades" remotas (agências, agentes, ...). Uma interface de texto para usuário permite que os administradores monitorem e controlem o processamento dos registros.
MAFFinder: Este componente implementa uma interface que faz parte do padrão OMG MASIF e que incrementa a interoperabilidade entre outras plataformas de agentes compatíveis.

Concluindo, deve-se ressaltar que a utilização deste serviço é opcional. Se tal estrutura não for considerada, a provisão de outros mecanismos, através dos quais os agentes e agências obtenham informações para realizar interações remotas, é apropriada. Se o ambiente de agentes for estabelecido com o serviço de registro região, o registro deve ser inicializado ***antes*** que a primeira agência seja criada.

7.5.6 - Os Agentes

Até o presente momento, como já foi comentado no capítulo que versava sobre a Tecnologia de Agentes, não existe uma definição única de um (software-) agente. Entretanto, destacou-se também que os agentes podem ser caracterizados por um conjunto de atributos, entre os quais se identifica a autonomia como uma referência comumente aceita. Levando-se em conta tal identificação, um agente, em consenso geral, é um programa que age com autonomia em benefício de uma pessoa ou organização.

No contexto idealizado para o Serviço de Agentes, verifica-se a atuação de dois tipos de agentes: os agentes estacionários e os agentes móveis.

7.5.6.1 - Agentes Móveis

Os agentes móveis são capacitados para se mover entre as localizações físicas de uma rede. Desta forma, podem ser considerados como “alternativa adicional / avanço / complemento” ao paradigma tradicional cliente / servidor. Enquanto a tecnologia cliente / servidor se sustenta nos protocolos de “Remote Procedure Calling” (RPC), os agentes móveis podem migrar para locais determinados e obter vantagens quanto à interações locais.

Entretanto, mesmo com o destaque dos inúmeros benefícios da tecnologia de agentes móveis já relacionados anteriormente, firma-se o pressuposto de que a metodologia baseada em agentes não visa substituir o paradigma cliente/servidor. Por outro lado, reafirma-se o objetivo quanto à integração das tecnologias, através da qual se considera, entre outras possibilidades, a implementação de agentes que se comportem tanto como clientes ou como servidores. Assim, um cenário dentro do qual um agente-cliente migra para um servidor (tradicional) ou um agente-servidor migra para um cliente (tradicional) pode ser idealizado. Neste caso, constata-se a adequação e vantagens de interações locais em relação à comunicação via RPC. Em outra concepção, dois agentes se comunicariam remotamente, ou seja, não estão presentes na mesma localização. Resta, portanto, a avaliação de cada cenário quanto às diferentes possibilidades, ou seja, uma tomada de decisão que se alterna entre a utilização da “migração/ interação local” ou uso da “interação remota”.

7.5.6.2 - Agentes Estacionários

Em contraste com os agentes móveis, os agentes estacionários não possuem a habilidade para migrar “ativamente” entre diferentes localidades de uma rede. Assim, tais agentes são associados com uma localização específica / fixa.

7.5.6.3 – Os Estados do Agente

Durante seu ciclo de vida, o agente pode se apresentar em diferentes estados: ativo, suspenso e desativado.

Ativo: um agente está ativo se, no momento, realiza sua tarefa. Um agente ativo pode ser suspenso ou desativado.

Suspenso: um agente está suspenso se a execução de sua tarefa é temporariamente interrompida. Entretanto, o agente permanece instanciado e dá continuidade, quando reiniciado, à execução de sua tarefa.

Desativado: um agente desativado não está instanciado. Desta forma, toda informação interna relevante é armazenada permanentemente, por exemplo, em um disco rígido. Um agente desativado pode ser ativado novamente, ou seja, uma nova instância do agente é criada, dados relevantes à execução são repassados à nova instância e o agente reinicia sua execução.

7.6 - A Funcionalidade do ADA: A Estrutura das Classes de Agentes

Consumando o conceito do Ambiente Distribuído de Agentes, descreve-se a estrutura das classes de agentes. Dentro desta estruturação, cada agente implementa as abstrações de um serviço, ou seja, um componente-software que oferece funcionalidades para outras "entidades" dentro do ambiente do Serviço de Agentes.

Cada "agente / serviço" pode ser subdividido em duas partes: uma que trata de aspectos comuns a todos agentes e outra relacionada individualmente à sua funcionalidade. A parte comum (ou "core functionality") é representada pelas classes **Service, MobileAgent** e **StationaryAgent**, todas disponibilizadas pela plataforma de agentes (Grasshopper 1999b). A parte que representa a funcionalidade individual acrescida ao agente é implementada, segundo critérios e necessidades da aplicação, pelo desenvolvedor.

Desta forma, fica definido que a “core funcionality” é disponibilizada pela infra-estrutura da plataforma de agente (tópicos de segurança, funções de controle de ciclo de vida, comunicação, acesso à plataforma, etc). Por outro lado, a funcionalidade “adicional” e customizada à aplicação, tais como interfaces para software/hardware externos, controle de tarefas (itinerários) e interfaces gráficas específicas, é implementada pelo desenvolvedor através de classes de agentes específicas .

A Figura 7.7 descreve os comentários dos últimos parágrafos.

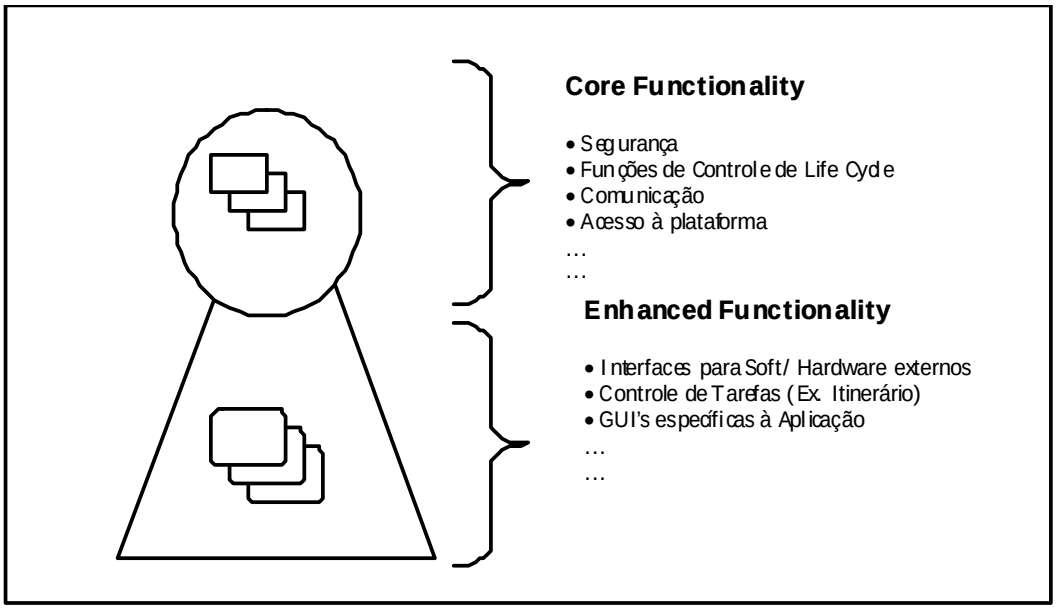

FIGURA 7.7: Funcionalidades Comuns e Adicionais.
FONTE: Grasshopper (1999b).

Segundo os princípios adotados para a arquitetura do ambiente ADA, distinguem-se duas categorias fundamentais de agentes: os agentes móveis (através da classe MobileAgent) e os agentes estacionários (através da classe StationaryAgent). Os agentes móveis são capazes de migrar de um place para outro. De modo contrário, os agentes estacionários aplicam a abrangência de suas atividades fixando-se dentro dos limites de um determinado “place”.

A fim de se atingir a granularidade mencionada acima, isto é, a separação de um agente em partes comum e específica, como também a subdivisão em agentes móveis e

estacionários, considera-se a hierarquia de classes (Grasshopper 1999b), mostrada na Figura 7.8 abaixo, associada a cada agente/serviço:

Os métodos definidos dentro da hierarquia apresentada devem permitir que os agentes interajam com o ambiente local – agência corrente na qual o agente executa, e se comuniquem, com transparência de localização, com outros agentes.

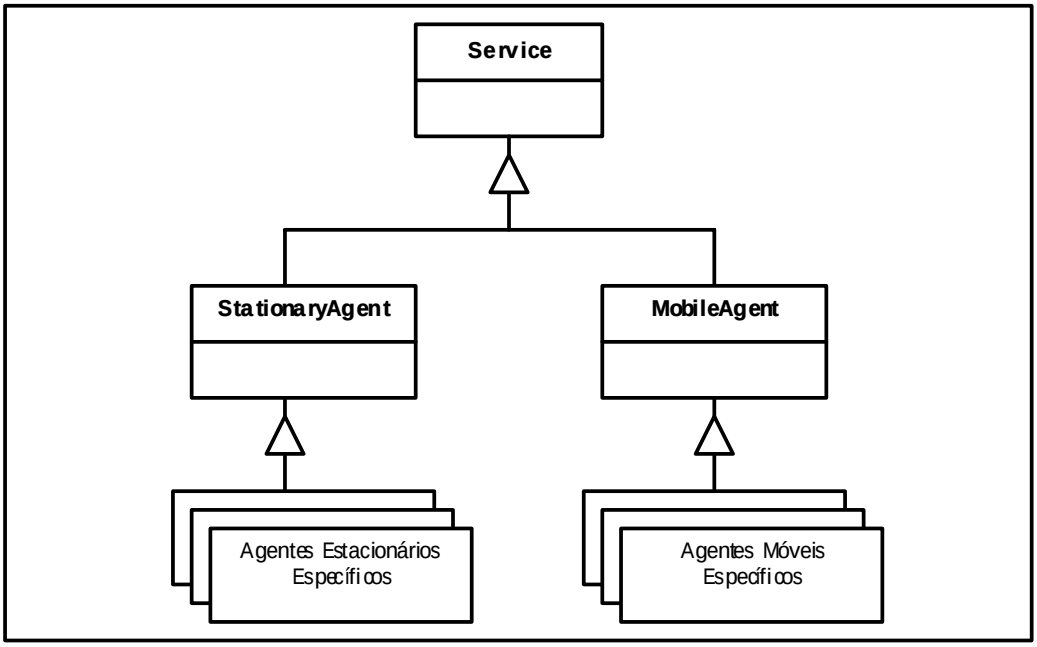

FIGURA 7.8: Hierarquia de classes para os agentes.
FONTE: Grasshopper (1999b).

Entretanto, um aspecto muito importante ainda merece tratamento: ***como o desenvolvedor especifica as diferentes tarefas, missões ou serviços para "seus" agentes?*** A reposta para esta questão é explicada nos parágrafos seguintes aproveitando-se dos conceitos sugeridos pela plataforma Grasshopper (Grasshopper 1999a) através do método "live ()".

7.6.1 - O Método "Live ()"

A missão completa de um agente, ou seja, todas atividades (declarações Java, enfim) realizadas durante seu ciclo de vida, é abrigada dentro do escopo de um método

denominado **live ()**. Trata-se de um método abstrato fornecido respectivamente pelas classes **MobileAgent** e **StationaryAgent**. Em resumo, o fato de o método **live ()** ser encontrado "vazio" dentro da definição da classe de um respectivo agente tem como consequência direta a implementação de uma instância "agente" que faz literalmente NADA.

O método **live ()** pode comportar qualquer declaração Java, incluindo a invocação de métodos definidos em outras classes. Logo, um agente pode ser "composto por" ou "consistir de" diferentes objetos Java. Entretanto, apenas uma única classe deriva ou da classe **MobileAgent** e **StationaryAgent**, sendo considerada como "coração" ou "núcleo" atual do agente compreendendo o método **live ()** e, por consequência natural, a tarefa e/ou missão do agente. A seção a seguir explica algumas regras que devem ser consideradas ao se implementar o método **live ()** de um agente móvel

7.6.1.1 - O Método "Live ( )" para Agente Móveis

Existe uma diferença significativa entre agentes móveis e o conceito de código móvel "tradicional", a qual pode ser descrita por dois tipos de mobilidade: execução remota e migração (Grasshopper 1999b). A execução remota significa que um objeto em movimento inicia sua execução de tarefa a partir do mesmo começo (ponto inicial) após cada movimento. Tal conceito não é inédito e não se relaciona diretamente com a tecnologia de agentes móveis. Os objetos que apresentam a habilidade de execução remota não deveriam ser considerados com agentes móveis e sim como código móvel.

Migração significa que um objeto (agente), após migrar para um novo local (place), retoma a execução de sua tarefa exatamente a partir do ponto onde ocorreu a interrupção que antecedeu o movimento. A migração requer, além da transferência do código do agente, a transferência do *estado de execução*. Tal conceito indica o ponto atual de execução da tarefa de um agente representado, por exemplo, pela próxima declaração a ser realizada (processada). Um agente móvel pode, então, iniciar sua execução na localidade A, migrar para a localidade B e ***continuar/retomar*** sua execução na localidade B

exatamente a partir do ponto em que foi interrompido antes da migração (Grasshopper 99b).

A linguagem de programação Java não permite o armazenamento da pilha de execução de um programa (agente, por exemplo). Portanto, o estado de execução de um agente deve ser representado de outra forma. Os seguintes conceitos podem ser considerados alternativamente a fim de se efetivar a migração de agentes:

As declarações compreendidas pelo método **live ()** de um agente móvel são particionadas em vários blocos de execução. Cada bloco é totalmente executado em um place específico e o último método de cada bloco é o método **move ()**, isto é, após efetivar um bloco de execução, o agente migra para o próximo "place" definido no seu itinerário. Após cada migração, o agente deve determinar o próximo bloco a ser executado, através do encapsulamento dos blocos de execução dentro de uma estrutura de controle **switch** ou várias declarações **if** respectivamente.

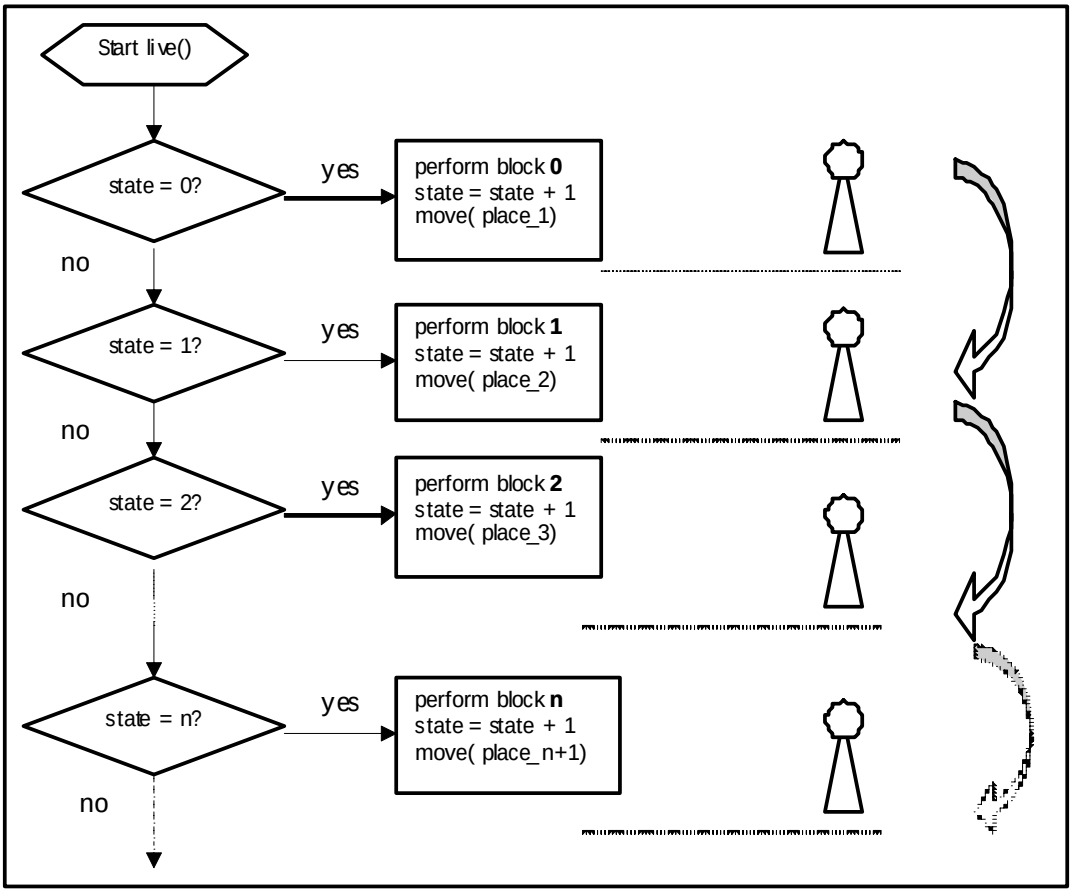

FIGURA 7.9: Diagrama de fluxo para método live ( ).
FONTE: Grasshopper (1999b).

A Figura 7.9 demonstra a estrutura do método **live ()** através de um diagrama de fluxo. No diagrama acima, a variável "state" indica o bloco a ser executado. Após cada migração, isto é, após a execução do método **move ()**, o método **live ()** é reinvocado. Entretanto, dependendo do valor atual da variável "state", que é incrementado após cada migração, apenas o bloco associado a este valor é executado.

7.7 - O Serviço de Agentes: o Serviço-Aplicação

Considerada a estrutura definida para o ambiente ADA, busca-se a conclusão do Serviço de Agentes através da disponibilização de serviços-aplicação para o sistema baseado na proposta SICSD. Os cenários possíveis para aplicações baseadas em agentes se multiplicam progressivamente.

Além do Serviço de Monitoração de Objetos, o serviço-aplicação destacadamente almejado, pode-se ressaltar algumas variedades de serviços tais como: a) Localização Distribuída de Falhas; b) Balanceamento de Carga; c) Avaliação de Performance; d) Instalação de Software; e) Auditoria; f) Estado da Rede; g) Segurança; h) FTP; i) Exame SNMP; j) Alarmes.

7.7.1 - O Serviço-aplicação: Monitoração de Objetos

Visto que a concepção deste serviço liderou a motivação para o desenvolvimento do Serviço de Agentes, a presente seção apresenta as peculiaridades na confecção do mesmo. Em suma, o serviço de monitoração de objetos abstrai a proposta de se implementar mecanismos para manutenção (entenda-se criação, manipulação e controle) de tabelas de representação do estado do sistema SICSD[3]. A elaboração deste serviço obedece o diagrama de estruturas apresentado na Figura **7.10** abaixo.

[3] o estado dos objetos que compõem o sistema.

A compreensão geral do cenário esboçado abaixo que compõe o serviço de Monitoração de Objetos será alcançada à medida em que se apresenta sucessivamente todos os "participantes" arrolados e os respectivos relacionamentos entre eles.

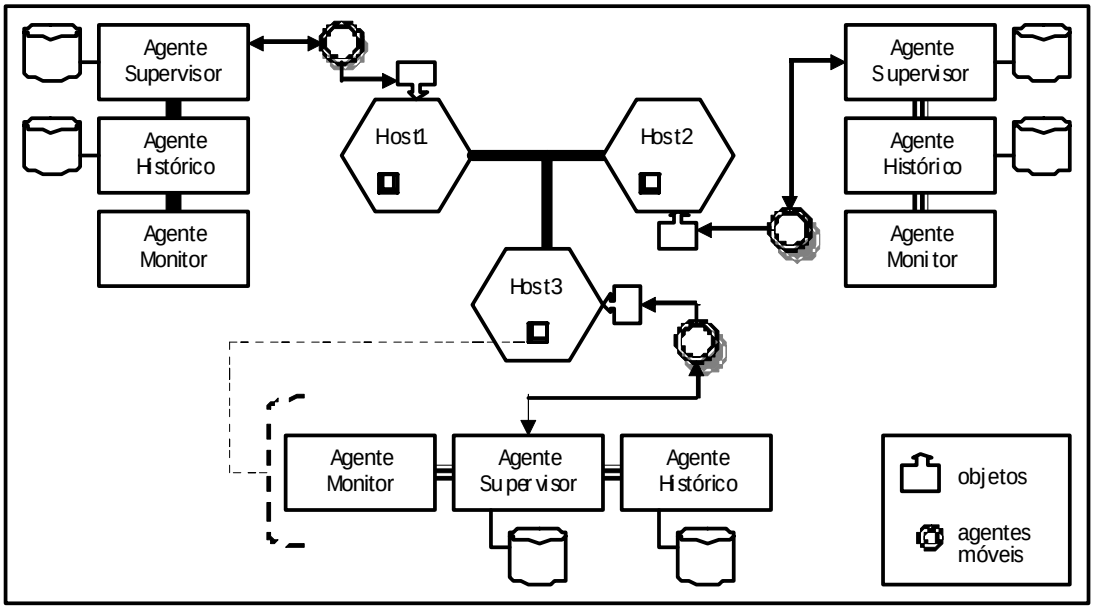

FIGURA 7.10: Serviço de Monitoração dos objetos SICSD através do Serviço de Agentes.

7.7.1.1 - Os Agentes Supervisores

Os Agentes Supervisores (AS) são agentes estacionários instanciados em todos os nós do ambiente, especificamente dentro do "place" apropriado do serviço de monitoração. Cada Agente Supervisor se responsabiliza pelo "place" em que é instanciado representando o papel de "entidade superior" na supervisão de todos procedimentos locais envolvidos.

Recordando que o "place" faz parte dos domínios de uma agência, o Agente Supervisor customiza algumas funcionalidades disponibilizadas pelos serviços desta última, efetivando e concluindo os limites de suas atividades. Nas entrelinhas de algumas das funções deste agentes, percebe-se a presença de similaridades com os serviços disponibilizados pelas agências descritos anteriormente.

As principais atividades (“tasks”) de responsabilidade do Agentes Supervisor são:

task1 : Figurando como função primordial, objetiva o controle e manipulação das tabelas de configuração que compõem a Base de Dados Global dos Objetos –BDGO (introduzida abaixo).

task2 : Coordenar e gerenciar a participação dos agentes móveis e outros agentes estacionários para o desempenho conveniente do serviço de monitoração dentro do “place”.

task3 : Acompanhar a migração dos agentes móveis que desempenham tarefas externas ao “place” e/ou à agência local, facilitando as “negociações” dentro da região e garantindo o sucesso da transferência. Se a migração de um agente móvel é rejeitada, o AS deve propor soluções para o reestabelecimento do agente e manipulação de outros destinos alternativos para efetivação da tarefa.

task4 : Verificar e autenticar as solicitações dos agentes móveis provenientes de outros “places” e/ou agências, rejeitando ou limitando o acesso ao ambiente de execução.

task5 : Intermediar o acesso aos recursos (tabelas, agentes,...) do “place”. Quando um agente móvel migra com sucesso, o AS fornece um “ticket” de permissão de acesso que subseqüentemente pode ser requisitado por outros agentes para comunicação e negociação de informações.

task6 : Fornecer uma área de registro central dentro do “place” onde os agentes possam se registrar e declarar suas “intenções” e “interesses”. Logo, o “place” pode atuar como ponto de encontro no qual os agentes compartilham informações e alcançam seus objetivos.

task7 : Garantir que o “place” não acolha uma “inundação” de agentes limitando o número destes dentro do ambiente num dado momento e impondo restrições de tempo de acesso aos recursos.

7.7.1.2 - A Base de Dados Global de Objetos

Seguindo as especificações da arquitetura proposta e atuando como protagonista dentro do cenário Figurado pela "task1", a Base de Dados Global de Objetos coleciona diversas tabelas de configuração concernentes tanto ao âmbito dos objetos como ao estado do sistema (através de algumas variáveis de estado). O termo "global" ressalta que as informações armazenadas pelas tabelas procuram retratar uma visão, a partir de cada "host" , generalizada da situação de todos os outros "hosts" do ambiente. Para tanto, definem-se as seguintes tabelas: a) Tabela de nós; b) Tabela de objetos; e c)Tabela de conexões.

A **tabela de nós**, contendo a relação dos "hosts" onde serão carregados o sistema, compõe-se pelos seguintes campos (ver Tabela 7.1 e 7.2): *nome do nó, desempenho, CPU disponível*; *I/O disponível, número de conexões, status, taxa de CPU disponível* e *taxa de I/O disponível*. A tabela fornece melhores detalhes a respeito dos referidos dados.

TABELA 7.1: Tabela de Nós ("Hosts" ou Agências).

Nome	desempenho	CPU disponível	I/O disponível	num. conexão	status	taxa de CPU disponível	taxa de I/O disponível
Patras	5	0.5	0.4	10	ready	2.5	2.0
Pelion	2	0.7	0.2	20	suspend	1.4	0.4
Andros	1	0.4	0.1	0	ready	0.4	0.1

TABELA 7.2: Dados da Tabela de Nós (“Hosts” ou Agências).

Nome do nó	Contém o nome do computador daquele ‘host” (agência).
Desempenho	O desempenho é baseado no “benchmark” de cada computador. Os testes de benchmark podem ser divididos em dois tipos: Benchmark para componentes isolados e para todo o sistema. Os benchmark para componentes isolados medem o desempenho específico de um determinado componente, como por exemplo, o tempo de acesso ao Winchester, o tempo de acesso a memória RAM de um computador, etc. O benchmark para todo o sistema mede o desempenho do sistema como um todo. Isto envolve tempo de acesso ao Winchester, memória, espera por I/O, etc. No intuito de mensurar o desempenho do sistema como um todo, cogita-se na seleção do benchmark “Winstone 99”. Este aplicativo pode ser utilizado para medir o desempenho de aplicações que executam nos sistemas operacionais da Microsoft (Wndows 95, 98 e NT). Através deste benchmark, pode-se obter uma idéia global do desempenho de cada máquina utilizada no sistema SICSD. A aplicação para controle de satélites é heterogênea, ou seja, tem objetos do sistema que utilizam somente os mecanismos de entrada e saída de dados e tem objetos que acessam somente a CPU. Portanto a utilização de um benchmark que medisse simplesmente o desempenho do processador não retrataria fielmente a realidade do sistema. A atribuição do desempenho para cada nó segue o seguinte critério: A partir da máquina com menor desempenho, a qual recebe peso referencial 1 (um), calcula-se os pesos das demais máquinas proporcionalmente ao peso padrão de referência. Por exemplo, a máquina que apresenta um desempenho duas vezes superior ao peso padrão recebe o valor de peso 2.(dois). O parâmetro desempenho é calculado para todas as máquinas pertencentes ao sistema e armazenados na tabela de nós.
CPU disponível	O parâmetro CPU disponível é capturado periodicamente pelo agente supervisor fixo em cada nó do sistema. Ele mostra em taxas percentuais a disponibilidade de CPU num dado momento. A freqüência de leitura deste parâmetro pode ser configurável a partir de restrições de calibragem que garantam a condição necessária e suficiente na obtenção conveniente de noções de carga do sistema em um determinado instante.
I/O disponível	O parâmetro I/O disponível é capturado de tempos em tempos pelo agente supervisor estacionário em cada nó do sistema. Traduz através de taxas percentuais a disponibilidade corrente do processador de entrada e saída. A freqüência de leitura deste parâmetro pode ser configurável. Os intervalos de “checagem” podem obedecer às mesmas taxas definidas para o parâmetro CPU disponível.
Número de Conexões	O número de conexões contém o somatório de todas as conexões ou solicitações de serviços para um determinado nó. Este parâmetro é atualizado a cada solicitação de serviço para um objeto instanciado naquele nó.
Status	Este parâmetro contém o estado de um nó em um determinado instante. A freqüência de leitura deste parâmetro pode ser configurável. Um nó pode se encontrar nos seguintes estados: a) “ready”: pronto para receber solicitações de serviços; b) “failed”: Ocorreu uma falha em o nó não conhecida (por exemplo a queda do sistema operacional, falha de conexão em rede, etc.); c) “suspend”: O nó pode estar suspenso para manutenção.
Taxa de CPU disponível	Este parâmetro mede a disponibilidade da CPU de um determinado nó em um determinado instante. A freqüência de cálculo deste parâmetro segue a freqüência de leitura do parâmetro CPU disponível. A taxa é calculada em função do resultado da multiplicação do parâmetro desempenho pelo parâmetro CPU disponível. Quanto maior o resultado, mais disponível está o sistema, como por exemplo, baseando-se na Tabela 7.1 o nó ou agência patras tem a taxa igual a 2.5, portanto ele está com uma carga de processamento menor do que o nó ou agência pelion que tem a taxa igual a 1.4. A taxa de CPU disponível permite uma visão geral a respeito de quais nós têm uma carga maior de processamento, atentando-se para o fato de que quanto menor o valor desta variável, mais sobrecarregado está o nó. A variação do valor desta taxa depende da freqüência de monitoração do parâmetro CPU disponível. Este parâmetro pode orientar a análise do balanceamento de carga de CPU do sistema. Ainda mais, em direção ao balanceamento total do sistema, pode-se estabelecer mecanismos que garantam a constatação de taxas com magnitudes aproximadas em cada nó quando comparadas aos demais nós.
Taxa de I/O disponível	O parâmetro taxa de I/O disponível mede a disponibilidade corrente do processador de entrada e saída (Input/Output) de um determinado nó. A freqüência de cálculo deste parâmetro segue a freqüência de leitura do parâmetro I/O disponível. É calculada em função do resultado da multiplicação do parâmetro desempenho pelo parâmetro I/O disponível. Quanto maior o resultado, mais disponível está o sistema, como por exemplo, baseando-se na Tabela 9.1 o nó ou agência patras tem a taxa igual a 2.0, portanto apresentando uma sobrecarga do processador de I/O menor do que a do nó ou agência pelion que tem a taxa igual a 0.4. A taxa de CPU disponível permite uma visão geral a respeito de quais nós têm uma carga maior de processamento, atentando-se para o fato de que quanto menor o valor desta variável, mais sobrecarregado está o nó. A variação do valor desta taxa depende da freqüência de monitoração do parâmetro I/O disponível. Este parâmetro pode orientar a análise do balanceamento de carga de I/O do sistema. Ainda mais, em direção ao balanceamento total do sistema, pode-se estabelecer mecanismos que garantam a constatação de taxas com magnitudes aproximadas em cada nó quando comparadas aos demais nós.

A **tabela de objetos** armazena a relação dos objetos instanciados em cada nó dedicado à execução do sistema distribuído para controle de satélites. Esta tabela é constituída pelos seguintes campos (ver Tabela 7.3): *nome do objeto, nome do "host" ou agência onde o objeto está instanciado, número de conexões existentes para este objeto* e *"status" do objeto* ("ready": objeto pronto para receber conexões, "failed": objeto não responde à solicitação de serviço evidenciando, possivelmente, falha no próprio objeto, falha na rede ou falha na máquina onde o objeto está instanciado).

De acordo com a tabela abaixo, o objeto "obj1" estaria instanciado no "host" Patras, contendo 5 conexões ativas e pronto ("ready") para receber novas conexões.

TABELA 7.3: Tabela de Objetos.

nome	nó	nconexao	status
obj1	patras	5	ready
obj2	pelion	0	failed
obj3	andros	10	ready
obj2	andros	4	ready

A **tabela de conexões** se diferencia da tabela de objetos evidenciando, para cada objeto, a origem das solicitações (ver Tabela 7.4).

TABELA 7.4: Tabela de Conexões.

nó	objeto	origem	num. conexão
patras	obj1	patras	3
patras	obj1	pelion	2
andros	obj2	patras	3
andros	obj3	pelion	10

De acordo com a tabela acima, o objeto "obj1" instanciado no "host" Patras manipula 3 conexões com origem no próprio "host" e 2 conexões com origem no "host" Pelion, totalizando 5 conexões.

7.7.1.3 - Os Agentes Históricos

Concomitante ao contexto de operação do Agente Supervisor, podemos considerar o seguinte cenário. Após a inicialização de todos agentes supervisores e conclusão de todos processos de carga do sistema, destaca-se a "instalação" ou instanciação dos objetos aplicativos em determinadas agências / "host" do ambiente. Quando estes objetos passam a receber solicitações de serviços, surge o estabelecimento de conexões entre clientes e serviços.

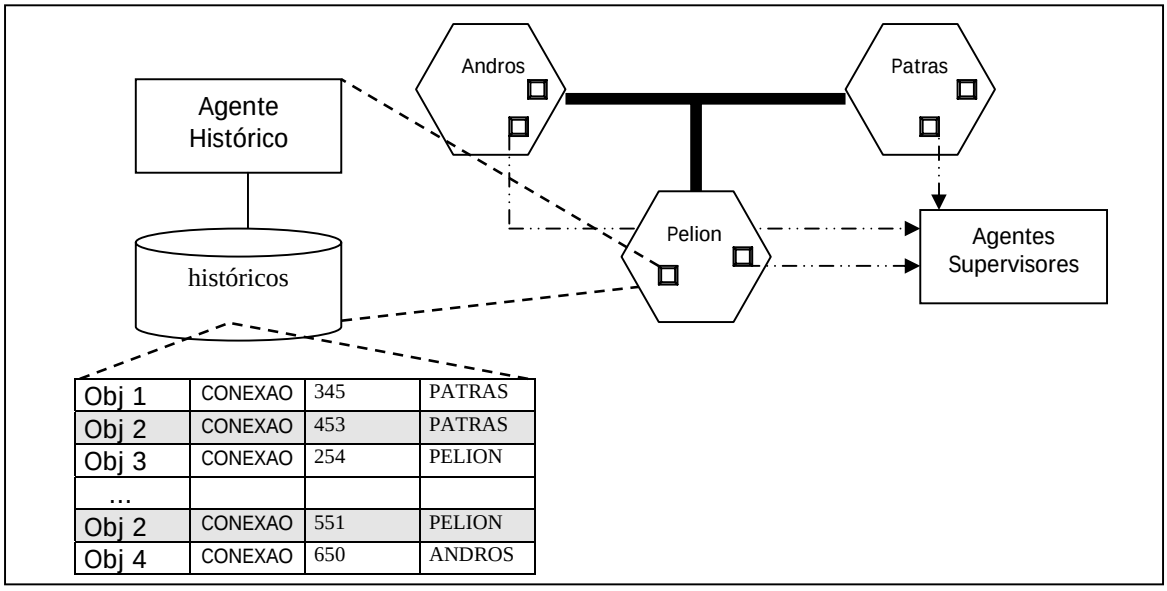

FIGURA 7.11: Agente Histórico "registrando" temporariamente as conexão dos objetos no "host" Pelion.

Seguindo requisitos e funções primordiais, os Agentes Supervisores devem manter-se atualizados quanto à situação, utilização e tipos de operações requeridas dos objetos no âmbito local e global do sistema. O Agente Histórico provê uma estrutura que sirva de entreposto para todas informações até que agentes apropriados concluam a atualização e disseminação das mesmas. Num exemplo simplista, podemos imaginar o objeto TMS instanciado na agência (host) Patras, que processa telemetrias, estabelecendo uma conexão com um cliente na agência (host) Andros (consideremos que tal conexão foi previamente avaliada como a mais apropriada seguindo restrições atuais do sistema). Todas as variáveis

pertinentes à conexão devem ser armazenadas até que agentes apropriados efetuem a disseminação pelo sistema permitindo, então, que as mesmas sejam eliminadas.

Fica evidente a utilidade de tal estrutura que, armazenando registros das "transações" pertinentes ao modelo, garante a persistência das mesmas enquanto não são atualizadas e disseminadas em todas as tabelas. O Agente Histórico é um agente estacionário cuja atuação se restringe ao contexto local de apenas uma agência (host), isto é, limita-se a considerar apenas as transações em andamento local sem se preocupar com a situação dos Agentes Históricos de outras agências. Inicialmente, seguindo os padrões do Agente Supervisor, as tabelas (Históricos DB) poderiam armazenar registros com nome do objeto, código da operação, hora e origem da solicitação. O cenário de atuação do Agente Histórico pode ser visualizado na Figura 7.11.

7.7.1.4 - Os Agentes Monitores de Objetos

Os Agentes Monitores de Objetos (AMO) são agentes estacionários instanciados dentro de um "place" específico estabelecendo um nível de abstração entre os objetos (serviços da proposta SICSD), o Agente Supervisor e os Agentes de Atualização. Assim que um novo objeto é instanciado dentro dos domínios da agência, o Agente Supervisor designa um agente AMO que concretiza um vínculo "vitalício" com o objeto e implementa periodicamente as seguintes funcionalidades (ver Figura 7.12):

As principais tarefas destes agentes são:

a) Verificação do estado do objeto e transferência das informações para o Agente Supervisor. O objeto pode apresentar os estados: ativo, inativo (manutenção) e inativo (sem resposta).

b) Análise da situação das conexões estabelecidas com o objeto de acordo com variáveis como: número de conexões, tempo de conexão, origem da conexão, etc. Os dados são transferidos para tabelas apropriadas sob o controle do Agente Supervisor.

c) Consultas ao Agente Histórico pesquisando a existência de registros pendentes para atualização. A seguinte situação é prevista: o agente AMO procura por registros relacionados ao objeto monitorado e "estocados" provisoriamente dentro da estrutura do Agente Histórico, transferindo-os para o armazenamento definitivo sob inspeção do Agente Supervisor. Tal mecanismo objetiva garantir que o processamento dos dados gerados em escala distribuída e simultânea atinja níveis aceitáveis de sincronização e eficiência.

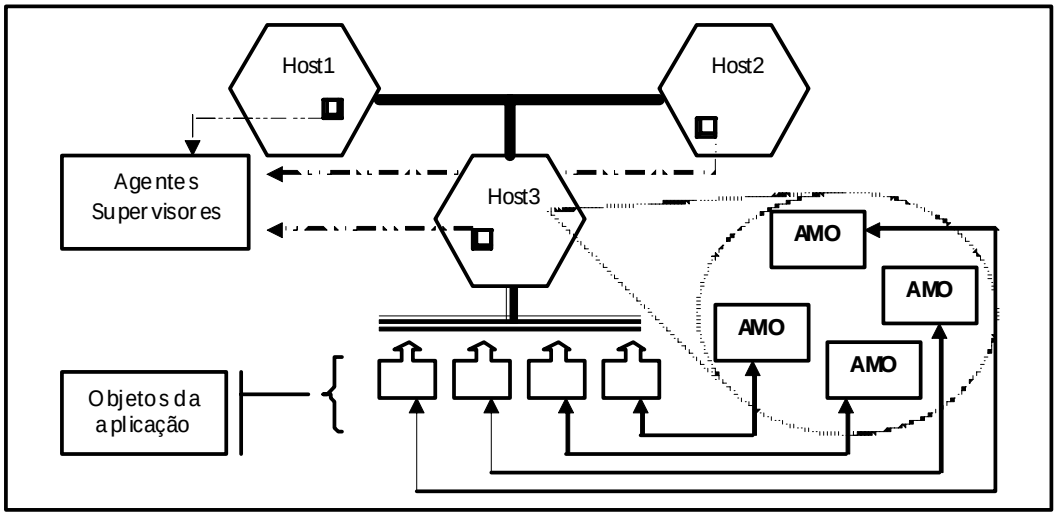

FIGURA 7.12: Agentes AMO monitorando objetos da aplicação (proposta SICSD).

7.7.1.5 - Os Agentes de Atualização e Disseminação

Os Agentes de Atualização e Disseminação (AAD) são agentes móveis que exploram a capacidade de migração entre os "domínios" do ambiente para efetivar suas "missões". Deduz-se, sem maiores dificuldades, que as funções essenciais destes agentes se relacionam diretamente aos "procedimentos" para atualização e disseminação das informações contidas nas tabelas de conFiguração. Os métodos de definição desta classe deve permitir abordagens flexíveis para os seguintes casos:

a) **Delegação (disparo) do agente**: Por "default", o Agente Supervisor se encarrega pelo "disparo" dos agentes AAD definindo estratégias que possibilitem adaptações frente a alterações previstas ou não. Por exemplo, em casos esporádicos, o Agente Supervisor pode transferir temporariamente o controle de tomada de decisões para migração ao próprio agente móvel.

b) **Determinação da freqüência das "operações"**: A freqüência em que o agente migra pelas agências atualizando e disseminando as informações, por "default", obedece a parâmetros determinados pelo Agente Supervisor. Convém destacar que a periodicidade pode ser calibrada de acordo com as necessidades do ambiente ou segundo as decisões do Agente Supervisor. Exemplificando: ao se detectar um acúmulo excessivo e indesejado de registros armazenados pelo agente histórico, os parâmetros que ditam a freqüência de atuação dos agentes AAD devem ser reavaliados e reprogramados.

c) **Definição de Itinerários**: Os itinerários orientam a seqüência de agências que devem ser visitadas durante as operações de atualização. Informados a respeito das condições de tráfego das "vias de acesso" pelas quais passará, o AAD obtém a descrição dos melhores roteiros a fim de obter um melhor desempenho durante as movimentações ao mesmo tempo em que evita a sobrecarga dos "caminhos" atualmente congestionados.

d) **Transferência de Informação (Replicação)**: A fim de cumprir sua "missão", o agente AAD basicamente deve transferir (leia-se replicar) informações para todas as agências a partir das tabelas locais. De acordo com critérios específicos, pode-se verificar a conveniência entre, pelo menos de início, duas abordagens para a replicação de informação: 1ª) o agente se responsabiliza por transferir registros associados individualmente a apenas um objeto. Por exemplo, o agente verifica a tabela de objetos à procura de alterações no registro de um objeto específico e, quando necessário, executa a replicação. 2ª) o agente se responsabiliza pela transferência de registros associados a todos os objetos instanciados em uma agência. Por exemplo, o agente verifica periodicamente a tabela de objetos à procura de alterações nos registros de todos os objetos instanciados em uma determinada agência e, quando necessário, executa a replicação.

Considerando-se, por exemplo, a tabela de objetos representada na Figura 7.13, verifica-se a presença de dois objetos - **Obj1** e **Obj2**- instanciados na agência ("host") "PATRAS". À medida que ocorram alterações nos registros de ambos os objetos, o agente AAD deve replicar a informação referente aos dois registros. Em outra situação, na hipótese de que durante um certo intervalo de tempo o registro do objeto **Obj1** não sofra alterações, o agente AAD poderia replicar a informação do registro referente ao objeto **Obj2**.

A Figura abaixo visa demonstrar o esquema básico desempenhado pelo agente:

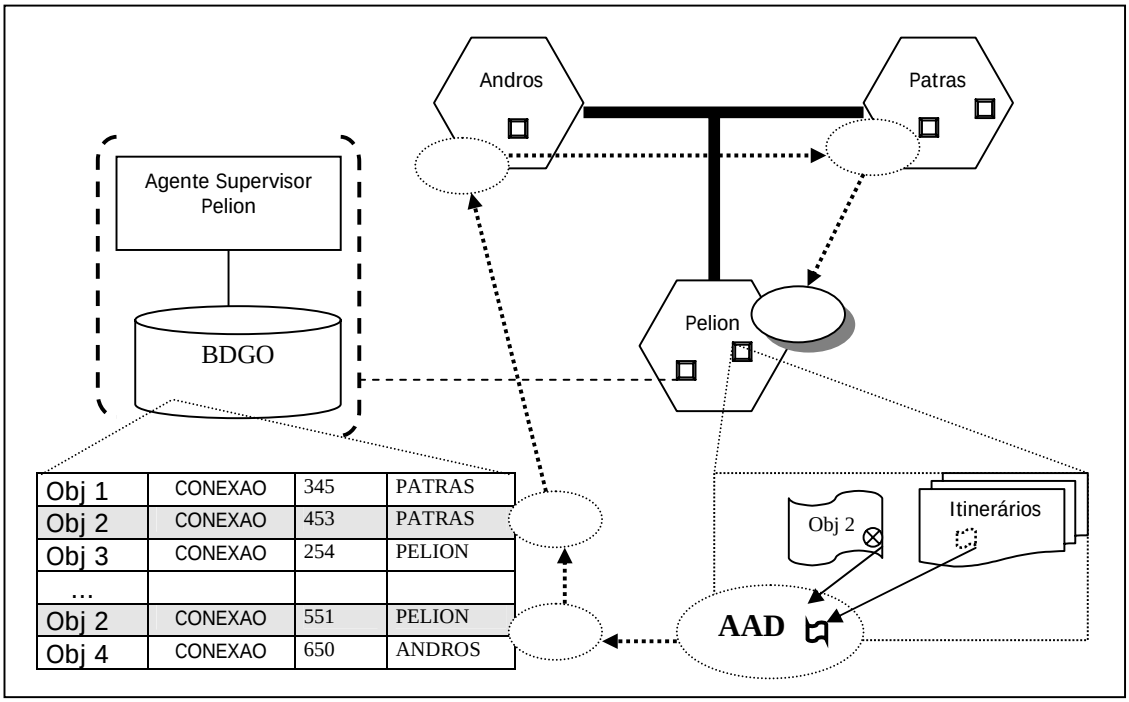

FIGURA 7.13: O Agente AAD disseminando informações sobre o objeto **Obj2**.

7.7.1.6 - Os Agentes Facilitadores de Conexões (AFC)

No momento em que um cliente solicita "serviços" de uma determinada aplicação, presume-se que a localização dos objetos seja intermediada ou previamente disponibilizada por uma camada específica dentro de um ambiente de objetos distribuídos. Não desejando "mergulhar" em tais detalhes, recorda-se que anteriormente foi citado a existência de diversas implementações de "serviço geral de nomes" ou Naming Service (Voyager Federated Directory Service, CORBA Naming Service, JNDI, Microsoft Active Directory e RMI Registry) as quais permitem a associação de nomes aos objetos para uma referência futura. Pode-se, a partir dos conceitos arrolados por tais implementações, idealizar a inserção dos agentes móveis AFC (Agentes Facilitadores de Conexão) apoiando e/ou aperfeiçoando a composição do serviço de localização de objetos. Visto que todas "instâncias" dos objetos possuem registros monitorados pelo Agente Supervisor, este último pode facilitar a localização e conexão entre uma aplicação cliente e as

“implementações” dos respectivos objetos. Mas, além de tais facilidades, o agente móvel AFC visa o aperfeiçoamento dos mecanismos de localização adicionando restrições ao estabelecimento de conexão dependendo de variáveis de “status” destes objetos.

Tal agente migraria por entre as agências (hosts) do ambiente consultando os agentes supervisores e analisando variáveis que informem o “status” atual de todos objetos instanciados localmente. Por exemplo, imaginemos que uma restrição importante a ser analisada dependa exclusivamente do número total de conexões estabelecidas para um determinado objeto. Supondo que dois objetos instanciados em “hosts” diferentes implementem o mesmo serviço apresentando, em certo momento, montantes desiguais quanto ao número de conexões. O agente AFC vinculado àquele tipo de objeto rastreia periodicamente a situação e “indica” os endereços mais apropriados, segundo a restrição imposta, armazenando tais informações com o Agente Supervisor.

Não se afasta a hipótese de se complementar a eficácia do serviço permitindo-se que o agente avalie outras variáveis de restrição - tráfego na rede, utilização dos recursos computacionais, “custo-distância” entre o cliente e o objeto, etc - além de verificar apenas o número de conexões.

7.7.1.7 - Agente Básico de Performance (ABP)

A missão fundamental desse agente móvel resume-se à produção de dados que permitam uma interpretação aproximada para alguns critérios simples de performance para a aplicação (o Serviço de Monitoração, no caso). Em outras palavras, por exemplo e definição, uma instância da classe do agente ABP realiza as seguintes tarefas:

a) Em intervalos regulares, o agente migra orientando-se através de um itinerário definido por uma “entidade” superior (agente supervisor) ou pelo próprio agente;
b) Assim que alcança cada agência (host), acessa uma faceta apropriada do agente supervisor e registra o tempo gasto a partir da agência origem até o presente local;
c) Inicializa um “timer”;
d) Efetua algumas operações padronizadas pela faceta;

e) Registra o tempo para conclusão das operações;

f) Migra para a próxima agência (host) e repete os passos de **b** a **e** até voltar à agência (host) de origem;

g) Armazena os dados e volta para o passo **a**.

Os dados colhidos pelo agente se resumem a duas medidas: o tempo de "migração" entre dois pontos e o tempo de "interação" com a faceta. A representação de um possível cenário na Figura 7.14, idealiza uma instância do agente ABP representando a agência (host) Patras. Após 10 "ciclos" de migração, o Agente Supervisor tem à disposição um conjunto de informações representando uma evolução, um decréscimo ou a manutenção relativa média das "taxas" armazenadas. Torna-se evidente a necessidade de se conFigurar o número máximo de "ciclos" armazenados evitando um acúmulo excessivo de informações.

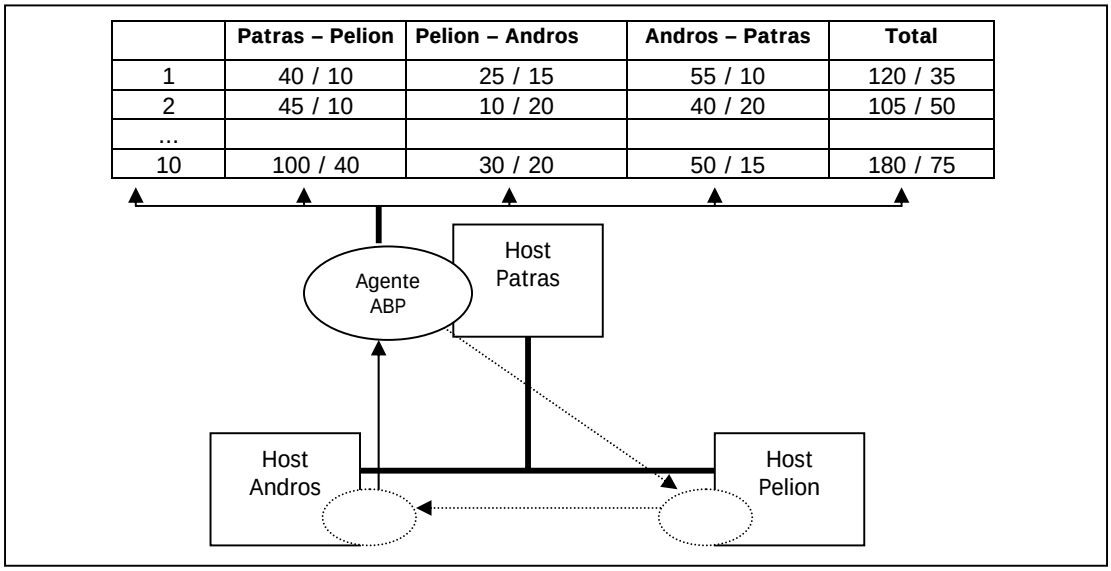

FIGURA 7.14: Agente ABP registrando o tempo gasto nas migrações entre "hosts" .

Ressalta-se que a presente classe não intenta, por enquanto, aprofundar-se na implementação de outros mecanismos, possivelmente mais eficazes, para a tomada de critérios de performance. (ver, em seção posterior, as proposições descritas para o serviço de Avaliação de Performance)

7.7.1.8 - Agente de Produção de Parâmetros (APP)

Na especificação das tabelas que compõem a Base de Dados Global de Objetos (BDGO), diversos parâmetros foram apresentados (desempenho, CPU disponível, I/O disponível, etc.). Naquela oportunidade, comentou-se que todas as variáveis eram "produzidas" pelo Agente Supervisor. Estas "tarefas" podem ser implementadas pelos agentes móveis APP - Agente de Produção de Parâmetros. Os agentes APP migrariam periódica e continuamente através das agências (host) efetivando os procedimentos programados. Após a conclusão da fase de autenticação, acessa o "place" apropriado para produção de parâmetros.

Por exemplo, na determinação do parâmetro *status* da tabela de objetos o agente APP "agiria" da seguinte forma: consulta o Agente Supervisor para recuperar a identificação e endereço (porta) de todos objetos instanciados na agência (host). Enquanto ocorre a transferência de tais dados, o agente tenta obter respectivamente um "reply" individual de cada objeto a fim de confirmar se o mesmo permanece "on line" (ativo) e informações pré-determinadas que permitam avaliar se a disponibilidade "definida" para tais objetos situa-se dentro dos níveis de "normalidade".

De forma semelhante o agente sucessivamente se concentra na produção dos outros parâmetros. A conveniência na utilização do agente móvel neste cenário se evidencia pelo fato de que em todas as agências se repetiriam os mesmos procedimentos. Para uma rede com **N** agências (host), a sobrecarga de se reservar **N** instâncias de uma determinada classe satisfazendo tal funcionalidade é amenizada pela "injeção" de um número consideravelmente menor de agentes APP migrando pela rede e atuando iterativamente em cada agência. Justifica-se também a consideração de que, por definição, os procedimentos de atualização das tabelas de conFiguração ocorrem periodicamente.

Logo, deve-se atentar para a possibilidade de que o cálculo dos intervalos de tempo definidos entre tais operações estará intimamente relacionado com o número mínimo de instâncias do agente APP. Grosso modo, diga-se por hipótese que um agente APP, em

média, gasta 0,1ms para concluir suas tarefas em cada uma das 12 agências da região considerada. O intervalo para atualização dos parâmetros está calibrado em 0,6ms. Desprezando o tempo de migração entre as agências, 3 instâncias do agente APP seriam suficientes garantindo uma margem de tolerância. A Figura 7.15 demonstra graficamente este provável cenário de atuação de alguns agentes APP.

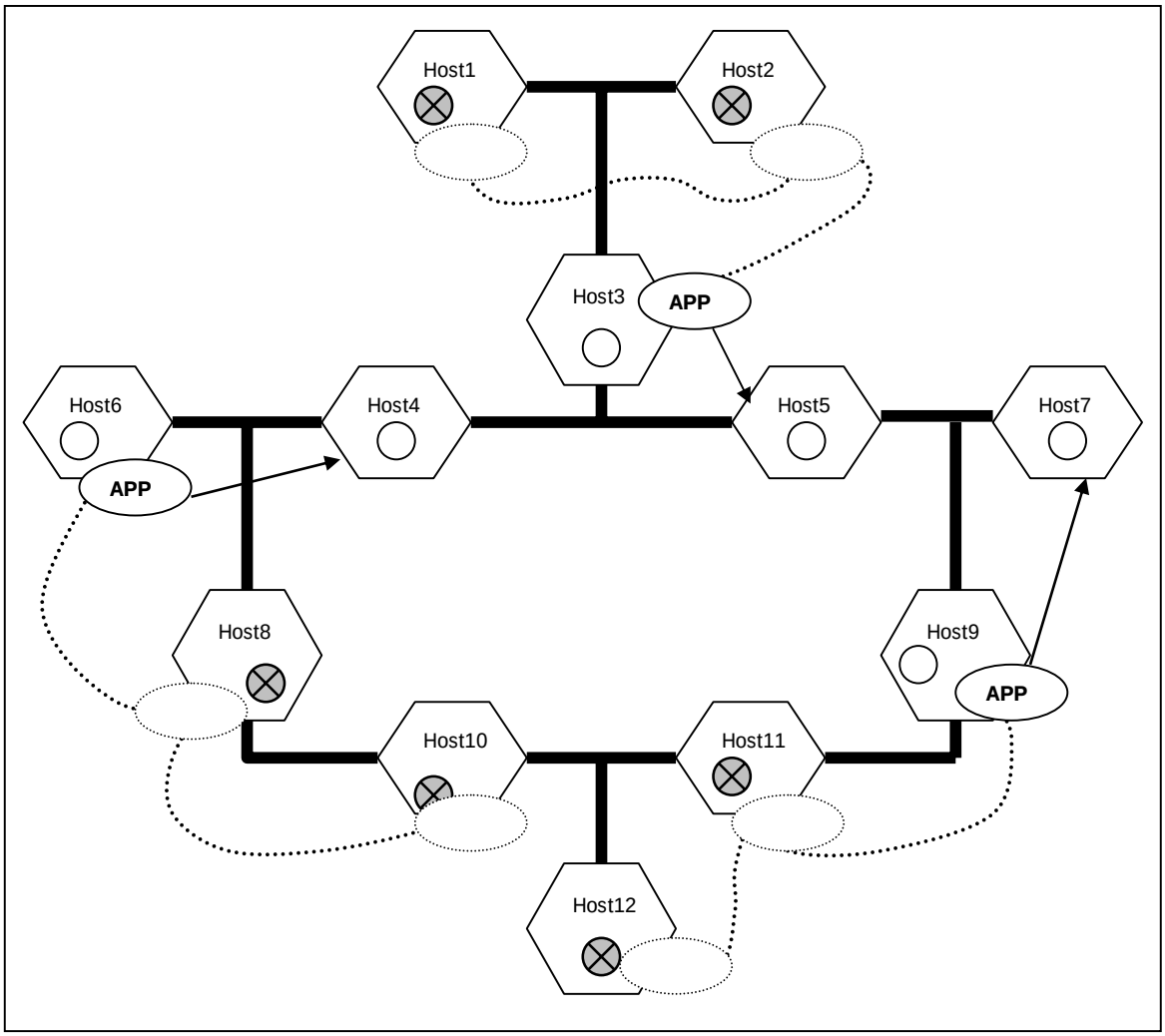

FIGURA 7.15: Instâncias do agentes APP “trabalhando” entre “hosts” de uma região.

CAPÍTULO 8
IMPLEMENTAÇÃO DE PROTÓTIPO PARA O SERVIÇO DE AGENTES

A partir da especificação dos conceitos e da própria arquitetura do Serviço de Agentes, o presente capítulo assume como objetivo a apresentação de um ambiente de teste o qual, permitindo a abrangência dos detalhes de implementação e o esboço de algumas considerações, represente, em linhas gerais, a aplicação prática da tecnologia de agentes móveis ao contexto da proposta SICSD. O protótipo para o serviço de monitoração de objetos participa de um cenário pré-definido para a simulação da arquitetura SICSD como representado na Figura 8.1 abaixo.

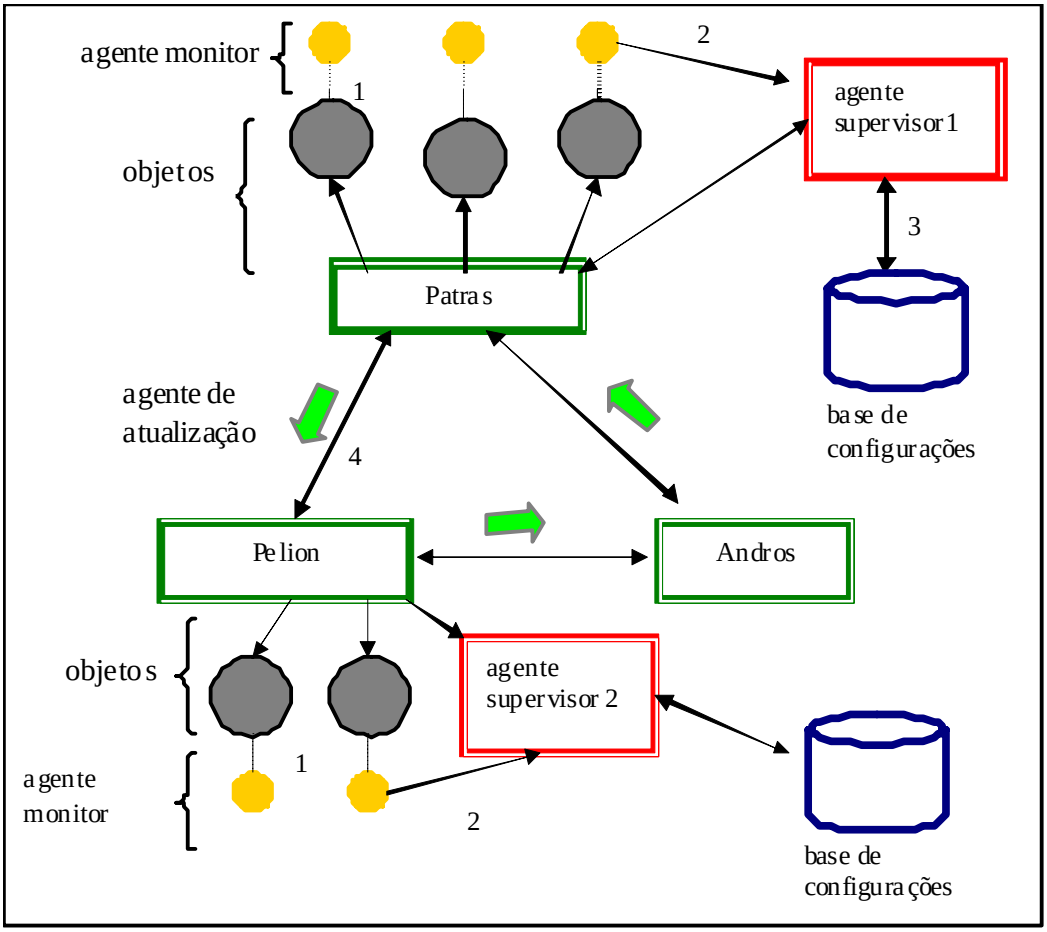

FIGURA 8.1: Cenário de atuação do protótipo do Serviço de Agentes.

Neste cenário, um “simulador” específico cria instâncias dos objetos nos “hosts” disponíveis. O ponto (**1**) representa agentes monitorando o “estado” dos objetos. À

medida que ocorrem alterações, os agentes transferem as respectivas informações para o agente supervisor (**2**) que, por sua vez, armazena as mesmas em base de configurações (**3**). Em paralelo, os agentes de atualização se incumbem de disseminar as informações entre as diversas bases de configurações (**4**). Para melhor compreensão quanto aos detalhes de definição dos agentes, faz-se referência oportuna ao capitulo anterior.

8.1 – O Cenário para implementação do ambiente

O ambiente para o protótipo foi acomodado em 3 máquinas pertencentes à rede local do Departamento de Controle de Satélites, as quais apresentam as seguintes características: 1) "host" **Patras** (Pentium III 266, 64 Mb RAM); 2) "host" **Andros** (Pentium 133, 32 Mb RAM) e 3) "host" **Pelion** (Pentium 133, 32 Mb RAM). Serão listadas, a seguir, as ferramentas de suporte (software) que compuseram o ambiente de desenvolvimento.

8.1.1 - Visibroker for Java version 3.2

Esta ferramenta disponibiliza o "CORBA 2.0 Object Request Broker" (ORB) e suporta o ambiente de desenvolvimento para implementação e gerenciamento de aplicações com objetos distribuídos. A simulação dos serviços[1] do software de controle de satélites (telemetria, telecomandos, etc.) foi elaborada através da concepção de objetos com o Visibroker for Java. Os detalhes referentes ao simulador não pertencem ao escopo da presente dissertação.

8.1.2 - Symantec Visual Café PDE version 2.0 for Java

Symantec Visual Café é o primeiro ambiente RAD ("*Rapid Application Development*") para criação de applets "web-hosted"[2] e aplicativos independentes Java. Oferece um conjunto integrado de ferramentas para desenvolvimento em Java, incluindo gerenciador de projetos, "form designer", biblioteca de componentes, editor de código, "class browser" e um "debugger" gráfico integrado a um compilador Java e um "applet viewer"

[1] ver no capítulo 2 a descrição dos serviços do software de controle de satélites
[2] applets acomodadas em web browsers

8.1.3 - Java Development Kit - JDK 1.1.7

Classifica-se como tecnologia de suporte necessária à execução de aplicativos Java. O ambiente de execução Java ("*Java runtime environment*") assume a responsabilidade de interpretar e executar um aplicativo Java. Em geral, este "módulo básico" se encontra embutido em ambientes de desenvolvimento Java mais robustos (**JBuilder** da Inprise, por exemplo).

8.1.4 - Voyager Core Technology 2.0.2 (ObjectSpace)

Especificamente durante a fase de implementação do protótipo para o Serviço de Agentes, dentre as "candidatas" colocadas em evidência pela comparação desenvolvida em capítulo[3] anterior, a opção por uma das plataformas, Grasshopper ou Voyager, sofreu influência, na prática, de vários fatores adicionais.

A facilidade de acesso e interação no site da ObjectSpace quanto aos procedimentos necessários para obtenção gratuita da plataforma Voyager estabeleceu um primeiro contato positivo. Além de disponibilizar mecanismos de notificação automáticos para atualização de novos lançamentos do produto, a empresa mostrou agilidade, através de um canal de comunicação privado, no acompanhamento de sugestões submetidas e na solução de problemas e dúvidas de suporte encontrados. O site também ofereceu um serviço de fóruns (lista de discussão[4], grupos de notícias – "*news groups*"- e listas de notificação – "*notification mailing lists*".) específico ao produto que serviu como via de acesso às notícias relevantes e como meio de compartilhamento de informações entre os usuários da plataforma.

Em contrapartida, além de oferecer uma versão "trial" da plataforma Grasshopper com recursos muito limitados, o site da IKV++ levantou barreiras de

[3] capítulo de comparação de plataformas
[4] a participação efetiva dos usuários e pesquisadores da plataforma contribuiu para a grande variedade de tópicos discutidos

acesso a versões mais completas do produto e dificilmente respondia as solicitações e questões enviadas.

A documentação de apoio, apresentando um excelente nível técnico de informações, foi exaustivamente consultada durante o estágio de instalação e configuração de ambas as plataformas. Em obediência à estrutura definida para o protótipo do Serviço de Agentes, foram instaladas e configuradas, em cada uma das três máquinas ("host") que participaram do ambiente, cópias individuais dos respectivos "pacotes" de aplicação. A interface gráfica diferenciada não disfarçou as primeiras restrições da plataforma Grasshopper que, nesta versão de teste, determinavam um limite máximo para o número de agências e agentes que podiam ser criados e manipulados. A bordo de uma interface comparativamente simplificada, a plataforma Voyager não impôs limitações deste gênero.

A quantidade variada de exemplos práticos foi um fator preponderante de realce aos méritos do Voyager quanto à compreensão e à consequente assimilação das características funcionais e dos tipos de aplicações suportadas pela plataforma. Não restrita a aspectos de desenvolvimento de soluções baseadas em agentes, a gama de exemplos se diversificou na abordagem de situações intimamente ligadas à composição de serviços em ambientes de objetos distribuídos. A plataforma estabeleceu detalhes claros sobre a execução, os comandos e os resultados esperados de cada exemplo. Além disso, a manipulação e adaptação descomplicadas do código fonte listado e subdividido em categorias[5] competentes permitiram a absorção gradual e progressiva da maneira peculiar de se implementar aplicativos com a ferramenta.

Enquanto a plataforma Grasshopper se restringiu à apresentação escassa e insuficiente de aplicações modelo que, além de se resumirem à simples migração de agentes e a tópicos de comunicação, impossibilitavam experimentos complementares de interação com o código fonte e com as funções e serviços da ferramenta; o Voyager exemplificou minuciosamente abordagens para transferência de mensagens, construção remota de objetos, "naming service"[6] e carga de classes remotas; criação e acesso remoto de facetas; utilização de formas mais avançadas de mensagens; atribuição da

[5] Os exemplos fornecidos pela plataforma foram dividas em categorias: Basics, Messaging, Mobility, Agents, ..., etc. Ver referência ao Guia do Usuário.
[6] serviço que "amarra" (bind) nomes aos objetos para futuras referências (por nome). Ver capítulo de destaque da plataforma Voyager

faceta de mobilidade para qualquer objeto serializável; atribuição da faceta de agente para que qualquer objeto serializável se tornasse um agente móvel e autônomo; manipulação do sistema de diretórios federados Voyager; integração com objetos CORBA; entre outros exemplos.

Enfim, através dos fatores recolhidos e analisados imparcialmente até o início do trabalho de implementação, a plataforma Voyager se estabeleceu como ferramenta mais adequada, na prática, para o desenvolvimento e codificação do protótipo para o Serviço de Agentes.

8.2 – A Hierarquia das Classes na Composição do Serviço

De acordo com a estrutura hierárquica para as classes de agentes definida na especificação do Serviço de Agentes, a composição do serviço de monitoração de objetos é representada pelo diagrama na Figura 8.2 abaixo:

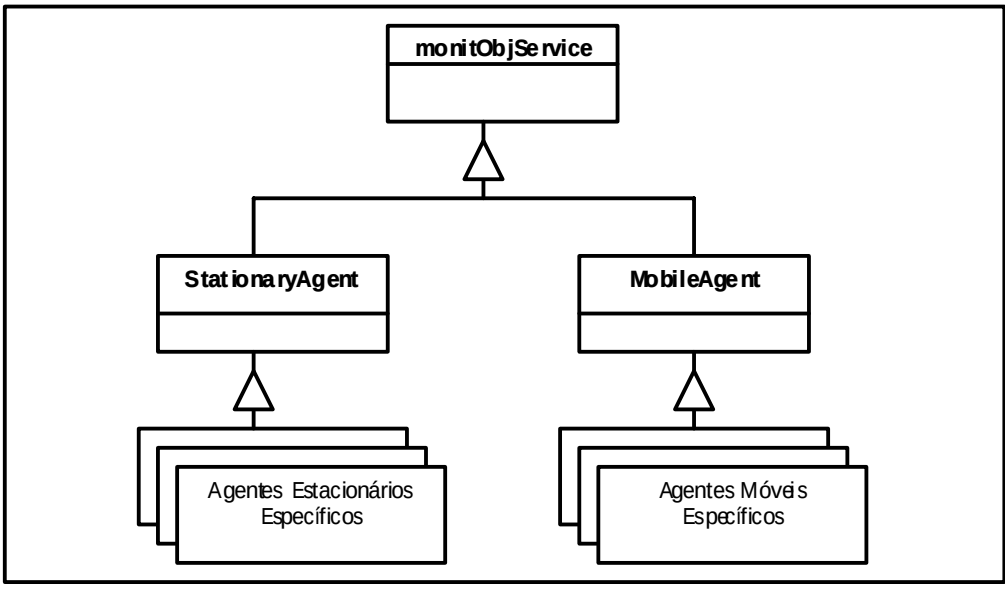

FIGURA 8.2: Hierarquia das Classes de Agentes para o serviço de monitoração.

Este modelo de hierarquia de classes, concebido a partir da funcionalidade fornecida pela plataforma Grasshopper através das classes pré-definidas **Service**, **mobileAgent** e **stationaryAgent**, não foi abordado na implementação do ambiente de teste atual visto que a plataforma Voyager não prevê uma estrutura de classes diferenciadas para agentes móveis e estacionários. Portanto, para a concepção do

protótipo do Serviço de Agentes, a atenção principal será focalizada na classe de agentes estacionários específicos e agentes móveis específicos (referindo-se, no caso, à hierarquia apresentada na parte teórica). O diagrama de classes apresentado abaixo na Figura 8.3 representa, portanto, o modelo básico para as classes do protótipo implementado. Ressalta-se, porém, que o diagrama não retrata, em particular, o relacionamento com as classes que compõem a simulação da arquitetura SICSD.

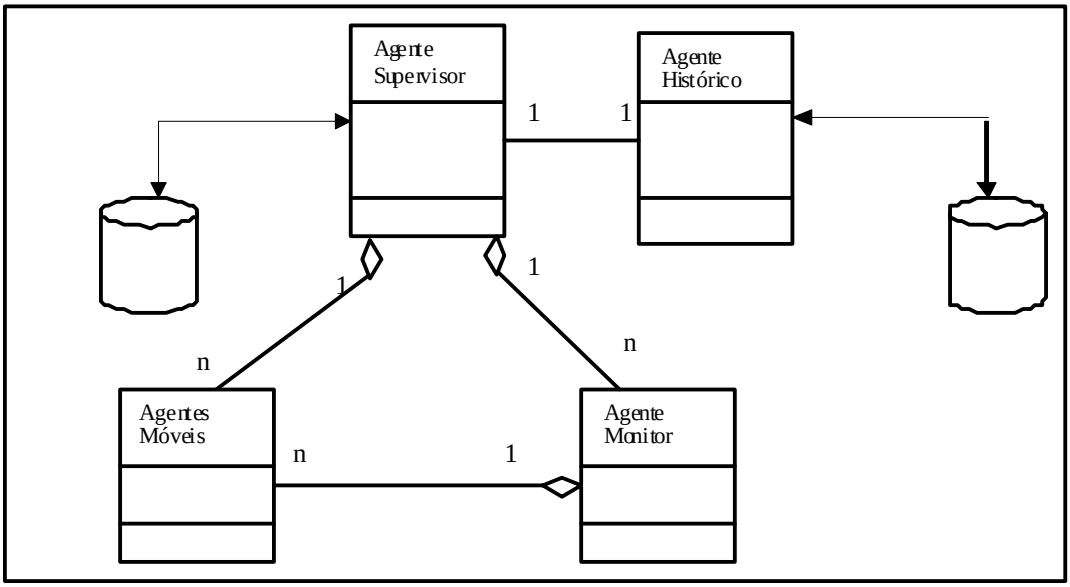

FIGURA 8.3: Diagrama de classes para o protótipo do Serviço de Agentes.

Pelo panorama exposto na figura acima, define-se como abstrações para os agentes estacionários específicos as classes Agente Supervisor, Agente Histórico e Agente Monitor. A abstração para os agentes móveis específicos é definida pela classe Agentes Móveis. Todo código fonte referente às classes introduzidas a seguir foi reunido à parte em apêndice específico.

8.2.1 - As classes dos Agentes Estacionários Específicos

Para a composição do serviço de monitoração, foram consideradas, inicialmente, três classes de agentes estacionários representando respectivamente a associação aos componentes fixos (agentes) definidos no capítulo anterior da seguinte forma:

Componente	Classe Associada	Sigla
Agente Supervisor	SupAgente.class	AS

Agente Histórico	HistAgente.class	AH
Agente Monitor de Objetos	MonitAgente.class	AMO

8.2.1.1 – O Agente Supervisor

Neste tópico, além da classe **SupAgente** que representa o processo (atuará como servidor) de inicialização, serão introduzidas as classes **Comp**, **CompTable**, **Account**, **Security** e **Clock** que complementam o cenário de atuação do Agente Supervisor. Destaca-se, de antemão, a utilização do mecanismo de agregação dinâmica[7] através do qual se define um agregado entre a classe primária **CompTable** e suas facetas - representadas pelas respectivas classes **Account**, **Security** e **Clock**.

O Agente Supervisor tem como tarefa primordial o "gerenciamento" sobre as tabelas definidas dentro da BDGO. A primeira questão levantada, então, se resumiu em decidir como as informações das tabelas seriam armazenadas. Não fazendo parte das intenções do trabalho uma simulação otimizada e robusta, nem mesmo a utilização, por enquanto, de um sistema de banco de dados distribuído, o armazenamento dos dados admite "ferramentas"- estruturas de dados básicas - que não afastem, ao extremo, a validade do protótipo da aplicação.

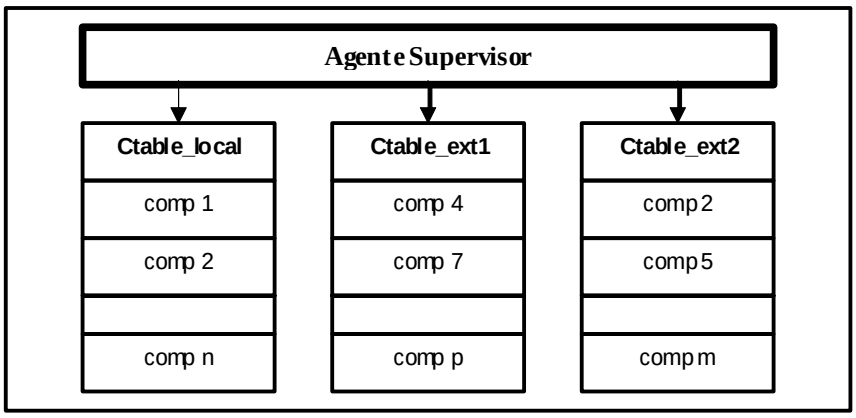

FIGURA 8.4: Agente Supervisor e respectivas tabelas.

Cada tabela de informação possui registros contendo atributos específicos (a tabela de objetos, por exemplo: nome, endereço IP, número de conexões e status). A abstração para as tabelas foi associada, portanto, aos conceitos de definição e estruturação de operações de um vetor a partir da classe **Vector** do Java. Ou melhor

dizendo, cada tabela será representada por um vetor de estruturas (registros), onde a classe **CompTable** abstrai o conceito de tabela e a classe **Comp** abstrai o conceito de registros.

Em resumo, uma instância da classe **CompTable** pode guardar uma coleção de instâncias da classe **Comp**. O Agente Supervisor, durante a inicialização do sistema, criará instâncias da classe **CompTable**. Oportunamente, pode se comentar que o Agente Supervisor responsável por (fixo em) um "host" específico determina uma instância da classe **CompTable** relacionada ao "host" onde será instanciada e **n-1** instâncias "externas" associadas aos **n-1 "**hosts externos" que compõem o ambiente. Ou seja, o Agente Supervisor determinará uma estrutura apontando para o "host" local e **n-1** "instâncias-espelho" direcionadas para os "hosts" restantes. A Figura **8.4** ilustra a abordagem sugerida.

Os métodos definidos dentro das classes **CompTable** e **Comp** estipulam os mecanismos apropriados para controle e manipulação das informações. Uma vez agregada à classe **CompTable**, a faceta de segurança (classe **Security**) pode implementar, através de métodos específicos a esta classe, os processos quanto ao controle de acesso dos agentes às informações do Agente Supervisor, controle das variáveis de "status" definidas para o "host" local e conferência de "passwords". De uma perspectiva ampla, para fins de hipótese, a faceta pode permitir a abordagem de diversos tópicos concernentes à segurança local do Agente Supervisor em relação às tabelas. A implementação atual apenas considera tais questões como ilustração do potencial da agregação dinâmica para trabalhos futuros.

Do mesmo modo, a agregação da faceta representada pela classe **Clock** à estrutura da classe **CompTable**, visa ilustrar a composição "modesta" de um mecanismo de sincronização para um "clock" interno da aplicação. Tal sincronização foi simulada através de um processo configurado pela classe **TimeStamp**, a qual será descrita posteriormente na seção 8.2.3. A classe **Account,** agregada como faceta, permite a criação de métodos através dos quais se disponibilizaria modelos funcionais mínimos para que os agentes móveis "prestem contas" a respeito dos resultados de suas tarefas ou efetuem operações para levantamento de parâmetros.

[7] O conceito de Agregação Dinâmica foi introduzido no capítulo 6 (Destaques do Voyager)

Encerrando o “cenário” das noções iniciais sobre o Agente Supervisor, destaca-se a classe **SupAgente**. Associada, na apresentação dos agentes estacionários específicos, como abstração do respectivo agente, esta classe atua funcionalmente como um processo servidor que coloca “no ar”, em um “host” determinado, uma instância do agente em questão. Em outras palavras, esta classe inicializa como um servidor Voyager numa determinada porta de comunicação e cria as instância da classe **CompTable** associadas tanto ao “host” local como aos outros “hosts” remotos que compõem o sistema. Visto que no ambiente em que se implanta o protótipo assume-se um número determinado e conhecido de máquinas (três), as associações são pré-determinadas.

Vale, porém, considerar a possibilidade de se complementar o cenário reproduzido. Por exemplo, a partir do momento em que ocorre a “inicialização” da classe **SupAgente**, a mesma dispara agentes móveis que, migrando para todos os nós participantes no ambiente à procura de referências que evidenciem a existência de outros Agentes Supervisores, sinalizam a necessidade de criação das tabelas externas.

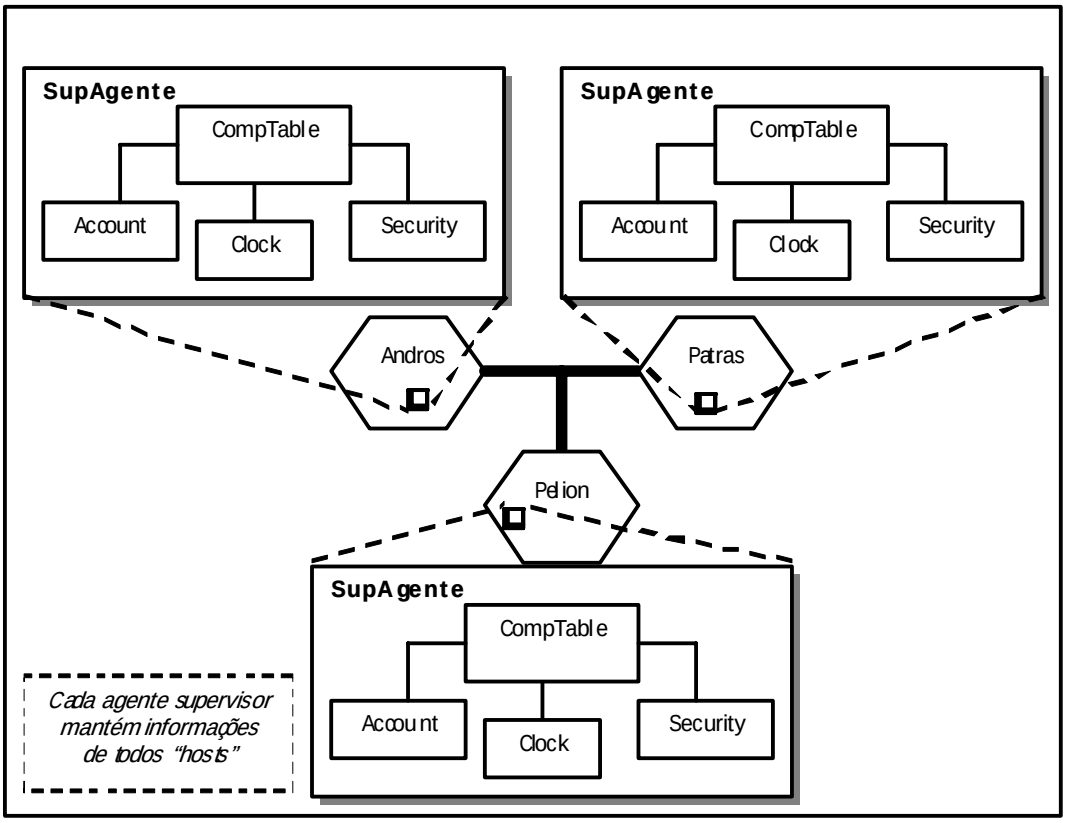

FIGURA 8.5: Cenário configurado para o Agente Supervisor a partir da classe SupAgente.

A Figura 8.5 esboça um resumo geral da especificação das classes intrinsecamente relacionadas à estrutura do agente supervisor e os “papéis” por elas

desempenhados no Serviço de Agentes. A classe **Comp** representa a estrutura básica para armazenar informações dos objetos. A classe **CompTable** representa o vetor de armazenamento da "coleção de objetos" da classe **Comp**. A agregação entre a classe **CompTable** e as facetas (classes **Account**, **Security** e **Clock**) formam o agregado que abstrai o conceito do agente estacionário responsável pelo controle de tabelas (local e externas). Por fim, a classe **SupAgente** representa o processo servidor que instancia o agregado.

8.2.1.2 - O Agente Histórico (AH)

A presente seção introduz as seguintes classes associadas à composição do agente histórico: **HistAgente**, **History** e **Line**. A partir das definições do cenário de sua atuação estabelecidas no capítulo anterior, a especificação das classes para o agente histórico segue, em resumo, o mesmo padrão das estruturas definidas para o agente supervisor. Responsável pelo armazenamento "temporário" de informações até que as mesmas sejam disseminadas, o agente histórico se restringe ao contexto local de apenas um "host".

Portanto, devido às similaridades herdadas das classes concebidas para o agente supervisor, define-se que a classe **History** está para a classe **Line** assim como a classe **CompTable** está para a classe **Comp**. Sendo mais uma vez utilizado o vetor como estrutura de dados satisfatória, uma instância da classe **History** pode armazenar uma coleção de instâncias da classe **Line**. Estabelecidas as estruturas de armazenamento "temporário" – classes **History** e **Line –** e definidos os métodos de manipulação das informações, idealizou-se a composição da classe **HistAgente**.

Pode se observar, através do código inicialmente implementado, que a classe **HistAgente** cria as instâncias dos agentes históricos em todas as máquinas - "host" – nas quais foram inicializadas instâncias do agente supervisor. A implementação do ambiente atual pressupõe que as operações[8] que alteram de alguma forma o estado dos objetos monitorados devem ser armazenadas através das instâncias dos agentes históricos. Em situações determinadas ou intervalos programados, agentes móveis

[8] definidas na simulação do ambiente SICSD. A primeira operação estipulada foi a monitoração das conexões aos objetos.

específicos se encarregam do processamento, atualização e disseminação apropriados dos registros "estocados".

Assim que os procedimentos previstos garantam o cumprimento perfeito da "missão" dos agentes móveis, os respectivos registros "temporários" são eliminados. Destaca-se, mais uma vez, a utilização do mecanismo de agregação dinâmica ao se adicionar à classe **History** uma faceta de segurança (classe **Security**), através da qual sugere-se como ilustração, entre outros, um mecanismo de verificação de permissões para o acesso dos agentes móveis.

Finalmente, pode-se verificar que o código da classe **HistAgente** modela um processo que permite a ativação das instâncias do agente a partir de qualquer "host" do ambiente. Ressalta-se que esta "criação" remota define como pré-requisito vital a configuração antecipada de portas[9] ativas e reservadas para "acomodação" destes agentes. Numa possível versão otimizada, sugere-se em caráter preventivo que, antes da inicialização dos agentes históricos, sejam delegados agentes móveis que migram verificando a existência das portas nas quais podem ser "instalados" os processos remotos.

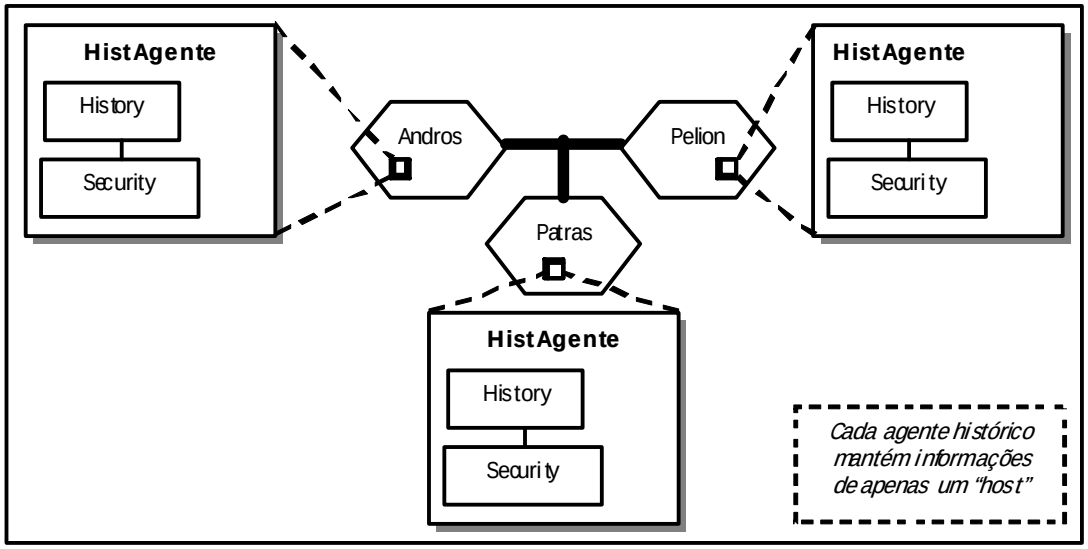

FIGURA 8.6: Cenário configurado para o agente histórico a partir da classe HistAgente.

A Figura 8.6 esboça um resumo geral da especificação das classes relacionadas à estrutura do agente histórico e os "papéis" por elas desempenhados no Serviço de

[9] servidor Voyager que "aceita" objetos e mensagens de outros programas Voyager. [Voyager 04], pág. 7

Agentes. A classe **Line** representa a estrutura básica para armazenar dados referentes às operações realizadas nos objetos. A classe **History** representa o vetor de armazenamento da "coleção de objetos" da classe **Line**. A agregação entre a classe **History** e a faceta de segurança (classe **Security**) forma o agregado que abstrai o conceito do agente estacionário responsável pelo controle do histórico local. A classe **HistAgente** ativa o processo servidor que instancia o agregado.

8.2.1.3 - O Agente Monitor de Objetos (AMO)

Concluindo a "família" dos agentes estacionários específicos, define-se a composição do agente monitor de objetos. A classe **MonitAgente** abstrai o modelo idealizado para o respectivo agente. Convém, de antemão, ressaltar que o código implementado para a classe não compreendeu todas as propriedades idealizadas teoricamente na especificação do agente no Capítulo 7, e que a atual abordagem foi sugerida, em caráter provisório, em virtude da necessidade de se concluir o primeiro protótipo para o Serviço de Agentes.

Deste modo, a classe se restringe a compor o serviço de consultas ao agente histórico em busca de registros associados ao objeto para o qual foi designada a monitoração. Como exemplificado no código da classe , pode-se destacar alguns pontos que retratam o contexto básico do agente monitor.

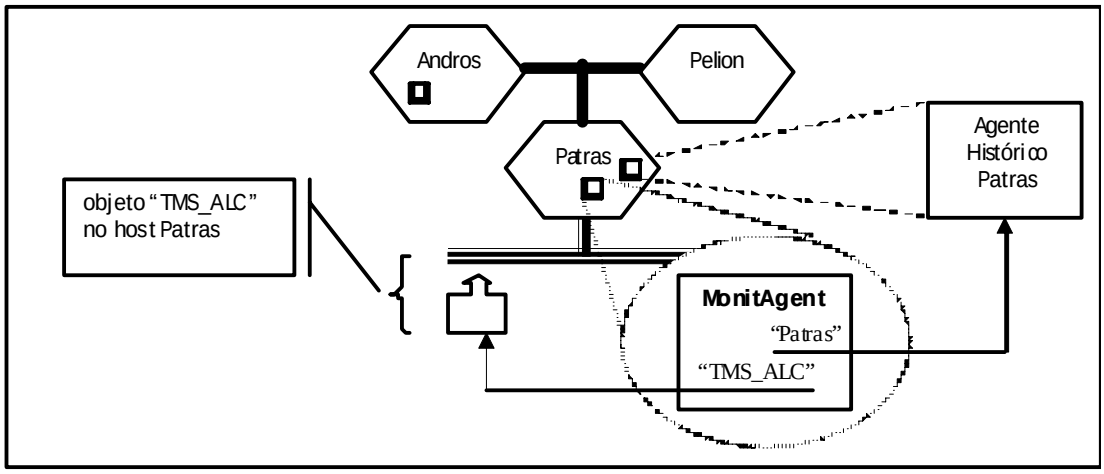

FIGURA 8.7: Agente, monitorando objeto "TMS_ALC", consulta o agente histórico.

Na inicialização do agente, são pré-definidos os parâmetros *nome_do_objeto* e a *origem_da_solicitação*. O *nome_do_objeto* refere-se a um novo objeto (por exemplo,

serviço de telemetria da arquitetura SICSD) criado e atuando um determinado “host”, cujo endereço, na forma do parâmetro *origem_da_solicitação*, representa o agente supervisor do “host”. Possuindo tais informações, o agente monitor estabelece um “canal de comunicação”, ou seja, obtém uma referência do respectivo agente histórico. Ou seja, se a origem da solicitação está associada ao “host” Patras, o agente monitor deve monitorar o histórico associado ao mesmo “host” (o agente histórico do “host” Patras).

Com este vínculo, o agente monitor procura, em intervalos definidos, por “registros” do agente histórico que se refiram ao objeto monitorado. Caso existam tais registros, o agente monitor se encarrega de instanciar os agentes móveis (classe **TraderHist** apresentada em seção posterior) para disseminação e atualização dos dados para as tabelas (locais e externas) do respectivo agente supervisor. A Figura 8.7 sintetiza o cenário configurado no protótipo do Serviço de Agentes. Um objeto aplicação (“TMS_ALC”) responsável pelo serviço de telemetria na arquitetura SICSD foi carregado no “host” Patras. Em seguida, designou-se uma instância do agente monitor que, para rastrear as conexões ao objeto, acessa e consulta o agente histórico do “host” Patras.

8.2.2 – As classes dos Agentes Móveis Específicos

Até o momento, foram abordados os principais tópicos referentes às classes implementadas dos agentes “servidores” fixos. Em outros termos, portanto, não se afasta a idéia de que o agente supervisor, o agente histórico e o agente monitor firmam-se como processos de serviços locais de suporte (ou “places”) onde os agentes móveis efetuarão suas tarefas. Visto que foram apresentados os tipos de classes para abstração das estruturas de armazenamento e controle de informações, resta, em sequência, transferir a argumentação para os pontos associados à composição dos agentes móveis específicos. O maior desafio se apresentou na necessidade de reconhecer e compreender os mecanismos da infra-estrutura para implementação do conceito de agentes móveis e autônomos disponibilizados pela plataforma Voyager. Ou seja, “*como adaptar os modelos de agentes móveis para o serviço de monitoração de objetos à metodologia de desenvolvimento da plataforma?*”.

Em primeiro plano, ficou explícita a ênfase da plataforma sobre os aspectos de mobilidade e autonomia de execução para o agente durante as experiências com o exemplo **Agents1** e com o respectivo código fonte. Visto que o padrão de implementação dos diversos agentes móveis (nomeados, oportunamente, como "traders") bem como da estrutura de relacionamento destes com os agentes estacionários foi totalmente adaptado segundo os moldes deste exemplo, torna-se adequada a dissecação dos pontos essenciais e úteis à codificação.

8.2.2.1 – Conceitos de "Codificação" do Voyager para Agentes Móveis

No cenário construído, esboçado na Figura 8.8, a partir do exemplo **Agents1**, presenciou-se os atributos da execução de um agente móvel e autônomo (representado pela classe **Trader** e pela respectiva interface **ITrader**) trabalhando num mercado de ações remoto (representado pela classe **StockMarket** e pela respectiva interface **IStockMarket**) enquanto a classe **Agents1** desempenhou as funções de um processo que administrava o desenrolar de atividades específicas ao ambiente proposto.

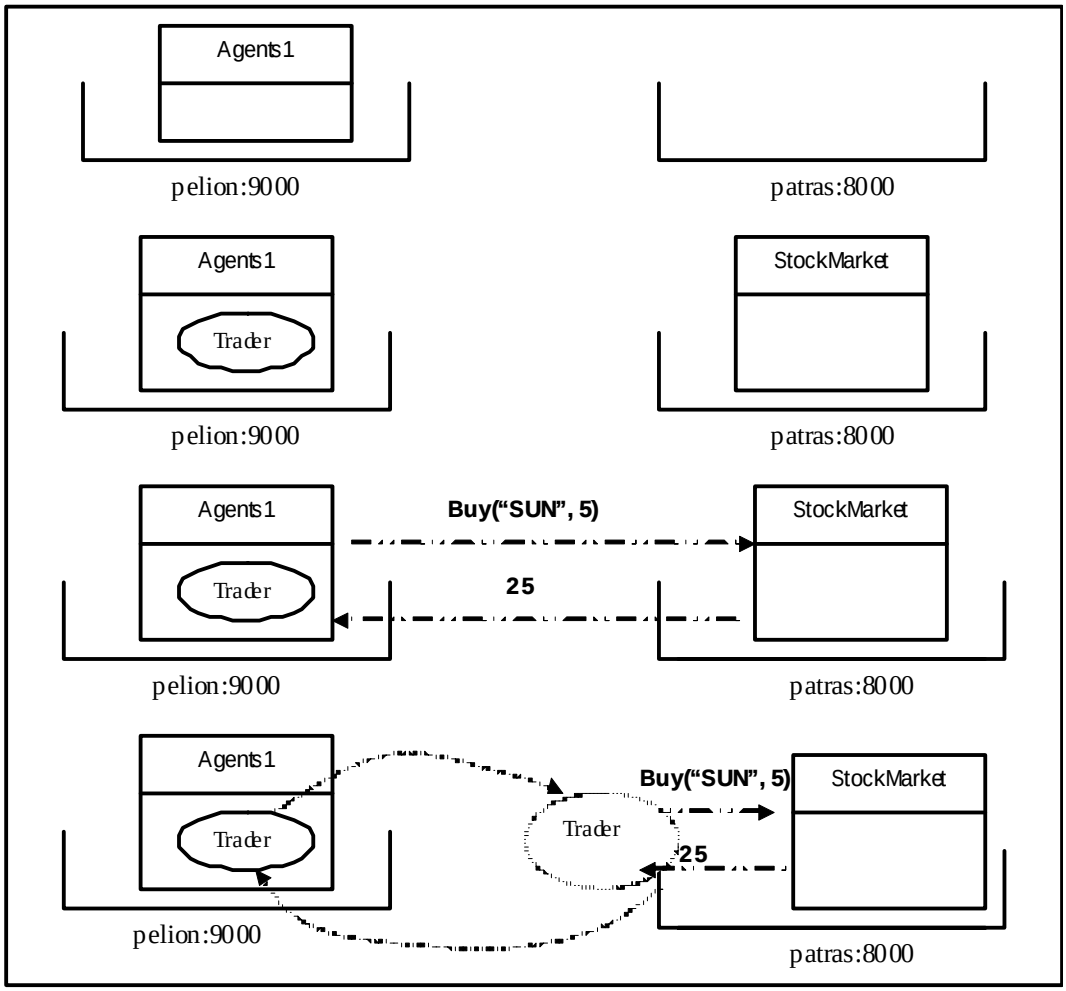

FIGURA 8.8: Representação do exemplo Agents1.
FONTE: Voyager (1998).

Para melhor assimilação do papel desempenhado pelas classes **ITrader** e **IStockMarket**, faz-se referência à conceituação apresentada no Capítulo 6 quanto ao uso de interfaces para computação distribuída. O diagrama esboçado pela Figura 8.9 resume e identifica os relacionamentos principais entre as classes **Agents1**, **Trader** e **StockMarket**:

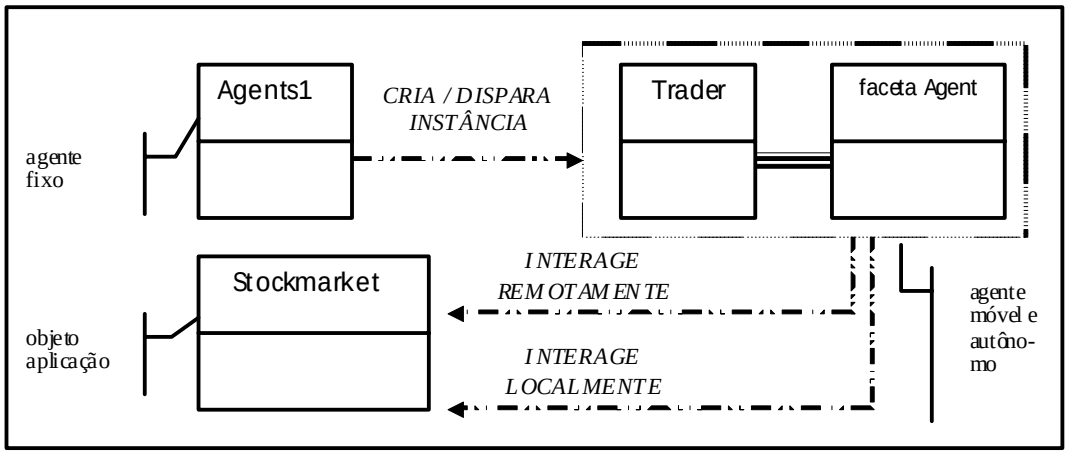

FIGURA 8.9: Relacionamento entre as principais classes do exemplo Agents1.

À medida que os comentários mais detalhados sobre o exemplo são organizados, o respectivo código fonte poderá ser consultado contribuindo para o reforço aos pontos chave em destaque. A porção de código listada abaixo se refere à classe **Agents1**.

Program voyager2.0.0\examples\agents\Agents1.Java

```
...
public class Agents1
 {
 public static void main( String[] args )
  {
  try
   {
   Voyager.startup();
   // create remote stockmarket
   String marketClass = Stockmarket.class.getName();
   IStockmarket market = (IStockmarket) Factory.create( marketClass, "//patras:8000" );

   // create trader agent
   ITrader trader = (ITrader) Factory.create( Trader.class.getName() );
   // trade from local machine, then trade on remote machine
   trader.work( market );
   }
  catch( Exception exception )
   {
   System.err.println( exception );
   }
  Voyager.shutdown();
  }
 }
```

A classe **Agents1** representa o "programa principal" que inicializa o ambiente para o exemplo considerado. O ponto assinalado destaca o momento em que se cria remotamente uma instância da classe **Stockmarket** na porta 8000 do "host" Patras. Logo após, se confere no ponto a criação de uma instância da classe **Trader**. Através do método definido na interface **ITrader** (*void **work(** IStockmarket market)*), "dispara-se" a execução do "trader" definido. Transfere-se, então, o foco de concentração para o código da classe **Trader** listado abaixo:

Classe voyager2.0.0\examples\agents\Trader.java

```
...
public class Trader implements ITrader, Serializable
 {
  public Trader()
  {
    System.out.println( "construct trader" );
  }
  public void finalize()
  {
    System.out.println( "finalize trader" );
  }
  public void work( IStockmarket market )
  {
    System.out.println( "remote trade" );
    tradeAt( market ); // trade with remote market
    System.out.println( "local trade" );

    try
      {
      Agent.of( this ).moveTo( market, "atMarket" ); // move to market
      }
    catch( Exception exception )
      {
      System.err.println( exception );
      }
  }

  public void atMarket( IStockmarket market )
  {
    System.out.println( "at remote market, home = " + Agent.of( this ).getHome() );
    tradeAt( market ); // trade with local market
    Agent.of( this ).setAutonomous( false ); // allow myself to be gc'ed
  }
  private void tradeAt( IStockmarket market )
  {
    System.out.println( "start trade" );
    Stopwatch watch = new Stopwatch();
    watch.start();
    for( int i = 0; i < 1000; i++ ) // do 1000 trades
       market.buy( 100, "SUN" );
    watch.stop();
    System.out.println( "stop trade" );
    System.out.println( "time = " + watch.getTotalTime() + "ms" );
  }
}
```

Reportando-se ao final do último parágrafo de comentários, encontra-se a invocação do método **work()** cujo parâmetro "aponta" para um objeto representado pela interface **IStockmarket**. O objeto "trader" adquire a referência para o objeto **StockMarket**. O ponto assinalado demarca o início da tarefa de "comercialização" desempenhada remotamente pelo "trader"[10] com o objeto **Stockmarket**. Após a interação remota, o ponto destaca a "tentativa" de o "trader" se mover para o local onde o objeto **Stockmarket** foi instanciado. Desenvolve-se, a seguir, toda situação implícita no comando "**Agent.of(this).moveTo(market, "atMarket")**".

"**Agent.of(this)**" representa a utilização da agregação dinâmica. A partir da invocação deste método, atribui-se ao objeto "trader" a faceta definida na classe **Agent** do Voyager. Em outras palavras, o objeto "trader" se agrega ao conceito de agente podendo utilizar os métodos definidos em **IAgent.** Sob esta nova perspectiva, o objeto "trader" assume a "credencial" de um agente móvel e autônomo. Como tal, faz uso do método **moveTo()** com o parâmetro "***market***" definindo o objeto para o qual tenta migrar e, ao obter sucesso neste movimento, reinicia suas atividades a partir do método representado pelo segundo parâmetro "***atMarket()***". Os detalhes dos mecanismos de agregação dinâmica e de mobilidade podem ser revistos no Capítulo 6.

O ponto retrata o ponto em que o agente "trader" retoma o controle de suas atividades logo após migrar para o objeto remoto. Inicia então, localmente, a tarefa de "comercialização" através do método **tradeAt()**. Tendo concluído, com êxito, sua missão, o agente "permite", no escopo do ponto , que seja recolhido pelo "garbage collector" utilizando o método **setAutonomous()**.

Como evidenciado no ponto , o método **tradeAt()** modela a tarefa de "comercialização". O agente cria um "timer" utilizado para cronometrar suas atividades enquanto acessa os métodos definidos na interface **IStockmarket**. Para fins de ilustração se notifica uma execução de experiência do exemplo **Agents1** cujos tempos permitem uma comparação clara e adequada, quanto à eficiência, entre interações locais e remotas. Realizando a interação remotamente em **4917** ms, o agente "trader" repetiu as mesmas operações, localmente, em **380** ms.

[10] evita-se referir o objeto como agente pois o mesmo, ainda, não se agregou à respectiva faceta **Agent**.

O código das classes **ITrader**, **IStockMarket** e **StockMarket** é listado abaixo:

Interface voyager2.0.0\examples\agents\ITrader.java

```
...
public interface ITrader
 {
 void work( IStockmarket market );
 }
```

Interface voyager2.0.0\examples\stockmarket\IStockmarket.Java

```
...
public interface IStockmarket
 {
 int quote( String symbol );
 int buy( int shares, String symbol );
 int sell( int shares, String symbol );
 void news( String announcement );
 }
```

Classe voyager2.0.0\examples\stockmarket\Stockmarket.java

```
...
public class Stockmarket implements IStockmarket, Serializable
 {
 static Random random = new Random();  private Hashtable prices = new Hashtable();

public Stockmarket()
   {
   System.out.println( "construct stockmarket" );
   }

 public int quote( String symbol )
   {
   Integer price = (Integer) prices.get( symbol );
   if( price == null )
     {
     price = new Integer( Math.abs( random.nextInt() ) % 100 + 20 );
     }
   else
     {
     double factor = 1.0 + (random.nextInt() % 20) / 100.0;
     price = new Integer( (int) (price.intValue() * factor) );
     }
   prices.put( symbol, price ); // store new price
   return price.intValue(); // return new price of stock
   }
 public int sell( int shares, String symbol )
   {
   if( shares < 0 )
     throw new IllegalArgumentException( "share count < 0" );

   return shares * quote( symbol ); // return total
   }

 public int buy( int shares, String symbol )
   {
   if( shares < 0 )
     throw new IllegalArgumentException( "share count < 0" );

   return shares * quote( symbol ); // return total
   }

 public void news( String announcement )
   {
   System.out.println( "news: " + announcement ); // display news
   }
 }
```

8.2.2.2 – A Classe Trader: modelo básico para os agentes móveis

Por que identificar a classe **Trader** como modelo funcional básico para a implementação dos agentes móveis? Não seria prudente uma abordagem que se aprofundasse em aspectos mais elaborados de maneira tal que o comportamento dos agentes sugerisse uma certa proximidade à dimensão de "software inteligente"?

Evidente que pela ênfase incorporada ao trabalho rumo à mobilidade e autonomia dos agentes, a direção cogitada na segunda pergunta não merece, por enquanto, justificativa ou reflexão algumas. Quanto à primeira questão levantada, pelo que pôde ser extraído do exemplo, a estruturação do serviço do "trader" e o relacionamento entre as classes apresentadas mostraram-se aptas e suficientes para acomodar e/ou suportar modelos interessantes para as aplicações do protótipo do Serviço de Agentes.

Em especial, fica em evidência a seguinte analogia adaptando-se, para a elaboração dos agentes móveis, a mesma lógica desenvolvida entre as classes **Trader** e **Agents1**. Ou seja, os agentes estacionários se encaixam, em geral, ao modelo da classe **Agents1** e a atribuição das tarefas idealizadas para os agentes móveis adota o perfil definido dentro do escopo da classe **Trader**. Por fim, o mesmo mecanismo de acesso do "trader" ao "mercado de ações" através das interfaces pode ser utilizado na interação dos agentes móveis com os agentes estacionários, com as tabelas e com os objetos do ambiente SICSD.

8.2.2.3 – As classes para os agentes móveis específicos: "Linha de Montagem"

O título do presente tópico pode parecer inoportuno ou fora de contexto ao se atentar parcialmente à expressão "linha de montagem". Que motivos justificariam a associação deste conceito, geralmente reservado aos domínios de um complexo industrial, às argumentações que se tecem quanto à "produção" de software? Sem devaneios, pode se eleger uma razão suficiente dentre as cabíveis. A partir do momento em que se reconhece uma estrutura – classe **Trader** - que possibilita a uniformização no desenvolvimento de um "produto" – as classes para os agentes móveis, busca-se a todo custo a moldagem e subseqüente instalação de uma "linha de montagem" que unifique

os processos de “fabricação” tanto para este “produto” quanto para futuras variações do mesmo.

Traduzindo esta linha de raciocínio, pode se reconhecer, a partir da classe **Trader**, a possibilidade de uniformização ou repetição de um padrão de desenvolvimento, por assim dizer, lógico e aplicável, para cada uma das respectivas classes de agentes móveis os quais, representando diversos tipos de serviços, se diferenciam por alguns métodos particulares à tarefa executada. Ao mesmo tempo, porém, identifica-se que invariavelmente compartilham similaridades quanto aos métodos para mobilidade e autonomia do agente. Grifa-se, portanto, como evidente a conveniência de se unificar o desenvolvimento de uma coleção de “traders”, cada qual representando um agente móvel, que se distinguem um dos outros justamente pelo tipo de serviço executado.

8.2.2.4 - As Classes para os Agentes Móveis Específicos

As classes implementadas para os agentes móveis específicos são: **TraderADD**, **TraderHist**, **TraderOS**, **TraderAddr**, **TraderMonitHist** e **TraderPerf**, além das respectivas interfaces as quais definem os métodos que podem ser acessados por outras classes.

Mencionando oportunamente o padrão de projeto **Master-Slave** (introduzido no Capítulo 4) no qual uma classe “superior” (master) delega atividades para a classe “inferior” (slave), afirma-se que os “traders” são “controlados”, sem que o conceito de autonomia seja anulado ou invalidado, por um agente estacionário (agente supervisor, agente histórico ou agente monitor). Cada classe, representando particularmente a atuação dos agentes móveis definidos, contém: a) métodos de criação, definição e identificação do agente; b) métodos de especificação do itinerário do agente; c) métodos de “disparo” controlado pelo agente estacionário; d) métodos de execução (interação e movimento); e e) métodos de desativação. O escopo de cada um destes métodos pode ser identificado no código fonte das respectivas classes.

Classe TraderADD

A classe implementa parte dos serviços de atualização e disseminação previstos na definição[11] do Agente de Atualização e Disseminação. No ambiente do protótipo atual, este agente é disparado pelo agente supervisor sempre que um novo objeto é instanciado.

Classe TraderHist

A presente classe implementa os serviços de atualização e disseminação para o Agente Monitor de Objetos (ver definição do agente AMO).

Classe TraderOS

O serviço idealizado para a classe **TraderOS** se resume à verificação periódica da situação dos objetos distribuídos pelos "hosts" do ambiente. Por meio de variáveis definidas dentro de uma faceta agregada ao objeto, este agente pode implementar diversos cenários para localização de eventuais problemas. Em primeira instância, foi considerado um agente que se move pelo ambiente do sistema e, em cada "host" específico, consulta o agente supervisor para obter a identificação e endereço de todos objetos instanciados localmente. Enquanto ocorre a transferência de tais dados, o agente móvel tenta obter respectivamente um "reply" (resposta) individual de cada objeto confirmando que está "on-line" e informações pré-determinadas que permitam avaliar se a disponibilidade "programada" para os objetos situa-se entre os níveis de "normalidade".

Torna-se imprescindível a elaboração de um mecanismo através do qual este agente móvel registre a hora e os resultados colhidos na operação, anexando-os, por exemplo, em uma faceta apropriada do agente supervisor.

[11] ver detalhes no capítulo 7

Classe TraderAddr

Esta classe implementa uma parte dos serviços previstos para os Agentes Facilitadores de Conexão (ver Capítulo 7). O código da classe **TraderAddr** aproxima-se, em similaridade, ao código da classe **TraderOS**, exceto no caso em que dentro do método **atMachine(),** os comandos devem pesquisar o número de conexões de cada objeto. No ambiente considerado atualmente, o único parâmetro avaliado é o número de conexões. Portanto, se uma aplicação cliente solicita conexão a um serviço implementado por dois ou mais objetos instanciados, o agente supervisor determina a melhor "opção" para o cliente baseando-se nas informações colhidas pelos agentes móveis da classe **TraderAddr**.

Classe TraderMonitHist

Este agente móvel específico se encarrega da monitoração dos agentes históricos. No código da referida classe, o agente verifica periodicamente o número total de registros armazenados pelos agentes históricos. Exceto no caso em que este agente acessa o agente histórico ao invés de acessar o agente supervisor, o código da classe se assemelha ao código das classes **TraderOS** e **TraderAddr**. Recordando que a atualização de operações envolvendo todos os objetos de um determinado "host" pode depender exclusivamente do processamento (disseminar e deletar os registros) dos "arquivos" do agente histórico, o agente da classe **TraderMonitHist**, ao avaliar que a quantidade de registros ultrapassa determinados parâmetros pré-configurados, pode alertar o agente monitor para que diminua o tempo dos intervalos em que é feita a verificação dos históricos.

Além disso, a presente classe pode ser customizada para análise de: a) origem dos registros, b) tipos de operações solicitadas e c) a "vazão" média, ou seja, qual o ritmo em que os registros são removidos dos históricos. Não obstante tais considerações, o código, na atual versão, não trata de todas estas situações.

Classe TraderPerf

A classe **TraderPerf** realiza o serviço idealizado e definido para o Agente Básico de Performance (ver definição do agente no Capítulo 7).

8.2.3 – A Simulação de um "TimeStamp": classes TimeStamp e Clock

Em algumas classes, como a **Line** (unidade básica dos históricos) por exemplo, foi projetado o estabelecimento de variáveis cujos dados representassem a hora "exata" da ocorrência de determinadas operações. Visto que o ambiente aborda tópicos de um sistema distribuído, verificou-se a oportunidade de se considerar uma implementação que simulasse, sem grandes pretensões, as atividades de um relógio sincronizado para os diversos "hosts". A Figura 8.10 representa o cenário configurado para esta simulação.

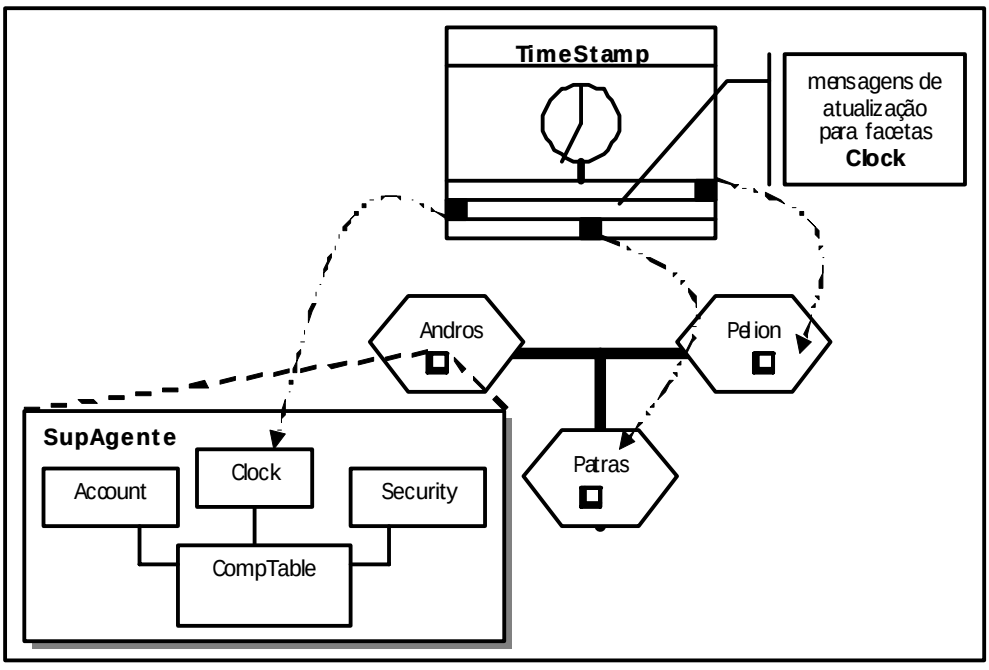

FIGURA 8.10: TimeStamp atualizando os "relógios" (faceta Clock) do agente supervisor.

As classes **TimeStamp** e **Clock** implementam tal mecanismo, simples, de simulação de um "clock" interno visando a possibilidade de sincronização na tomada de "tempos" para determinadas operações entre os agentes e o aplicativo.

Simplicidade justificada pela constatação de que a metodologia básica consistiu em um serviço específico de cronometragem que dispara, em intervalos regulares e configuráveis, "mensagens" para "objetos-relógios" localizados em cada "host". No ambiente do protótipo considerado, este serviço de cronometragem é disponibilizado pela classe **TimeStamp** que, por sua vez, acessa e atualiza variáveis pré-definidas nas facetas (agregada ao agente supervisor), representadas pela classe **Clock** (objetos-relógios). Futuramente, o serviço para "disparo" de mensagens pode se apoiar na arquitetura **Space** do Voyager através da qual se permite o "multicast"[12] de mensagens.

8. 3 – Execução do Ambiente

O capítulo de implementação é finalizado relatando-se a sequência de execução e considerações sobre o comportamento do ambiente. Este relato foi particionado em fases que podem facilitar a identificação de todos os passos da execução.

Fase 1: ***Instanciação dos Agentes Supervisores***

Os processos referentes ao agente supervisor são instanciados em portas pré-definidas em cada "host" participante do ambiente. Uma vez inicializados, cada agente supervisor tenta obter referências externas dos outros agentes antes de configurar suas próprias tabelas externas. A verificação do sucesso na inicialização de todos os agentes supervisores é evidenciada por mensagens específicas emitidas pelos mesmos.

Fase 2: ***Instanciação dos Agentes Históricos***

Os processos referentes ao agente histórico são instanciados em portas pré-definidas em cada "host" participante do ambiente.

Fase 3: ***Instanciação de objetos da arquitetura SICSD***

Um processo de simulação ativa algumas instâncias representando objetos-aplicação (serviços) do software de controle de satélites (SICSD). Agentes de

[12] ver referência à arquitetura Space [Voyager 4]

atualização disseminam a informação referente à entrada dos "novos" objetos no ambiente. Os objetos ainda não receberam nenhuma solicitação de serviço (conexão).

Fase 4: ***Instanciação do Agente Monitor de Objetos***

O agente monitor de objetos é ativado. Os parâmetros (objeto, "host") indicando respectivamente o objeto a ser monitorado e o "host" onde está instanciado, são fornecidos ao agente monitor de objetos, responsável, então, pelo "disparo" periódico dos agentes móveis (classe **TraderHist**). Assim que as operações de solicitação de serviço sejam iniciadas, os agentes móveis se encarregam da disseminação das informações.

Fase 5: ***Ativação do Relógio Interno (TimeStamp)***

O processo referente à ativação do relógio interno (Classe **TimeStamp**) é inicializado. As variáveis da faceta **Clock** dos agentes supervisores são periodicamente atualizadas.

Fase 6: ***Solicitação de Serviços***

À medida que processos de solicitação de serviços (conexão) aos objetos são executados, as informações sobre a situação de cada objeto podem ser rastreadas através das tabelas do agente supervisor. Destaca-se que, existindo duas instâncias (em "hosts" diferentes) de um objeto fornecendo o mesmo serviço, o ambiente simulou um balanceamento de carga ao distribuir as conexões consultando a situação dos objetos nas tabelas do agente supervisor.

CAPÍTULO 9
CONCLUSÃO E CONSIDERAÇÕES FINAIS

Explicitado no conjunto de palavras que compõe o título da dissertação, o foco que se assinalou em destaque como síntese geral dos objetivos do presente trabalho, dentre o número diversificado de conceitos e considerações, repousa, em suma, sobre aspectos relacionados à aplicação da tecnologia de agentes móveis ao contexto de pesquisa da arquitetura SICSD.

Pelo que se pôde constatar durante a compilação dos capítulos, a assimilação destes aspectos, moldada ao estágio atual de conclusão de trabalho, sugere uma subdivisão lógica e ilustrativa de fases compreendidas na seguinte ordem: uma fase de reconhecimento do contexto da aplicação, uma fase de fundamentação da tecnologia aplicada (ou a ser aplicada) e, por fim, uma fase de modelagem e implementação do protótipo. Portanto, as conclusões levantadas e referenciadas, sem perdas de generalidades como resultados da dissertação, são delineadas, a seguir, por meio da apresentação de variados aspectos relevantes às fases anteriormente identificadas.

Fase de Reconhecimento do Contexto de Aplicação

Um aspecto importante se destaca frente à constatação de que a aplicação proposta não divagou por soluções genéricas para diversos problemas. A motivação básica se relacionou à disponibilização do Serviço de Agentes dentro da arquitetura SICSD pré-definida. Ou seja, a partir do ponto em que se definiram especificamente o limite e o tipo de aplicação, facilitou-se a tarefa de se compreender o “porquê utilizar agentes”. Sem promover a supervalorização ou superestimação da tecnologia de agentes, o trabalho defendeu e acatou a necessidade restritiva de um estudo minucioso quanto à adequabilidade, à aplicabilidade, às vantagens e desvantagens da solução baseada em agentes a partir do panorama definido.

Fase de Fundamentação da Tecnologia Aplicada

Os principais aspectos levantados na elaboração do Capítulo 3 induzem às seguintes constatações: i) a definição do conceito de agente foi moldada através da identificação de atributos respectivamente associados às dimensões da agência, da inteligência e da mobilidade; ii) a documentação pesquisada sobre as tendências atuais para utilização de agentes e sobre os tipos de aplicações em desenvolvimento colaborou na constatação de que os objetivos traçados para o trabalho estavam sintonizados, em geral, aos grandes centros de pesquisa; iii) a análise das armadilhas no desenvolvimento orientado a agentes, além de se firmar como bússola preventiva contra metodologias errôneas, sinaliza para a necessidade latente de submissão e concordância aos aspectos da engenharia de software (Woo/Jen 1999).

A concretização da tecnologia de agentes como um paradigma da engenharia de software ainda exige a satisfação de algumas restrições. A principal delas, talvez, se refira à necessidade de desenvolvimento de técnicas de engenharia de software, especificamente ajustadas à tecnologia de agentes, através das quais os desenvolvedores possam naturalmente compreender, modelar e implementar soluções para determinadas classes de sistemas distribuídos. Ou seja, pelas impressões colhidas no decorrer dos trabalhos de elaboração do Serviço de Agentes, o suporte disponibilizado pelas plataformas de agentes não apresentou avanços em direção à consideração de mecanismos específicos para análise e projeto de soluções baseadas em agentes. Em outras palavras, faz-se referência à presença irrestrita da "*engenharia de software orientada a agentes*".

A tecnologia de agentes móveis, ocupando lugar "privativo" no Capítulo 4, participa crucialmente como bagagem de suporte e ferramenta de apoio para a idealização do Serviço de Agentes como um ambiente distribuído de agentes. Dentre as diversas "vantagens previstas" na utilização da tecnologia de agentes móveis, deve se ressaltar os primeiros retornos: i) a execução assíncrona e autônoma (por exemplo, os agentes de atualização, uma vez disparados, desfrutam de autonomia no cumprimento de suas tarefas sem exigir que os agentes supervisores "bloqueiem" suas atividades enquanto aguardam por uma mensagem de retorno); ii) mobilidade / redução no tráfego de rede (por exemplo, os agentes móveis "carregam" a funcionalidade para

disseminação de informações enquanto migram através dos "hosts" do ambiente e as interações locais podem "desafogar" a utilização constante da rede); iii) paradigma de desenvolvimento conveniente (a criação de um sistema distribuído e flexível é simplificada pela implantação e suporte de plataformas para agentes móveis).

A introdução do Capítulo 5 anunciou a diversidade de sistemas de agentes móveis (considerando-se apenas os "candidatos" implementados em Java) como fato incontestável. O aspecto evidenciado frente ao "leque de opções" produzido por diversas empresas e institutos de pesquisa levou a deduções como: i) o desenvolvedor de aplicativos baseados em agentes não busca "reinventar a roda" visto que a infra-estrutura de suporte é disponibilizada por diferentes fornecedores da tecnologia. ii) por outro lado, o desenvolvedor, por precaução e prudência, deve definir requisitos que permitam a escolha da plataforma mais apropriada aos seus objetivos.

À primeira vista, como resultado imediato da comparação entre plataformas apresentada, as plataformas Grasshopper e Voyager se equipararam em relação às características avaliadas como vantagens. Em uma perspectiva contextualizada ao domínio do presente trabalho, porém, as contribuições fornecidas por ambas as plataformas se posicionaram em proporções e áreas de aplicação diferentes. Apresentando módulos compatíveis à especificação MASIF e uma detalhada e extensa metodologia conceitual, Grasshopper ofereceu parâmetros suficientes e satisfatórios para a concepção teórica do Serviço de Agentes (Capítulo 7), mas a carência de exemplos para desenvolvimento de aplicações e a dificuldade no manuseio do código e dos pacotes fornecidos frustraram as tentativas de implementação prática. Portanto, justifica-se a priori, o capítulo de destaque para a plataforma Voyager (Capítulo 6), visto que a mesma, resultante de uma integração evolutiva a partir da tecnologia de computação distribuída e complementada pela tecnologia de agentes, evidenciou facilidade, flexibilidade e praticidade justamente nos pontos de "deficiência" (leia-se complexidade de uso) da outra plataforma.

Fase de Modelagem e Implementação do Protótipo

Na extensão dos Capítulos 7 e 8, o Serviço de Agentes, sem permitir a associação ou comparações a um sistema comercial ou produto exaustivamente testado

e aprovado cientificamente, foi modelado e implementado através da integração de três conceitos principais: os **agentes** (móveis e estacionários), atuando em **plataformas** de execução, implementam a funcionalidade do **serviço-aplicação** (monitoração de objetos). Os tópicos abordados na concepção dos comentários, a seguir, se orientam através destes três termos sublinhados.

O primeiro e fundamental aspecto se refere à intenção explícita de se priorizar a compatibilidade com padrões em evidência e consagrados além de se beneficiar das tecnologias correlatas em desenvolvimento. Assim, as diretrizes que definiram os modelos para os ambientes de execução - plataformas de agentes (incluindo os serviços fornecidos "built-in") - e para os próprios agentes obtiveram respaldo nas especificações MASIF e FIPA e nos produtos Voyager e Grasshopper. O padrão CORBA, firmando-se como infra-estrutura de comunicação dentro do contexto de ambientes distribuídos, dispensa maiores comentários ou referências além dos apontados no trabalho. A difusão da linguagem Java pôde ser diagnosticada frente à presença majoritária da mesma nas plataformas de agentes móveis. Os atributos diferenciais do Java que o impulsionam rumo à posição de "linguagem líder" para aplicações distribuídas baseadas em agentes podem ser sintetizados em: a portabilidade de uma linguagem 100% orientada a objetos. O retorno imediato desta abordagem vinculada às padronizações e tecnologias de suporte se evidenciou na aceleração do processo de fundamentação e desenvolvimento. Ou seja, sem enfrentar a sobrecarga de se conceber toda a infra-estrutura necessária "a partir do zero", prevaleceu a canalização de esforços para o tratamento de tópicos particulares ao aplicativo de monitoração (aplicação objetivo).

Ainda sob o foco posicionado no parágrafo anterior, ressaltam-se duas constatações inerentes à opção por um "produto acabado" (Voyager) para a instalação da plataforma de agentes. 1ª) O desenvolvimento do Serviço de Agentes não foi sobrecarregado pela necessidade de codificação de detalhes de baixo nível implícito nas aplicações distribuídas baseadas em agentes. Os mecanismos de suporte à mobilidade dos agentes e dos objetos da aplicação, à interação e à compatibilidade com objetos e interfaces CORBA, à criação e ao gerenciamento de objetos remotos, à carga dinâmica de classes e à localização transparente de objetos, entre outros "serviços" referenciados no Capítulo 6, ficaram sob a responsabilidade da plataforma. 2ª) Ao lado destes "benefícios" proporcionados, entretanto, não se desconsidera o "custo" razoável

implícito no tempo reservado e requerido para instalação, configuração e assimilação dos requisitos básicos para o funcionamento adequado da plataforma. A documentação disponível e um suporte eficaz (através do "site" da empresa na Internet) colaboraram para a efetivação, em tempo hábil, deste estágio.

Um outro aspecto, não menos importante, se relaciona à flexibilidade para se diversificar e/ou realçar, futuramente, a funcionalidade e os tipos de serviço-aplicação a partir do protótipo apresentado. Partindo da constatação de que as abstrações para o serviço de monitoração de objetos se restringem e estão intimamente ligadas, em princípio, aos tipos de agentes designados para esta determinada aplicação, a providência mínima para se "incrementar" o Serviço de Agentes exige, em tese, a especificação de outras classes de agentes "específicos" que se incumbiriam pela efetivação de novos serviços A flexibilidade sugerida neste parágrafo, entretanto, não deve induzir a conclusões de que o Serviço de Agentes se declara como solução genérica para uma infinidade de problemas e cenários.

A flexibilidade estampa outra faceta, desta vez, relacionada aos tipos de agentes abordados para a concepção do serviço-aplicação. Ainda que evidenciado o atributo da mobilidade, o Serviço de Agentes foi, com equilíbrio, composto por agentes móveis e agentes estacionários. Ratifica-se a adoção desta "metodologia maleável" no conceito de Mobilidade Estratégica (Chia 1996), fruto de pesquisas que atestaram a utilização balanceada e inteligente da estrutura de rede através de uma modelagem subdividida em agentes-tarefa móveis e estacionários. Fica reservado ao desenvolvedor, portanto, o estabelecimento de diretrizes que facilitem ou conduzam a uma decomposição tão otimizada quanto possível.

Em relação ao serviço-aplicação proposto, as impressões conclusivas que procedem do Serviço de Agentes aceitam especificamente a abrangência do contexto da arquitetura SICSD e permanecem válidas enquanto proposições através das quais se permite verificar a aplicação, as propriedades, o impacto e os resultados do protótipo configurado no ambiente de teste. Verifica-se, portanto: 1) A aplicação: o emprego da tecnologia de agentes (móveis e estacionário), não se resumindo ao aspecto teórico, comprovou-se adequado na execução prática para a monitoração de objetos; 2) As propriedades do aplicativo: totalmente orientado a objetos, implantado sobre uma

camada de comunicação robusta (CORBA), ambiente idealizado com a tecnologia de agentes, interage com objetos participando de uma arquitetura distribuída e dinâmica; 3) O impacto: utilização de conceito de abstração inédito, alternativo e promissor que, por carência de metodologias específicas para análise e/ou projeto de software orientado a agentes, ainda se apóia nas técnicas de programação orientada a objetos; e 4) Os resultados: sem apoio em medidas de performance e desempenho ou em suposições, precipitadas ou otimistas em demasia, a partir de um protótipo, o aplicativo representa um avanço, sem precedentes no departamento, rumo à incorporação da tecnologia de agentes, cujas "vantagens previstas", já destacadas em parágrafo anterior e comprovadas na prática, servem como prognóstico favorável.

Durante a fase de implementação para o protótipo do Serviço de Agentes, não foi possível desprezar a existência de um aspecto, polêmico talvez, diretamente relacionado à metodologia utilizada para a codificação dos agentes: **Qual o relacionamento entre as tecnologias de objetos e de agentes?** Tal questão representa um certo impasse a partir do momento em que se identificou a utilização de técnicas e "ferramentas" de desenvolvimento orientada a objetos para a "confecção" dos agentes. Sem a responsabilidade de se estabelecer um "ponto final" de consenso quanto às similaridades, às diferenças e aos mecanismos para a interoperabilidade destas duas áreas, destaca-se a conveniência de se concordar com três pontos de vista relevantes.

1º) Agentes são Diferentes de Objetos: os agentes apresentam habilidades peculiares tais como autonomia na realização de tarefas, utilização de uma linguagem específica para comunicação com outros agentes, capacidade para representação de conhecimento, realização de inferências e alteração do comportamento baseado em aprendizado. Este ponto de vista admite, porém, o fato de que a tecnologia de objetos pode ser e foi utilizada na implementação da tecnologia de agentes e de que os agentes podem "trabalhar" com objetos "a bordo" do estilo orientado a objetos.

2º) Agentes são Objetos: Assume-se, também, que os agentes são objetos ou componentes a partir da perspectiva plausível de que possuem identidade (pode-se diferenciar um agente do outro), estado e comportamento próprios (distinto de outros agentes) e interfaces por e através das quais se comunicam com outros agentes e com outras "entidades" (objetos, por exemplo). Faz-se oportuna, como complemento, a

referência de (Bradshaw 1997) na qual os agentes são conceituados como "***objetos com uma atitude***". O termo "objetos", neste contexto, se associa ao conceito de um mecanismo genérico de abstração ou modelagem, sem qualquer dependência ao fato de que os agentes foram implementados como objetos (através de técnicas de programação orientadas a objetos).

3º) Alvo: Interoperação de Tecnologias (Agentes & Objetos): Existe a real necessidade de que ambas as tecnologias coexistam, e ainda mais, se "integrem" de maneira tal que os agentes possam interagir com objetos e vice-versa. As manifestações nesta direção já esboçam níveis promissores de avanço. O grupo da OMG - Agent Working Group, em particular, privilegia suas atenções para o relacionamento entre agentes e objetos (destaque no objetivo de transposição da "distância" entre as duas áreas, evidenciado pela concentração conjunta de foco para os agentes dentro e a partir de um consórcio de empresas, antes de tudo, vinculado à tecnologia de objetos). Com propósito similar, a empresa ObjectSpace imprimiu na plataforma de agentes Voyager uma filosofia para desenvolvimento na qual se assume que os agentes são, de fato, tipos "especiais" de objetos que se movem com autonomia na efetivação de suas tarefas e, exceto tais peculiaridades, se comportam como um outro objeto qualquer.

Considerações Finais

Os aspectos de mobilidade e autonomia enfatizados no protótipo para o Serviço de Agentes não representam a exploração completa das propriedades de agentes propícias ao contexto da arquitetura SICSD. Comenta-se, portanto, a possibilidade de futuras abordagens que considerariam outras "facetas" destacadas como interação, adaptação, coordenação, inteligência, "wrapping" e sistemas multi-agentes (OMG 2000).

O próprio contexto de aplicação dentro do qual foi inserido o Serviço de Agentes evidenciou que a tecnologia de agentes não emerge isoladamente, tratando-se, na realidade, de uma aplicação integrada de múltiplas tecnologias. Do mesmo modo, os agentes não representam uma "forma" inédita e independente de aplicação. Pelo contrário, assumem o perfil de uma metodologia, com características apropriadas à ambientes distribuídos, através da qual se buscou adicionar novas habilidades, como

autonomia e mobilidade, à monitoração de objetos na arquitetura SICSD. Não se deve desprezar, portanto, a complexidade adicional associada aos mecanismos (padrão CORBA) para interoperação dos objetos distribuídos, à distribuição dinâmica dos objetos e agentes (mobilidade), à infra-estrutura de rede e à interação dos agentes e objetos, entre outros.

Outro aspecto inerente às dificuldades para a composição do Serviço de Agentes se refere ao "status" da tecnologia de agentes, a qual, posicionada atualmente no campo de etapas e atividades de pesquisa, acumula alguns "obstáculos" previsíveis e contrários à sua maturação. Em suma, a infra-estrutura necessária à completeza da tecnologia não está suficientemente testada e disponível, o conjunto de técnicas propostas para o desenvolvimento baseado em agentes não está adequadamente integrado, os produtos que surgem são pioneiros em áreas específicas e isoladas, e os projetos dos "defensores ferrenhos" que já empregam a "nova religião" ainda se encontram em estágios a partir dos quais se permite apenas a demonstração precipitada e imprecisa do valor da tecnologia de agentes.

Por fim, a essência da contribuição do trabalho não está sintetizada tão somente na comprovação da praticabilidade da tecnologia de agentes móveis dentro do contexto do Serviço de Agentes. Ao lado, porém, deste passo significante, a pesquisa impulsiona as engrenagens da inovação ao ampliar os horizontes para o avanço da proposta da arquitetura SICSD. Ou seja, sintonizado à tendência de "reestruturação" da arquitetura do software para controle de satélites, o Serviço de Agentes posiciona como retorno proeminente para o departamento – CCS – a integração de um ambiente distribuído de agentes com o ambiente flexível e dinâmico baseado em objetos distribuídos (SICSD), consumada, a priori, pela concepção do protótipo para o serviço de monitoração de objetos.

> **"É difícil prever ao extremo seus resultados práticos. ... O dilema dos que têm de aprovar verbas para pesquisa reside exatamente no fato de que o teste de praticabilidade não é um critério apropriado. ... Se uma**

proposta tem óbvia aplicação, ela é, quase certamente, assunto para desenvolvimento, não para pesquisa." ***Dr. Aldo Vieira da Rosa***[1]

[1] 81, doutor em Engenharia, professor emérito da Universidade Stanford e fundador do Instituto Nacional de Pesquisas Espaciais em 1963, em artigo "A estação espacial brasileira"- Folha de São Paulo – 04/05/1999

REFERÊNCIAS BIBLIOGRÁFICAS

Aaron, B.; Aaron, A. **ActiveX:** technical reference. Prima Publishing, U.S.: Pap/CD edition, 1997.

ACTS – **Agent Cluster Website.** Disponível em: < http://www.fokus.gmd.de/ima/miami/public/service/agent_cluster >. Acesso: 01 jul. 1998.

Agha, G**, Actors:** a model of concurrent computation in distributed systems. Cambridge, MA: The MIT Press, 1986.

AGLETS - **Homepage of Aglets.** Disponível em: < http://www.trl.ibm.co.jp/aglets/ >. Acesso: 01 jul. 1998.

APRIL - **Homepage of April.** Disponível em: <http://www.fujitsu.co.jp/ hypertext/ Products/ Software/April/Eindex.html.>. Acesso: 01 jul. 1998.

Baumer, C., Breugst, M.; Choy, S.; Magedanz, T. **Grasshopper:** A universal agent platform based on OMG MASIF and FIPA Standards. Berlin, Germany: IKV++ GmbH, 1999.

Benech, D.; Desprats, T.; Raynaud, Y. A KQML-CORBA based architecture for intelligent agents communication in cooperative service and network management. In: IFIP/IEEE International Conference on Management of Multimedia Networks and Services, 1197, [S.l]. **Proceedings…**[S.l], 1997. p.8-10.

Bieszczad, A. , Pagurek, B., White, T. Mobile agents for network management. **IEEE Communications Surveys**, v.2, n.5, Oct 1998. Disponível em: <http://www.comsoc.org/pubs/surveys>. Acesso: 01 ago. 1998.

Bigus, Joseph P.; Bigus, J. **Constructing intelligent agents with Java*:*** a programmers's guide to smarter applications. USA: Wiley Computer Publishing, 1998 p. 193.

Bratman, M. E. **Intention, plans, and practical reason**. Cambridge, MA: Harvard University Press:, 1987.

Brooks, F. P. No silver bullet. In: World Computer Conference, 20., of the IFIP, 10., 1986. **Proceedings…** Amsterdam: Elsevier Science Publishers, 1986. p.1069-1076.

Caglayan, A.; Harrison, C. **Agent sourcebook :** a complete guide to desktop, internet, and intranet agents. New York: John Wiley & Sons, 1987.

Chess, D.; Grosof, B.; Harrison, C.; Levine, D.; Parris, C.; Tsudik, G. Itinerant agents for mobile computing. **IEEE Personal Communications**, v.2, n.5, p.34-49, Oct. 1995.

CRU. **Trusted computer system evaluation criteria.** Disponível em: <URL:http://www.cru.fr/securite/Documents/generaux/orange.book>. Acesso: 04 jul. 1998.

DAGENTS. **Homepage of D`Agents.** Disponível em: <http://www.cs.dartmouth.edu/~agent/>. Acesso: 05 jul. 1998.

Etzioni, O. Moving up the information food chain: deploying softbots on the World Wide Web. In: AAAI-96 National Conference on Artificial Intelligence, 13., 1996, Portland, OR**. Proceedings…** Portland: AAAI, 1996**.** p.1322-1326.

Farmer, W.; Guttman, J.; Swarup, V. Security for mobile agents: authentication and state appraisal. In: ESORICS '96 European Symposium on Research in Computer Security, 4., 1996, . **Proceedings…** Roma: [s.n], 1996. p.118-130.

FERREIRA, M. G. V. **Uma arquitetura flexível e dinâmica para objetos distribuídos aplicada ao software de controle de satélites**. 2001. 244 p. (INPE-8602-TDI/787). Tese (Doutorado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos. 2001. Disponível em: <http://urlib.net/dpi.inpe.br/lise/2003/01.16.09.56>. Acesso em: 08 jun. 2009.

Finin, T.; McKay, D.; Fritzson, R.; McEntire, R. **Knowledge building and knowledge sharing.** [S.l]: Ohmsha and IOS Press, 1994.

Foundation for Intelligent Physical Agents (FIPA). **Agent Management.** FIPA 1997 specification, part 1, version 2.0, October 1998. Disponível em: < http://www.fipa.ord/spec/fipa97/fipa97.html>. Acesso: 11 ago. 1998.

Franklin, S.; Graesser, A. Is it an agent, or just a Program?: A Taxonomy for Autonomous Agents. In: International Workshop no Agent Theories, Architectures, and Languages, 3., 1996, **Proceedings…** [S.l]: Springer-Verlag, 1996. p.21-35.

Fuggetta, A.; Picco, G.P.; Vigna, G. **Understanding code mobility**. Disponível em: < http://www.cs.ucsb.edu/~vigna/listpub.html>. Acesso: 14 set. 1999.

Garone, S. ObjectSpace: An agent for change in distributed object computing. **IDC Bulletin** #17289. Sept., 1998.

Georgeff, M. P.; Lansky, A.L. Reactive Reasoning and Planning. In: AAAI-87 National Conference on Artificial Intelligence,6., 1987, Seattle: WA, **Proceedings…** Seattle: AAAI, 1987. p.677-682.

Ghezzi, C.; Vigna, G. Mobile code paradigms and technologies: a case study. In: International Workshop on Mobile Agents, MA'97, 1., 1997, [s.l.]. **Proceedings…** [S.l.]: Springer, 1997.p.39-49.

Gilbert, D. **Intelligent agents:** the right information at the right time. Durham, NC: IBM Corporation, Research Triangle Park, May 1997. IBM Report.

GMI. **Mobile agents white paper**. Disponível em: <http://www.generalmagic.com >. Acesso: 22 mar. 1999.

GRASSHO. **Homepage of grasshopper.** Disponível em: <http://www.ikv.de/products/ grasshopper.html>. Acesso: 13 abr. 1998.

Grasshopper. **Grasshopper development system –** programmer's guide. Release 1.2 – (Revision 1.3), IKV++, Febr., 1999a

Grasshopper **Grasshopper development system** – basics and concepts**.** Release 1.2 – (Revision 1.1), IKV++, Febr., 1999b.

Grasshoper. **Grasshopper –** a platform for mobile software agents. IKV++, 1998a.

Green, S. **Software agents: a** review. [S.l.]: Trinity College Dublin, Broadcom Éireann Research Ltd., 1997. Technikai Report, TCS-CS-1997-06

Hermans, B. Peer-reviewed journal on the Internet. **Intelligent Software Agents on the Internet.** Disponível em: <http://www.firstmonday.org/issues/issue2_3/ch_123/>. Acesso: 05 mai. 1999.

Hohl, F. Time limited blackbox security: protecting mobile agents from malicious hosts**.** In: Vinga G. (ed.). **Mobile agents and security.** [S.l.]: Springer-Verlag, 1998. p.92-113. Lecture Notes in Computer Science No. 1419.

IEEE Network Magazine, May/June, v.12, n.3, May/June 1998. Special Issue on Active and Programmable Networks.

Mitsubishi Electric ITA **Mobile agent computing:** a white paper. Waltham, MA: Horizon Systems Laboratory, January, 1998.

Jansen, W.; Karygiannis, T. **Mobile agent security.** Gaithersburg, National Institute of Standards and Technology - Computer Security Division, 1999, MD 20899. NIST Special Publication 800-19.

JAT. **Java agent template.** Disponível em: <http://Java.stanford.edu/Java_agent/html>. Acesso: 17 mai. 1998.

Jennings, N. R.; Corera, J.M.; Laresgolti, L. Developing industrial multi-agent systems. In: International Conference on Multi-Agent Systems (ICMAS-95), 1., San Francisco. **Proceedings...** San Francisco: [s.n], 1995. p.423-430.

JINI. **Site oficial do Jini**. Disponível em: <http://java.sun.com/products/jini>. Acesso: 14 ago. 1998.

Karjoth, G.; Asokan, N.; Gülcü, C. Protecting the computation results of free-roaming agents. In: International Workshop on Mobile Agents, Stuttgart, 2., 1998. Stuttgart,Germany. **Proceedings…** Stuttgart: [S.n.], Sept. 1998.

Kiniry, J.; Zimmerman, D. A hands-on look at java mobile agents. **IEEE Internet Computing**, v.2, n.4, p.21-30, July / Aug.1997.

KNABE. **Language support for mobile agents.** Disponível em: <ftp://reports-archive.adm.cs.cmu.edu/1995/CMU-CS-95-223.ps.Z>. Acesso: 22 jan. 1999.

KQML. **Specification of the KQML agent-communication language.** Disponível em: <http://www.cs.umbc.edu/kqml/kqmlspec/spec.html>. Acesso: 4 nov. 1999.

Lange, D.B.; Oshima, M. **Programming and deploying Java™ Mobile Agents with aglets.** [S.l.]: Addison-Wesley, 1998.

_____. **Mobile objects and mobile agents:** the future of distributed computing. California, USA: General Magic, Inc.,1998.

Lange, D.B.; Aridor, Y. Agent design patterns: elements of agent application design. In: ACM international conference on Autonomous Agents (Agents '98), 2., [S.l.]. **Proceedings...** ACM Press, 1998.

Lieberman, H. Letizia: an Agent that assist Web Browsing. In **Proceedings of the Fouteenth International Joint Conference on Artificial Intelligence** (UCAI-95). Montreal, Canada, 1995. p.924-929.

Magedanz, T.; Hagen, L.; Breugst, M. Impacts of Mobile Agent Technology on Mobile Communications System Evolution. **IEEE Personal Communications**, v.5.n.4, p.56-69, 1998.

Magedanz, T.; Busse, I.; Covaci, S.; Breugst, M. Grasshopper - A Mobile Agent Platform for IN Based Service Environments. In: IEEE IN Workshop, 1998, Bordeaux, France. **Proceedings…** Bordeaux: IEEE, 1998. p.279-290.

Manola, F. **Agent standards overview**. [S.l.]: Inc. July 1998. OBJS Technical Note Object Services and Consulting.

Matthiske, B.; Matthes, F.; Schmidt, J. On migrating threads. **Journal of Intelligent Information Systems**, v..8, n.2, p.167-191, Mar. 1997.

MIAMI. **The MIAMI Project home page**. Disponível em: <www.fokus.gmd.de/research/cc/ima/miami>. Acesso: 04 jun. 1999.

NECULA. **Safe kernel extensions without run-time checking**. Disponível em: < http://www.cs.cmu.edu/~necula/papers.html>. Acesso: 17 ago.1999.

Nelson, J. **Programming mobile objects with java**. [S.l.]: John Wiley & Sons, 1999.

NEVAREZ, A. **Novell, Java & Voyager –** case study. Java Technology Group Enterprise Java – March 1999. Disponível em: <http://www.javaworld.com/channel_content/jw-enterprise-index.html> Acesso em 01 Nov. 2008.

ODYSSEY. **Homepage of odyssey.** Disponível em: <http://www.genmagic.com/technology/odyssey.html>. Acesso: 01 fev. 1999.

OMG. **Mobile agent system interoperability facilities specification.** Disponível em: < http://www.omg.org/techprocess/meetings/schedule/tbl_MOF_Specification>. Acesso: 25 mar. 1998.

Ordille, J.J. When Agents Roam, Who Can You Trust?. In: Conference on Emerging Technologies and Applications in Communications, 1., 1996, Portland, Oregon. **Proceedings…**Portland: [S.l.], 1996p.188–191.

Outerhout, J. K. Scripting: Higher-Level Programming for the 21st Century**. IEEE Computer**, p.23-30, 1998.

Papaioannou, T.; Edwards, J. Mobile Agent Technology in Support of Sales Order Processing in the Virtual Enterprise. In: MSI Research Institute. IEEE/IFIP international conference on Intelligent systems for manufacturing, 3., 1998, Prague, Czech Republic, **Proceedings…** Prague: IEEE, 1998. p.23-32.

RETICULAR. **Agent construction tools**. Disponível em: <www.agentbuilder.com/AgentTools/index.html>. Acesso: 16 abr. 1999.

Riordan, J.; Schneider, B. **Environmental key generation towards clueless agents.** [S.l.]: Springer-Verlag, 1998. Lecture Notes in Computer Science n.1419,

Roth, V. Secure recording of itineraries through cooperating agents. in: secure internet mobile computations. In: COOP Workshop on Distributed Object Security and Workshop on Mobile Object Systems, 4., France. **Proceedings…** France, INRIA, 1998. p.147-154.

Rothermel, K.; Hohl, F.; Radouniklis, N. **Mobile agent systems:** what is missing? London, UK: Chapman & Hall, 1998. p.111-124.

Russel, S. J.; Norvig, P. **Artificial intelligence:** a modern approach. Englewood Cliffs, NJ: Prentice Hall, 1995.

Rymer, J. The Muddle in the Middle. **Byte Magazine**, v.21, n.4, p.67-70, Apr. 1996.

COSTA, S. R. **Objetos Distribuídos**: Conceitos e Padrões. 2000. 230 p. (INPE-7939-TDI/744). Dissertação (Mestrado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, Sao Jose dos Campos. 2000. Disponível em: <http://urlib.net/sid.inpe.br/deise/2001/04.24.14.21>. Acesso em: 08 jun. 2009.

Sander, T.; Tschudin, C. **Protecting mobile agents against malicious hosts**. Lecture Notes in Computer Science n.1419, 1998.

Schneider, F.B. Towards fault-tolerant and secure agentry. In: International Workshop on Distributed Algorithms, 11., Saarbucken, Germany, 1997. **Proceedings…** Saarbucken, Germany: [s.n.], 1997. v.1320, p.1-14. Lecture Notes In Computer Science.

Steels, L. Cooperation between distributed agents through self organization. In: European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89), 1., Amsterdam. **Proceedings…** Amsterdam: Elsevier, 1990. p.175-196. Decentralized AI.

SUN. **The java language.** Technical Report. Disponível em: < http://java.sun.com/docs/white/langenv/>. Acesso: 21 out. 1999.

TODD. **An introduction to agent**. JavaWorld. Disponível em: < http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html>. Acesso: 23 abr. 1998.

Vigna, G. Protecting mobile agents through tracing. In: ECOOP Workshop, 3., 1997, Jyvälskylä, Finland, **Proceedings…** Jyvälskylä, Finland, [s.n], 1997. Mobile Object Systems.

VISIGENIC. **Distributed object computing in the internet age**. Disponível em: <http://dn.codegear.com/article/26026>. Acesso: 19 set. 1999.

VOYAGER. **Homepage of Voyager**. Disponível em: < http://www.objectspace.com/voyager/ index.html>. Acesso: 5 abr.1998.

VOYAGER01. **Voyager and agent platforms comparison**. Disponível em: <http://www.objectspace.com/voyager/ index.html>. Acesso: 5 abr. 1998.

VOYAGER02. **ObjectSpace Voyager core package technical overview**. Disponível em: <http://www.objectspace.com/voyager/ index.html>. Acesso em: 5 abr. 1998.

VOYAGER03. **Voyager CORBA integration technical overview**. Disponível em: <http://www.objectspace.com/voyager/ index.html>. Acesso: 5 abr.1998.

VOYAGER04. **Voyager core technology 2.0 –** user guide. Disponível em: <http://www.objectspace.com/voyager/ index.html>. Acesso: 5 abr.1998.

VOYAGER05. **The world's first platform for agent-enhanced distributed computing in Java**. Disponível em: <http://www.objectspace.com/voyager/ index.html>. Acesso: 6 abr. 1998.

WAHBE. **Efficient software-based fault isolation**. Disponível em: <http://www.cs.duke.edu/~chase/vmsem/readings.html>. Acesso: 2 set.1998.

Webster, B. F. **Pitfalls of objetct oriented development**. New York: M&T Books, 1995.

Wooldridge, M.; Jennings, N.R. Inteligent agents**:** theory and practice. The **Knowledge Engineering Review**, v.10, n.2, p. 115--152, 1995.

YEE. **A Sanctuary for mobile agents**. Disponível em:
< http://www-cse.ucsd.edu/users/bsy/index.html> Acesso: 01 nov. 1998.

Young, A.; Yung, M. In: Sliding encryption: a cryptographic tool for mobile. In: International Workshop on Fast Software Encryption, 4., 1997, Haifa, Israel. **Proceedings…** Haifa, Israel, FSE , 1997.

APÊNDICE A

CÓDIGO FONTE

1 – AGENTE SUPERVISOR

1.1 – Classe SupAgente.class

```
package douto.tms_age;
import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;
import java.util.*;

public class SupAgente
 {
 public static void main( String[] args )
  {
  try
   {
    // Inicialização do servidor voyager na porta 8000
    Voyager.startup( "8000" ); String tabClass = CompTable.class.getName();
   // Criação das tabelas para referências interna e externa
   Object[] args2 = new Object[]{"SupAgentPatras", "WWW", "WWW", "WWW", new Integer(0),
   new Integer(0), new Integer(0)};
   ICompTable tabela1= (ICompTable)Factory.create(tabClass,
   args2,"//150.163.20.7:8000/SuperAgentPatras");
   Object[] args3 = new Object[]{"SuperAgentPatrasExt", "WWW", "WWW", "WWW", new
   Integer(0), new Integer(0), new Integer(0)};
   ICompTable tabela2 = (ICompTable)Factory.create(tabClass,
   args3,"//150.163.20.4:8000/SuperAgentPatrasExt");
   Object[] args4 = new Object[]{"SuperAgentPatrasExt", "YYY", "YYY", "YYY", new Integer(0), new
   Integer(0), new Integer(0)};
    ICompTable tabela3 = (ICompTable)Factory.create(tabClass,
   args4,"//150.163.20.19:8000/SuperAgentPatrasExt");
    //Agente Supervisor instanciado
   System.out.println("Agente Patras e agentes externos inicializados: operando...");
   }
  catch( Exception exception )
   {
   System.err.println( exception );
   }
  }
 }
```

1.2 – Interface IComp.class

```
package douto.tms_age;

public interface IComp
{
   String getName();    String getIp_addr();String getInc_time();
   void   setInc_time(String novo_time);  void   setcon_patras(int con);
   void   setPort_num(String novo_port_num);    void   setIp_addr(String novo_ip);
   int    getcon_patras(); String getPort_num();
   int    getcon_pelion(); int    getcon_andros();
   void   setcon_andros(int con); void   setcon_pelion(int con);
 }
```

1.3 – Classe Comp.class

```
package douto.tms_age;
import java.io.*;
```

```
public class Comp implements IComp, Serializable
 {
  String name;
  String ip_addr;
  String port_num;
  String incoming_time;
  int    con_patras;
  int   con_pelion;
  int  con_andros;

  public Comp( String name, String ip_addr, String port_num, String incoming_time,
                  int con_patras, int con_pelion, int con_andros)
   {
    this.name          = name;
    this.ip_addr       = ip_addr;
    this.port_num      = port_num;
    this.incoming_time = incoming_time;
    this.con_patras   = con_patras;
    this.con_pelion   = con_pelion;
    this.con_andros   = con_andros;
    }

 public String toString()
  {    return "Componente( " + name + " )";   }

 public String getName()
  {    return name;   }

 public String getIp_addr()
  {    return ip_addr;   }

 public void setIp_addr(String novo_ip)
  {    this.ip_addr = novo_ip;   }

 public String getPort_num()
 {    return port_num;  }

 public void setPort_num(String novo_port_num)
 {    this.port_num = novo_port_num;  }

 public String getInc_time()
   {    return incoming_time;    }

public void setInc_time(String novo_time)
   {    this.incoming_time = novo_time;    }

 public int getcon_patras()
 {    return con_patras;  }

 public void setcon_patras(int con)
 {    this.con_patras = con;  }

public int getcon_pelion()
 {    return con_pelion;  }

 public void setcon_pelion(int con)
 {    this.con_pelion = con;  }

 public int getcon_andros()
 {    return con_andros;  }

 public void setcon_andros(int con)
  {    this.con_andros = con;  }
 }
```

1.4 – Interface ICompTable.class

```
package douto.tms_age;
import java.util.Vector;

public interface ICompTable
{
   void addComp( IComp comp );
   void getComp( String name );
   int getRedundancia_Comp(String name );
   void updateNum_conexao_Patras(String name, int novo_num_conexao );
   void updateNum_conexao_Pelion(String name, int novo_num_conexao );
   void updateNum_conexao_Andros(String name, int novo_num_conexao );
   String getAddress_Comp(String name );
   String getConexao_Comp(String name, String origem, IHistory historico);
   int getComp_by_parametro(String name,String parametro);
   String removeComp(String name );
   Vector getVector();
}
```

1.5 – Classe CompTable.class

```
package douto.tms_age;
import java.util.Vector;
import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;
import java.util.*;
import java.lang.*;

public class CompTable extends Comp implements ICompTable
 {
   public String Nome_para_tabela;
   Vector tabela = new Vector();

   public CompTable( String name, String ip_addr, String port_num, String incoming_time,
                        int con_patras, int con_pelion, int con_andros)
   {
     super( name, ip_addr, port_num, incoming_time, con_patras, con_pelion, con_andros );
     IFacets facets = Facets.of( this );
     IAccount account = (IAccount) facets.of( "douto.tms_age.IAccount" );
     account.deposit( 10000 );
     ISecurity security = Security.of(this);
     security.setStatus(true);
     security.setAmount(0);
     IClock clock = Clock.of(this);
     clock.setSecond(0);
     System.out.println( "primary = " + facets.getPrimary() );
     Object[] array = facets.getFacets();
     for( int i = 0; i < array.length; i++ )
          System.out.println( "facet " + i + " = " + array[ i ] );
     System.out.println("Agent is ready...");
   }

  public void addComp( IComp comp )
   {
       tabela.addElement( comp );
       System.out.println("Componente adicionado!");    }

  public Vector getVector()
  {        return tabela;   }

  public void getComp(String name )
   {
      Comp aux;
      int tam = tabela.size();
```

```
        System.out.println("Table com: " + tam + " comp");
        for (int i = 0; i < tam; i++)
         {
           aux = (Comp) tabela.elementAt(i);
           String n1 = aux.getName();
           System.out.println(n1);
           if (n1.equals(name))
           System.out.println("Encontrou!");
         }
    }

    public int getRedundancia_Comp(String name )
    {
        Comp aux;
        int counter = 0; // contador para verificar ocorrencias de um mesmo componente
        int tam = tabela.size();
        System.out.println("Table com: " + tam + " comp");
        for (int i = 0; i < tam; i++)
         {
           aux = (Comp) tabela.elementAt(i);
           String n1 = aux.getName();
           System.out.println(n1);
           if (n1.equals(name)) counter++;
         }
        return counter;
    }

    public String removeComp(String name )
    {
        Comp aux;
        String NULL = "null";
        boolean encontrou = false;
        int tam = tabela.size();
        System.out.println("Table com: " + tam + " comp");
        for (int i = 0; i < tam; i++)
         {
           aux = (Comp) tabela.elementAt(i);
           String n1 = aux.getName();
           System.out.println(n1);
           if (n1.equals(name))
           {
              encontrou = true;
              System.out.println("Encontrou!");
              tabela.removeElementAt(i);
              System.out.println("Componente: " + n1 + " removido da tabela!");
              i = tam; //após encontrar PARAR!!!
           }
         }
        if(!encontrou) return NULL;
        else           return "OK";
    }

  // getComp_by_parametro retorna um inteiro com o total de conexoes de um componente
  // procurado. A String parametro pode ser direcionada para outra medida
     public int getComp_by_parametro(String name,String parametro)
    {
        Comp aux;
        int total_conexao = 0;
        int nao_encontrou = 99999;
        boolean encontrou = false;

        int tam = tabela.size();
        System.out.println("Table com: " + tam + " comp");
        for (int i = 0; i < tam; i++)
         {
           aux = (Comp) tabela.elementAt(i);
           String n1 = aux.getName();
```

```
            System.out.println(n1);
            if (n1.equals(name))
            {
               encontrou = true;
               System.out.println("Componente encontrado!");
               System.out.println("avaliando parametro...");
               if(parametro.equals("NUM_CONEXAO"))
               {
                  int x1 = aux.getcon_patras();//pegando o numero de conexoes por no
                  int x2 = aux.getcon_pelion();//pegando o numero de conexoes por no
                  int x3 = aux.getcon_andros();//pegando o numero de conexoes por no
                  total_conexao = x1 + x2 + x3;
               }
            }
          }

        if(encontrou)return total_conexao;
        else         return nao_encontrou;
    }

// getAddress_Comp retorna a String contendo o PATH completo apontando para o componente
// procurado. O parametro name é o nome do componente a ser recuperado
    public String getAddress_Comp(String name )
    {
        Comp aux;
        String path1;
        String path2;
        String path3;
        String path_retorno = "";
        String NULL = "null";// para o caso de o componente nao ser encontrado, retorna null
        boolean encontrou = false;

        int tam = tabela.size();
        System.out.println("Table com: " + tam + " comp");
        for (int i = 0; i < tam; i++)
         {
           aux = (Comp) tabela.elementAt(i);
           String n1 = aux.getName();
           System.out.println(n1);
           if (n1.equals(name))
           {
              encontrou = true;
              System.out.println("Componente encontrado!");
              System.out.println("Retornando endereco...");
              path1 = aux.getName();
              path2 = aux.getIp_addr();
              path3 = aux.getPort_num();
              path_retorno = "//" + path2 + ":" + path3 + "/" + path1;
           }
         }
        if(encontrou)return path_retorno;
        else         return NULL;
    }

// getAddress_Comp retorna a String contendo o PATH completo apontando para o componente
// procurado. O parametro name é o nome do componente a ser recuperado
public String getConexao_Comp(String name, String origem, IHistory historico )
    {
        Comp aux;
        String cod_operacao = "CONEXAO";
        String path1;
        String path2;
        String path3;
        String path_retorno = "";
        String NULL = "null";// para o caso de o componente nao ser encontrado, retorna null
        boolean encontrou = false;
```

```
    IClock clock = Clock.of(tabela); //tentando acesso a faceta de TimeStamp
    int time = clock.getHour();//pegando hora
    Line line_aux = new Line(name,cod_operacao,time,origem);
    int tam = tabela.size();
    System.out.println("Table com: " + tam + " comp");
    for (int i = 0; i < tam; i++)
     {
       aux = (Comp) tabela.elementAt(i);
       String n1 = aux.getName();
       System.out.println(n1);
       if (n1.equals(name))
       {
          encontrou = true;
             path1 = aux.getName();
          path2 = aux.getIp_addr();
          path3 = aux.getPort_num();
          path_retorno = "//" + path2 + ":" + path3 + "/" + path1;
       }
     }
    if(encontrou)return path_retorno;
    else         return NULL;
}

public void updateNum_conexao_Patras(String name, int novo_num_conexao)
{
   Comp aux;
    int old_value;
    int tam = tabela.size();
    System.out.println("Table com: " + tam + " comp");
    for (int i = 0; i < tam; i++)
     {
       aux = (Comp) tabela.elementAt(i);
       String n1 = aux.getName();
       System.out.println(n1);
       if (n1.equals(name))
       {
          old_value = aux.getcon_patras();
          System.out.println("Numero atual:" + old_value);
          old_value += novo_num_conexao;
          aux.setcon_patras(old_value);
          old_value = aux.getcon_patras();
          System.out.println("Numero atualizado:" + old_value);
       }
     }
}

public void updateNum_conexao_Pelion(String name, int novo_num_conexao )
{
    Comp aux;
    int old_value;
    int tam = tabela.size();
    System.out.println("Table com: " + tam + " comp");
    for (int i = 0; i < tam; i++)
     {
       aux = (Comp) tabela.elementAt(i);
       String n1 = aux.getName();
       System.out.println(n1);
       if (n1.equals(name))
       {
          old_value = aux.getcon_pelion();
          System.out.println("Numero atual:" + old_value);
          old_value += novo_num_conexao;
          aux.setcon_pelion(old_value);
          old_value = aux.getcon_pelion();
          System.out.println("Numero atualizado:" + old_value);
       }
     }
```

```
   }

public void updateNum_conexao_Andros(String name, int novo_num_conexao )
   {
      Comp aux;
      int old_value;
      int tam = tabela.size();
      System.out.println("Table com: " + tam + " comp");
      for (int i = 0; i < tam; i++)
       {
         aux = (Comp) tabela.elementAt(i);
         String n1 = aux.getName();
         System.out.println(n1);
         if (n1.equals(name))
         {
            old_value = aux.getcon_andros();
            System.out.println("Numero atual:" + old_value);
            old_value += novo_num_conexao;
            aux.setcon_andros(novo_num_conexao);
            old_value = aux.getcon_andros();
            System.out.println("Numero atualizado:" + old_value);
         }
       }
   }

  static public ICompTable get( Object object )
   {
   // método de conveniência para agregação dinâmica
   return (ICompTable) com.objectspace.voyager.Facets.get( object, ICompTable.class );
   }
  static public ICompTable of( Object object ) throws ClassCastException
   {
   // método de conveniência para agregação dinâmica
   return (ICompTable) com.objectspace.voyager.Facets.of( object, ICompTable.class );
   }
  }
```

1.6 – Interface IAccount.class

```
package douto.tms_age;

public interface IAccount
  {
  void deposit( int amount );
  void withdraw( int amount );
  int getBalance();
  }
```

1.7 – Classe Account.class

```
package douto.tms_age;

public class Account implements IAccount
  {
   int balance;

  public String toString()
  {    return "Account( " + balance + " )";  }

  public void deposit( int amount )
  {    balance += amount;  }

  public void withdraw( int amount )
    {
    if( amount > balance )
      throw new IllegalArgumentException( "only have " + amount );
```

```
    balance -= amount;
    }

  public int getBalance()
    {    return balance;    }
}
```

1.8 – Interface ISecurity.class

```
package douto.tms_age;

public interface ISecurity
{
  int getAmount();
  void setAmount( int access );
  void setStatus(boolean status );
  boolean getStatus();
  String getCode();
  void setCode( String code );
  long getPerformance();
  void setPerformance( long perf );
 }
```

1.9 – Classe Security.class

```
package douto.tms_age;

public class Security implements ISecurity
{
  boolean is_active;
  String code;
  int amount_access;
  long performance;

  public boolean getStatus()
    {    return is_active;    }

  public void setStatus( boolean status )
    {    this.is_active = status;    }

  public String getCode()
    {    return code;    }

  public void setCode( String code )
    {    this.code = code;    }

  public int getAmount()
    {    return amount_access;    }

  public void setAmount( int access )
    {    this.amount_access = access;    }

  public long getPerformance()
    {    return performance;    }

  public void setPerformance( long perf )
    {    this.performance = perf;    }

  public String toString()
    {    return "Security:( " + is_active + ", " + code + " )";    }

  static public ISecurity get( Object object )

   {
    // método de conveniência para agregação dinâmica
    return (ISecurity) com.objectspace.voyager.Facets.get( object, ISecurity.class );
```

```
   }

  static public ISecurity of( Object object ) throws ClassCastException
   {
   // método de conveniência para agregação dinâmica
   return (ISecurity) com.objectspace.voyager.Facets.of( object, ISecurity.class );
   }
 }
```

1.10 – Interface IClock.class

```
package douto.tms_age;
import com.objectspace.lib.timer.*;
import com.objectspace.voyager.*;

public interface IClock
{
  void setSecond( int second );
  int getHour();
}
```

1.11 – Classe Clock.class

```
package douto.tms_age;
import com.objectspace.lib.timer.*;
import com.objectspace.voyager.*;

public class Clock implements IClock
{
  static int hour;

  public String toString()
    {    return "Clock( " + hour + " )";    }

  public Clock()
  {    System.out.println("Clock criado...");  }

  public void setSecond( int second )
  {    hour = second;  }

  public int getHour()
    {     return hour;    }

  static public IClock get( Object object )
    {// método de conveniência para agregação dinâmica
    return (IClock) com.objectspace.voyager.Facets.get( object, IClock.class );
    }

  static public IClock of( Object object ) throws ClassCastException
   {// método de conveniência para agregação dinâmica
    return (IClock) com.objectspace.voyager.Facets.of( object, IClock.class );
    }
 }
```

1.12 – Classe TimeStamp.class

```
package douto.tms_age;
import java.io.*;
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.lib.facets.*;

public class TimeStamp
{
  public static int counter_second = 0;
```

```
  public static void main( String[] args )
    {
      try
      {
      Voyager.startup("8888");
      ICompTable tab_pe1 = (ICompTable) Namespace.lookup(
                                "//150.163.20.19:8002/SuperAgentPelion" );
      IClock clock_1             = Clock.get(tab_pe1);
      ICompTable tab_pe2 = (ICompTable) Namespace.lookup(
                               "//150.163.20.7:8002/SuperAgentPelionExt" );
      IClock clock_2             = Clock.get(tab_pe2);
      ICompTable tab_pe3 = (ICompTable) Namespace.lookup(
                                "//150.163.20.4:8002/SuperAgentPelionExt" );
      IClock clock_3             = Clock.get(tab_pe3);
      ICompTable tab_pa1 = (ICompTable) Namespace.lookup(
                               "//150.163.20.7:8000/SuperAgentPatras" );
      IClock clock_4             = Clock.get(tab_pa1);
      ICompTable tab_pa2 = (ICompTable) Namespace.lookup(
                              "//150.163.20.19:8000/SuperAgentPatrasExt" );
      IClock clock_5             = Clock.get(tab_pa2);
      ICompTable tab_pa3 = (ICompTable) Namespace.lookup(
                               "//150.163.20.4:8000/SuperAgentPatrasExt" );
      IClock clock_6             = Clock.get(tab_pa3);
      ICompTable tab_an1 = (ICompTable) Namespace.lookup(
                               "//150.163.20.4:8001/SuperAgentAndros" );
      IClock clock_7             = Clock.get(tab_an1);
      ICompTable tab_an2 = (ICompTable) Namespace.lookup(
                              "//150.163.20.19:8001/SuperAgentAndrosExt" );
      IClock clock_8             = Clock.get(tab_an2);
      ICompTable tab_an3 = (ICompTable) Namespace.lookup(
                               "//150.163.20.7:8001/SuperAgentAndrosExt" );
      IClock clock_9             = Clock.get(tab_an3);

  for(;;)
      {
       try{ Thread.sleep( 1000 ); } catch( InterruptedException exception ) {}
       counter_second++;
       clock_1.setSecond(counter_second);
       clock_2.setSecond(counter_second);
       clock_3.setSecond(counter_second);
       clock_4.setSecond(counter_second);
       clock_5.setSecond(counter_second);
       clock_6.setSecond(counter_second);
       clock_7.setSecond(counter_second);
       clock_8.setSecond(counter_second);
       clock_9.setSecond(counter_second);
      }
    }
    catch( Exception exception )
    {
      System.err.println( exception );
    }
  }
}
```

2 – AGENTE HISTÓRICO

2.1 – Classe HistAgente.class

```
package douto.tms_age;
import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;
import java.util.*;
import java.lang.*;

public class HistAgente
  {
```

```
  public static void main( String[] args )
   {
    try
     {
      Voyager.startup( "11111" );
      String hisClass = History.class.getName();

      // criando agente historico para o host Pelion
     Object[] args2 = new Object[]{"HisPelion", "YYY",new Integer(5), "Pe"};
     IHistory his_pelion = (IHistory)Factory.create(hisClass,
                           args2,"//150.163.20.19:9992/HisPelion");
     ISecurity sec_pe = Security.of(his_pelion);
     sec_pe.setStatus(true);
     sec_pe.setAmount(0);

     // criando agente historico para o host Patras
     Object[] args3 = new Object[]{"HisPatras", "XXX",new Integer(10), "Pa"};
     IHistory his_patras = (IHistory)Factory.create(hisClass,
                           args3,"//150.163.20.7:9990/HisPatras");
     ISecurity sec_pa = Security.of(his_patras);
     sec_pa.setStatus(true);
     sec_pa.setAmount(0);

    // criando agente historico para o host Andros
    Object[] args4 = new Object[]{"HisAndros", "WWW",new Integer(15), "An"};
    IHistory his_andros = (IHistory)Factory.create(hisClass,
                           args4,"//150.163.20.4:9991/HisAndros");
     ISecurity sec_and = Security.of(his_andros);
     sec_and.setStatus(true);
     sec_and.setAmount(0);

     // Todos os agentes historicos instanciados com sucesso
     System.out.println("Historicos inicializados: operando...");
     }
   catch( Exception exception )
     {
     System.err.println( exception );
     }
   System.out.println("Voyager: shutdown...");
   Voyager.shutdown();
   }
 }
```

2.2 – Interface ILine.class

```
package douto.tms_age;

public interface ILine
{
   String getName();
   void   setName(String novo_name);
   String getOrigem();
   void   setOrigem(String novo_origem);
   int    getTime();
   void   setTime(int novo_time);
   String getCod_operacao();
   void   setCod_operacao(String novo_op);
}
```

2.3 – Classe Line.class

```
package douto.tms_age;
import java.io.*;
import java.util.*;
import java.lang.*;
```

```
public class Line implements ILine, Serializable
 {
  String name;
  String cod_operacao;
  int    time;
  String origem;

  public Line( String name, String cod_operacao, int time, String origem)
  {
   this.name         = name;
   this.cod_operacao  = cod_operacao;
   this.time         = time;
   this.origem       = origem;
  }

 public String toString()
 {    return "Linha( " + name + " )";  }

 public void setName(String novo_name)
 {    this.name = novo_name;   }

 public String getName()
 {    return name;  }

 public void setOrigem(String novo_origem)
 {    this.origem = novo_origem;  }

 public String getOrigem()
 {    return origem;  }

 public String getCod_operacao()
 {    return cod_operacao;  }

 public void setCod_operacao(String novo_op)
 {    this.cod_operacao = novo_op;  }

 public int getTime()
 {    return time;  }

 public void setTime(int novo_time)
 {    this.time = novo_time;  }
}
```

2.4 – Interface IHistory.class

```
package douto.tms_age;
import java.util.Vector;

public interface IHistory
{
   void addLine( ILine line );
   void getOneline( String name );
   void getAll_Lines();
   int getRedundancia_Line(String name );
   String removeLine(String name, String cod_oper, int time );
   Vector getVector();
}
```

2.5 – Classe History.class

```
package douto.tms_age;
import java.util.Vector;
import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;
```

```
import java.util.*;
import java.lang.*;

public class History extends Line implements IHistory
{
 public String Nome_para_tabela;
 Vector historico = new Vector();

 public History(String name, String cod_operacao, int time, String origem)
 {
   super( name, cod_operacao, time, origem );
   System.out.println("History is ready...");
  }

  public void addLine( ILine line )
  {
    historico.addElement( line );
    System.out.println("Linha adicionada!");
   }

  public Vector getVector()
  {     return historico;  }

  public void getOneline(String name )
  {
     Line aux;
     int tam = historico.size();
     System.out.println("History com: " + tam + " comp");
     for (int i = 0; i < tam; i++)
          {
            aux = (Line) historico.elementAt(i);
            String n1 = aux.getName();
            System.out.println(n1);
            if (n1.equals(name))
            System.out.println("Encontrou!");
          }
  }

  public void getAll_Lines()
  {
     Line aux;
     int tam = historico.size();
     System.out.println("History com: " + tam + " comp");
     for (int i = 0; i < tam; i++)
          {
            aux = (Line) historico.elementAt(i);
            String na = aux.getName();
            String nb = aux.getCod_operacao();
            int    ta = aux.getTime();
            String    oa = aux.getOrigem();
            //String n1 = aux.getName();
            System.out.println("Line" + " " + i + ": " + na + " -" + nb + " -" + ta + " -" + oa);
          }
    }

  public int getRedundancia_Line(String name )
  {
     Line aux;
     int counter = 0; // contador para verificar ocorrencias de um mesmo componente
     int tam = historico.size();
     System.out.println("History com: " + tam + " comp");
     for (int i = 0; i < tam; i++)
          {
            aux = (Line) historico.elementAt(i);
            String n1 = aux.getName();
            System.out.println(n1);
            if (n1.equals(name)) counter++;
```

```
            //System.out.println("Encontrou!");
          }
        return counter;
  }

  public String removeLine(String name, String cod_oper, int time )
  {
    Line aux;
    String NULL = "null";
    boolean encontrou = false;
    int tam = historico.size();
    System.out.println("History com: " + tam + " comp");
    for (int i = 0; i < tam; i++)
        {
          aux = (Line) historico.elementAt(i);
          String n1 = aux.getName();
          String n2 = aux.getCod_operacao();
          int    n3 = aux.getTime();

          System.out.println(n1);
          if ( (n1.equals(name)) && (n2.equals(cod_oper)) && (n3 == time) )
          {
             encontrou = true;
             System.out.println("Encontrou!");
             historico.removeElementAt(i);
             System.out.println("Line: " + n1 + " removido da tabela!");
             i = tam; //após encontrar PARAR!!!
          }
        }
      if(!encontrou) return NULL;
      else           return "OK";
   }

  static public IHistory get( Object object )
  {
  // método de conveniência para agregação dinâmica
    return (IHistory) com.objectspace.voyager.Facets.get( object, IHistory.class );
  }

  static public IHistory of( Object object ) throws ClassCastException
  {
  // método de conveniência para agregação dinâmica
    return (IHistory) com.objectspace.voyager.Facets.of( object, IHistory.class );
  }
}
```

3 – AGENTE MONITOR DE OBJETOS

3.1 – Classe MonitAgente.class

```
package douto.tms_age;
import com.objectspace.voyager.*;
import java.io.*;
import java.util.*;
import java.lang.*;

public class MonitAgente
{
   public static void main( String[] args )
   {
      Vector his1 = new Vector();
      IHistory historico;
      String name = args[0];
      String hist = args[1];
      int con_pa = 0;
      int con_pe = 0;
      int con_an = 0;
```

```
        boolean alterou = false;
        try
        {
          Voyager.startup();
           System.out.println( "Voyager started locally!\n");

           //linha adicionada apenas para inicializar o historico
           historico = (IHistory) Namespace.lookup( "//150.163.20.7:9990/HisPatras" );

           if(hist.equals("patras"))
           {
              System.out.println( "acessando historico Patras!\n");
              historico = (IHistory) Namespace.lookup( "//150.163.20.7:9990/HisPatras" );
           }

           if(hist.equals("pelion"))
           {
              System.out.println( "acessando historico Pelion!\n");
              historico = (IHistory) Namespace.lookup( "//150.163.20.19:9992/HisPelion" );
           }

          if(hist.equals("andros"))
           {
              System.out.println( "acessando historico Andros!\n");
              historico = (IHistory) Namespace.lookup( "//150.163.20.4:9991/HisAndros" );
           }

// A variavel origem vai ter o valor da string hist proveniente de um dos parametros args[] do inicio
// do programa. Desta forma, quando for inicializar o programa posso especificar qual historico trabalhar e
// qual roteiro vai ser definido para atualizacao das tabelas. NAO ESQUECER QUE QUANDO CRIO O
//AGENTE HISGENT, SEU CONSTRUTOR EXIGE A STRING ROTA PARA DEFINICAO DO ITINERARIO

           //String   origem   = System.getProperty("user.name");
           String origem = hist;
          Object[] arg = new Object[]{origem};
           for(;;)
           {
            System.out.println("launch Hisgent each 20 sec to verify: " + name);
            System.out.println("Work in: " + hist);
            try{ Thread.sleep( 20000 ); } catch( InterruptedException exception ) {}

            Line nova_line1;
            his1 = historico.getVector();
            int tam1 = his1.size();
            if(tam1 != 0)
            {
              con_pa = 0;
              con_pe = 0;
              con_an = 0;
              System.out.println("History: " + hist + " com: " + tam1 + " lines");
              for (int i = 0; i < tam1; i++)
              {
                nova_line1 = (Line) his1.elementAt(i);
                String na1 = nova_line1.getName();
                String nb1 = nova_line1.getCod_operacao();
                if((na1.equals(name)) && (nb1.equals("CONEXAO")))
                {
                  alterou = true; //variavel true pois o agente deve ser lancado
                  int    ta1 = nova_line1.getTime();
                  String oa1 = nova_line1.getOrigem();
                  System.out.println("Line" + " " + i + ": " + na1 + " -" + nb1 + " -" + ta1 + " -" +
                                     oa1);
                  if((oa1.equals("PATRAS")))
                     con_pa++;
                  if((oa1.equals("PELION")))
                     con_pe++;
                  if((oa1.equals("ANDROS")))
```

```
                    con_an++;
                //apos atualizar conexao devemos eliminar linha do historico
                String eliminou_line = historico.removeLine(na1, nb1, ta1 );
                if(eliminou_line.equals("OK"))
                  System.out.println("Remocao de linha efetuada!");
                if(eliminou_line.equals("null"))
                  System.out.println("Remocao SEM EFEITO!");
              }
            }
            if(alterou)
            {
               IHisgent myagent = (IHisgent) Factory.create( Hisgent.class.getName(), arg);
               System.out.println( "I am leaving: \n");
               System.out.println( "Atualizar com: " + "(" + con_pa + " -" + con_pe + " -" +
                                   con_an + ")" );
               myagent.go(name, con_pa, con_pe, con_an);
            }
          }
        }
      }
      //error handling
      catch( Exception exception )
      {
         System.err.println( exception);
         System.out.println
              ("Failure before reaching the test-environment.\n");
      }

      // end of program
  }
}
```

4 – AGENTES MÓVEIS ESPECÍFICOS

4.1 – Interface ITraderADD.class

```
package douto.tms_age;

public interface ITraderADD
{
   void go(String hello, Comp novo_comp, String no);
   void defineRota(String no);
   String data(String hello);
}
```

4.2 – Classe TraderADD.class

```
package douto.tms_age;

import java.io.*;
import java.util.*;
import java.lang.*;
import com.objectspace.lib.timer.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;
import com.objectspace.voyager.agent.*;
import com.objectspace.lib.facets.*;

public class TraderADD implements ITraderADD, Serializable
{
   String machines         = "none ";
   String patras_user       = " ";
   String pelion_user       = " ";
   String andros_user       = " ";
   String patras_time       = " ";
   String pelion_time       = " ";
```

```
String andros_time       = " ";
String patras_agent_local      = "//150.163.20.7:8000/SuperAgentPatras";
String patras_agent_na_pelion  = "//150.163.20.19:8000/SuperAgentPatrasExt";
String patras_agent_na_andros  = "//150.163.20.4:8000/SuperAgentPatrasExt";
String pelion_agent_local      = "//150.163.20.19:8002/SuperAgentPelion";
String pelion_agent_na_patras  = "//150.163.20.7:8002/SuperAgentPelionExt";
String pelion_agent_na_andros  = "//150.163.20.4:8002/SuperAgentPelionExt";
String andros_agent_local      = "//150.163.20.4:8001/SuperAgentAndros";
String andros_agent_na_pelion  = "//150.163.20.19:8001/SuperAgentAndrosExt";
String andros_agent_na_patras  = "//150.163.20.7:8001/SuperAgentAndrosExt";

String patras_time_agent = " ";
String pelion_time_agent = " ";
String andros_time_agent = " ";
String return_time       = " ";

// definicao de variaveis para interface com os agentes supervisores (Tabelas)
ICompTable ag_sup_patras_local;
ICompTable ag_sup_patras_na_pelion;
ICompTable ag_sup_patras_na_andros;

public Comp comp_aux;// para armazenar o conteudo do novo componente a ser adicionado
Stopwatch my_watch     = new Stopwatch();

public TraderADD()
{
 System.out.println( "Hello Supervisor Agent constructed.\n" );
}

public void finalize()
{
 String end = data("Agent TraderADD is finalized and data is destroyed at");
}

public String data(String hello)
{
   System.out.println("\n" + hello + ": ");
   GregorianCalendar cal = new GregorianCalendar();
   cal.setTimeZone (TimeZone.getTimeZone( "ECT+1"));
   String actual_date = "Local time:  "
        +cal.get (Calendar.HOUR_OF_DAY)
        +":"
        +cal.get (Calendar.MINUTE)
        +":"
        +cal.get (Calendar.SECOND)
        +":"
        +cal.get (Calendar.MILLISECOND)+"\n";
   System.out.println( actual_date );
   return actual_date;
}

public void write_file( String hello)
{
   FileWriter f1;
   try
   {
      f1 = new FileWriter("hello.message");
      f1.write(hello);
      f1.close();
   }
   catch (IOException exception )
   {
      String message = "Error while writing file: "+hello;
      System.out.println(message);
      try
      {
        IAlarm alarm = (IAlarm) Factory.create(IAlarm.class.getName());
```

```
          Object[] args = new Object[] {exception, message};
          OneWay.invoke( alarm, "alert", args);
        }
        catch (Exception exception2 )
        {
           System.err.println (exception2);
        }
    }
}

public void go(String hello, Comp novo_comp, String no)
{
 defineRota(no); //definindo as tabelas a serem atualizadas
 System.out.println( "Before action, starting StopWatch.\n");
 comp_aux = novo_comp;// Lembrar-se que para qualquer alteracao, deve se referir a
                      // comp_aux ao inves de comp
    my_watch.start();
    try
    {
     Agent.of(this).moveTo(ag_sup_patras_local, "atpatras");
    }
    catch ( Exception exception )
    {
      System.err.println( exception );
      System.out.println("Error moving to Patras. \n");
      Agent.of(this).setAutonomous(false);
    }
}

public void atpatras(ICompTable tabela )
{
 //Adicionar componente na tabela
 tabela.addComp(comp_aux);
 machines     =  "Patras.";
 patras_user  = System.getProperty("user.name");
 patras_time  = data("I am at Patras. \n");
 write_file("I was at Patras!");

 //move to pelion
 try
 {
   Agent.of(this).moveTo( ag_sup_patras_na_pelion, "atpelion" );
  }
 catch ( Exception exception )
 {
  System.err.println( exception );
  System.out.println("Error moving to Pelion. \n");
  try
     {
      IAlarm alarm = (IAlarm) Factory.create(IAlarm.class.getName());
      Object[] args = new Object[] {exception, "moving to Pelion"};
      OneWay.invoke( alarm, "alert", args);
      Agent.of(this).setAutonomous(false);
      }
  catch (Exception exception2 )
  {
   System.err.println (exception2);
  }
 }
}
public void atpelion(ICompTable tabela2)
{
 tabela2.addComp(comp_aux);
 machines     =  "Patras and Pelion.";
 pelion_user  = System.getProperty("user.name");
 pelion_time  = data("I am at Pelion. \n");
 write_file("I was at Pelion!");
```

```
    //move to andros
    try
    {
     Agent.of(this).moveTo( ag_sup_patras_na_andros, "atandros" );
    }
    catch ( Exception exception )
    {
     System.err.println( exception );
     System.out.println("Error moving to Andros. \n");
     try
       {
        IAlarm alarm = (IAlarm) Factory.create(IAlarm.class.getName());
        Object[] args = new Object[] {exception, "moving to Andros"};
        OneWay.invoke( alarm, "alert", args);
        Agent.of(this).setAutonomous(false);
        }
     catch (Exception exception2 )
        {
         System.err.println (exception2);
        }
    }
}

public void atandros(ICompTable tabela3)
{
 tabela3.addComp(comp_aux);
 machines     =  "Patras, Pelion and Andros.";
 andros_user  = System.getProperty("user.name");
 andros_time  = data("I am at Andros. \n");
 write_file("I was at Andros!");

 //move home
 try
    {
     Agent.of(this).moveTo( "tcp://150.163.20.7:9999", "athome" );
    }
 catch ( Exception exception )
 {
  System.err.println( exception );
  System.out.println("Error moving to" + Agent.of(this).getHome() +" \n ");
  try
     {
      IAlarm alarm = (IAlarm) Factory.create(IAlarm.class.getName());
      Object[] args = new Object[] {exception, "moving home"};
      OneWay.invoke( alarm, "alert", args);
      Agent.of(this).setAutonomous(false);
     }
   catch (Exception exception2 )
    {
     System.err.println (exception2);
     }
    }
}

public void athome()
{
 //stop stopwatch and present collected data
 return_time = data("I am back at Patras, stopping Stopwatch. \n");
 my_watch.stop();
 System.out.println("I was at: " + machines + "\n");
 System.out.println("Mission time: "  +my_watch.getLapTime() +" Milliseconds. \n");
 System.out.println("At " +patras_time +" user "
                    + patras_user
                    +" was logged in at Patras and running the server.\n");
 System.out.println("At "+pelion_time +" user " +pelion_user
                    +" was logged in at Pelion and running the server.\n");
```

```
    System.out.println("At "+andros_time +" user " +andros_user
                       +" was logged in at Andros and running the server.\n");

    //allow garbage collection
    Agent.of(this).setAutonomous(false);
   }

   public void defineRota(String no)
   {
    if(no.equals("Patras"))
         {
          try
          {
            ag_sup_patras_local         =(ICompTable) Namespace.lookup(patras_agent_local);
            ag_sup_patras_na_pelion  =(ICompTable) Namespace.lookup(patras_agent_na_pelion);
            ag_sup_patras_na_andros =(ICompTable)
                                        Namespace.lookup(patras_agent_na_andros);
          }
          catch( Exception exception )
          {
            System.err.println( exception );
            System.out.println("Possivel Erro: Patras agente supervisor fora do ar!!");
          }
         }

       if(no.equals("Pelion"))
         {
          try
          {
            ag_sup_patras_local         =(ICompTable) Namespace.lookup(pelion_agent_local);
            ag_sup_patras_na_pelion  =(ICompTable) Namespace.lookup(pelion_agent_na_patras);
            ag_sup_patras_na_andros =(ICompTable)
                                        Namespace.lookup(pelion_agent_na_andros);
          }
          catch( Exception exception )
          {
            System.err.println( exception );
            System.out.println("Possivel Erro: Pelion agente supervisor fora do ar!!");
          }
         }
       if(no.equals("Andros"))
         {
          try
          {
           ag_sup_patras_local         =(ICompTable) Namespace.lookup(andros_agent_local);
           ag_sup_patras_na_pelion  =(ICompTable) Namespace.lookup(andros_agent_na_pelion);
           ag_sup_patras_na_andros =(ICompTable)
                                       Namespace.lookup(andros_agent_na_patras);
          }
          catch( Exception exception )
          {
           System.err.println( exception );
           System.out.println("Possivel Erro: Andros agente supervisor fora do ar!!");
          }
         }
   }
}
```

4.3 – Interface ITraderHist.class

```
package douto.tms_age;

public interface ITraderHist
{   void go(String name, int conec_pa, int conec_pe, int conec_an);}
```

4.4 – Classe TraderHist.class

```
package douto.tms_age;
import java.util.Vector;
import com.objectspace.lib.facets.*;
import java.io.*;
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import java.lang.*;

public class TraderHist implements ITraderHist, Serializable
{

   String[] machine = new String[3];
   ICompTable tabela1;
   ICompTable tabela2;
   ICompTable tabela3;
   Vector tab_aux = new Vector();  //é executado em cada máquina!
   Vector his     = new Vector();
   String name_aux;
   int con_pa = 0;
   int con_pe = 0;
   int con_an = 0;

   public  TraderHist(String rota)
   {
      if((rota.equals("PATRAS")) || (rota.equals("patras")))
      {
         System.out.println("Tentando definir rota Patras");
         machine[0]= "//150.163.20.7:8000/SuperAgentPatras";
         machine[1]= "//150.163.20.4:8000/SuperAgentPatrasExt";
         machine[2]= "//150.163.20.19:8000/SuperAgentPatrasExt";
       }
      if((rota.equals("PELION")) || (rota.equals("pelion")))
        {
         System.out.println("Tentando definir rota Pelion");
         machine[0]= "//150.163.20.19:8002/SuperAgentPelion";
         machine[1]= "//150.163.20.4:8002/SuperAgentPelionExt";
         machine[2]= "//150.163.20.7:8002/SuperAgentPelionExt";
       }
      if((rota.equals("ANDROS")) || (rota.equals("andros")))
        {
         System.out.println("Tentando definir rota Andros");
         machine[0]= "//150.163.20.4:8001/SuperAgentAndros";
         machine[1]= "//150.163.20.7:8001/SuperAgentAndrosExt";
         machine[2]= "//150.163.20.19:8001/SuperAgentAndrosExt";
        }

      System.out.println( "Rota definida e acao inicializada.\n");
   }

   public void finalize()
   {    System.out.println( "Agente TraderHist is finalized.");    }

   public void go(String name, int conec_pa, int conec_pe, int conec_an)
   {
     name_aux = name;
     for(int lap = 0; lap < 3; lap++)//deixar o valor de lap sempre 3
     {
      con_pa = conec_pa;// variaveis utilizadas para atualizar a tabela
      con_pe = conec_pe;// com os dados obtidos nos historicos
      con_an = conec_an;
      try
      {
         tabela1 =(ICompTable) Namespace.lookup(machine[lap]);
         Agent.of(this).moveTo( tabela1, "atmachine" );
```

```
        }

        catch( Exception exception )
        {
          System.err.println (exception);
          System.out.println("Error moving to testnet. \n");
          Agent.of(this).setAutonomous(false);
        }
       }
    }

    public void atmachine(ICompTable tab)
    {
      System.out.println("Numeros de atualizacao: " + con_pa + con_pe + con_an);
      if(con_pa != 0)
           tab.updateNum_conexao_Patras(name_aux,con_pa);
      if(con_pe != 0)
           tab.updateNum_conexao_Pelion(name_aux,con_pe);
      if(con_an != 0)
           tab.updateNum_conexao_Andros(name_aux,con_an);
      System.out.println("I moved to: "+Proxy.of(this).getURL() +"\n");
    }
}
```

4.5 – Interface ITraderOS.class

```
package douto.tms_age;

public interface ITraderOS
{   void go();}
```

4.6 – Classe TraderOS.class

```
package douto.tms_age;
import java.io.*;
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;

public class TraderOS implements ITraderOS, Serializable
{
 String agente_supervisor[] = {"//150.163.20.7:8000/SuperAgentPatras",
                               "//150.163.20.4:8001/SuperAgentAndros",
                               "//150.163.20.19:8002/SuperAgentPelion"}


    public TraderOS()
    {
       System.out.println( "Agente para status de objetos construido.\n");
    }

    public void finalize()
    {
      System.out.println( "TraderOS is finalized.");
    }

    public void go()
    {
     //move to the environment
     try
       {
        Agent.of(this).moveTo( agente_supervisor[0], "atmachine" );
        }
     catch( Exception exception )
     {
```

```
      System.err.println (exception);
      System.out.println("Error moving to environment. \n");
      Agent.of(this).setAutonomous(false);
      }
    }

    public void atmachine(ICompTable tab)
    {
     //é executado em cada máquina!
      int loop = 0;
      System.out.println("I moved to: "+Proxy.of(this).getURL() +"\n");
      Comp novo_comp;
      tab_aux = tab.getVector();
      int tam = tab.size();
      System.out.println("Table com: " + tam + " comp");
      for (int i = 0; i < tam; i++)
       {
        aux = (Comp) tab.elementAt(i);
        String n1 = aux.getName();
        System.out.println("Retornando endereco...");
        String path1 = aux.getName();
        String path2 = aux.getIp_addr();
        String path3 = aux.getPort_num();
        String endereco_obj = "//" + path2 + ":" + path3 + "/" + path1;

        //Obj deve ser do mesmo tipo do objeto consultado
        Obj  = (Iobj)Namespace.lookup(endereco_obj);

        //Status refere-se aa faceta do objeto consultado
       IStatus status = (IStatus) Facets.of(Obj).of("douto.tms_age.IStatus");
        ...
       }

       //definir a proxima máquina randomicamente
       int loop = Math.abs(loop + 1) % 3;
       System.out.println(" I woke up and move on to " +machine[loop] + " !\n");
       //move on
       try
        {
          Agent.of(this).moveTo( machine[loop], "atmachine");
        }
       catch( Exception exception )
        {
          System.err.println (exception);
          System.out.println("Error moving to "+machine[loop] +". \n");
          Agent.of(this).setAutonomous(false);
         }
    }
}
```

4.7 – Interface ITraderPerf.class

```
package douto.tms_age;

public interface ITraderPerf
{    void go();}
```

4.8 – Classe TraderPerf.class

```
package douto.tms_age;

import java.util.Vector;
import com.objectspace.lib.facets.*;
import com.objectspace.lib.timer.*;
import java.io.*;
```

```
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import java.lang.*;

public class TraderPerf implements ITraderPerf, Serializable
{
   String[] machine = new String[3];
   ICompTable tabela;
   Vector tab_aux = new Vector();  //is executed at each machine!
   static int local;
   Stopwatch watch = new Stopwatch();

public  TraderPerf()
{
   machine[0]= "//150.163.20.7:8000/SuperAgentPatras";
   machine[1]= "//150.163.20.19:8002/SuperAgentPelion";
   machine[2]= "//150.163.20.4:8001/SuperAgentAndros";
   System.out.println( "Rota definida e acao inicializada.\n");
 }

public void finalize()
{    System.out.println( "Table-Agent is finalized.");}

public void go()
{
   watch.start();
   local = 0;
   for(int lap = 0; lap < 3; lap++)
   {
     //migrar pelo ambiente de teste
     try
       {
          System.out.println("Tentando mover...");
          tabela =(ICompTable) Namespace.lookup(machine[lap]);
          System.out.println("passei");
          local ++;
          Agent.of(this).moveTo( tabela, "atmachine" );
       }

       catch( Exception exception )
       {
          System.err.println (exception);
          System.out.println("Error moving to testnet. \n");
          Agent.of(this).setAutonomous(false);
       }
      }
   }

public void atmachine(ICompTable tab)
{
   System.out.println("Local value: "+ local);
   if (local != 3)
   {
     int ref = 0;
     Stopwatch watch2 = new Stopwatch();
     watch2.start();
     IAccount account = (IAccount) Facets.of( tab ).of( "douto.tms_age.IAccount" );
     ISecurity sec    = (ISecurity) Security.of( tab );
     ref = account.getBalance();
     for(int d = 0; d < 30; d++)
     {
      account.deposit(1);
      account.withdraw(1);
     }
     watch2.stop();
     sec.setPerformance(watch2.getTotalTime());
```

```
    System.out.println("time = " + watch2.getTotalTime() + "ms");
    System.out.println("I moved to: "+Proxy.of(this).getURL() +"\n");
   }
   else
   {
    try
       {
        Agent.of(this).moveTo( "tcp://150.163.20.7:9999", "athome" );
       }
    catch( Exception exception )
       {
         System.err.println (exception);
         System.out.println("Error moving to testnet. \n");
         Agent.of(this).setAutonomous(false);
       }
   }
}

public void athome()
{
 watch.stop();
 System.out.println("Tempo da missao: "+ watch.getTotalTime() + "ms");
 Agent.of(this).setAutonomous(false);
}
}
```

4.9 – Classe TraderAddr.class

```
package douto.tms_age;
import java.io.*;
import com.objectspace.lib.timer.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import com.objectspace.lib.facets.*;

public class TraderAddr implements  Serializable, Runnable
 {
   ICompTable tabela;
   String name_aux;
   String parametro_aux;

 public TraderAddr( ICompTable tab, String name, String parametro)
   {
   tabela = tab;
   name_aux = name;
   parametro_aux = parametro;
   System.out.println( "Trader de enderecos construido" );
   }

 public void finalize()
   {
   System.out.println( "finalize trader" );
   }
 public void run()
 {
   try{
       System.out.println( "remote trade" );
       try
       {
          Agent.of( this ).moveTo( tabela, "atTabela" ); // move to market
       }
       catch( Exception exception )
       {
        System.err.println( exception );
       }
   }
```

```
    catch(Exception e2)
    {
       System.err.println( e2);
    }
  }


  public void atTabela( ICompTable tab )
    {
    System.out.println( "at remote tabela, home = " + Agent.of( this ).getHome() );
    tradeAt( tab ); // trade with local market
    Agent.of( this ).setAutonomous( false ); // allow myself to be gc'ed
    }

  private void tradeAt( ICompTable tab )
  {
    Comp aux;
    int total_conexao = 0;
    int nao_encontrou = 99999;
    boolean encontrou = false;

    int tam = tab.size();
    System.out.println("Table com: " + tam + " comp");
    for (int i = 0; i < tam; i++)
     {
        aux = (Comp) tab.elementAt(i);
        String n1 = aux.getName();
        System.out.println(n1);
        if (n1.equals(name_aux))
          {
           encontrou = true;
           System.out.println("Componente encontrado!");
           System.out.println("avaliando parametro...");
           if(parametro_aux.equals("NUM_CONEXAO"))
             {
              int x1 = aux.getcon_patras();//pegando o numero de conexoes por no
              int x2 = aux.getcon_pelion();//pegando o numero de conexoes por no
              int x3 = aux.getcon_andros();//pegando o numero de conexoes por no
              total_conexao = x1 + x2 + x3;
             }
          }
     }

     if(encontrou)return total_conexao;
     else         return nao_encontrou;
}
```

4.10 – Interface ITraderMonitHist.class

```
package douto.tms_age;

public interface ITraderMonitHist
{
   void go();
}
```

4.11 – Classe TraderMonitHist.class

```
package douto.tms_age;
import java.util.Vector;
import com.objectspace.lib.facets.*;
import java.io.*;
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
```

```
import java.lang.*;

public class TraderMonitHist implements ITraderMonitHist, Serializable
{
   String[] historicos = new String[3];

   IHistory historico_aux;
   Vector hist_aux = new Vector();
   Vector hist     = new Vector();
   public  MonitorHist()
   {
    System.out.println("Tentando definir rota para os históricos");
    hist_addr[0]= "//150.163.20.7:9990/HisPatras";
    hist_addr[1]= "//150.163.20.4:9991/HisAndros";
    hist_addr[2]= "//150.163.20.19:9992/HisPelion";
    System.out.println( "Rota definida.\n");
   }
   public void finalize()
   {
    System.out.println( "Agente Monitor de Historicos is finalized.");
   }
   public void go()
   {
      for(int lap = 0; lap < 3; lap++)
      {
       try
       {
          historico_aux =(IHistory) Namespace.lookup(hist_addr[lap]);
          Agent.of(this).moveTo( historico_aux, "atmachine" );
       }
       catch( Exception exception )
       {

          System.err.println (exception);
          System.out.println("Error moving to historico. \n");
          Agent.of(this).setAutonomous(false);
       }
      }
   }

   public void atmachine(IHistory historico_aux)
   {
    System.out.println("Agente Monitor acessando historico");
    hist = historico_aux;
    Line nova_line;
    his = his_patras.getVector();
          int tam = his.size();
          System.out.println("Verificando History");
          System.out.println("Atualmente  com: " + tam + " lines");
          for (int j = 0; j < tam; j++)
            {
              nova_line = (Line) his.elementAt(j);
              // acessando os dados quanto a hora e origem da operaçao
              // pode-ser fazer uma analise póstuma baseado nas estatiscas
              // levantadas e processadas.
              String na = nova_line.getName();
              int    ta = nova_line.getTime();
              String oa = nova_line.getOrigem();
              if(oa.equals("PATRAS"))  con_pa++;
              if(oa.equals("PELION"))  con_pe++;
              if(oa.equals("ANDROS"))  con_an++;
             }
     System.out.println("I moved to: "+Proxy.of(this).getURL() +"\n");
   }
}
```