5
Trabalhos Relacionados

Existem diferentes tipos de *frameworks* de *middleware* para SMAs, cada um focaliza mais fortemente alguma propriedade de agencia, tal como mobilidade focalizada no *framework* SOMA (Secure and Open Mobile Agent) [23], colaboração, enfatizada no *framework* CASA (Collaborative Agent Software Architecture) [43], interoperabilidade, focalizada pela plataforma JADE [9], ou, alguma outra propriedade não funcional, tal como escalabilidade enfatizada no *framework* AgentScape [89]. Uma breve descrição das abordagens é apresentada na parte inicial do capítulo e comparações entre as abordagens são descritas na segunda parte do capítulo.

5.1
JADE – Java Agent DEvelopment

JADE é um *framework* de *middleware* cujo propósito é facilitar o desenvolvimento de aplicações baseadas em agentes interoperáveis na Internet. O objetivo é simplificar e ao mesmo tempo assegurar o desenvolvimento em conformidade com padrões, descritos através de um compreensível conjunto de serviços de sistema e agentes.

A arquitetura de JADE segue o modelo de referência FIPA (Foundation for Intelligent Physical Agents) [39], cujo princípio básico seguido por JADE é que somente o comportamento externo dos agentes deve ser especificado, deixando os detalhes da implementação e a arquitetura interna de agentes para os desenvolvedores. A Figura 57 [39] mostra o modelo de referência FIPA de uma plataforma de agentes. O modelo é composto por três principais agentes:

1. AMS (Agent Management System): é o agente que exerce supervisão e controle sobre o acesso e uso da plataforma. Ele é responsável pela autenticação e registro dos agentes residentes, e provê alguns assistentes

GUIs para efetuar as tarefas de gerenciamento que exigem interação com atores humanos.

2. ACC (Agent Communication Channel): é o agente que provê o caminho básico para a conexão entre agentes dentro ou fora da plataforma. A comunicação intra-container é feita utilizando a ACL FIPA, baseada em transporte de mensagens. A interoperabilidade entre diferentes plataformas de agentes é feita utilizando IIOP (Internet Inter-Orb Protocol).
3. DF (Directory Facilitator): é o agente que provê as páginas amarelas de serviços, que podem ser acessadas por agentes e visualizadas por desenvolvedores.

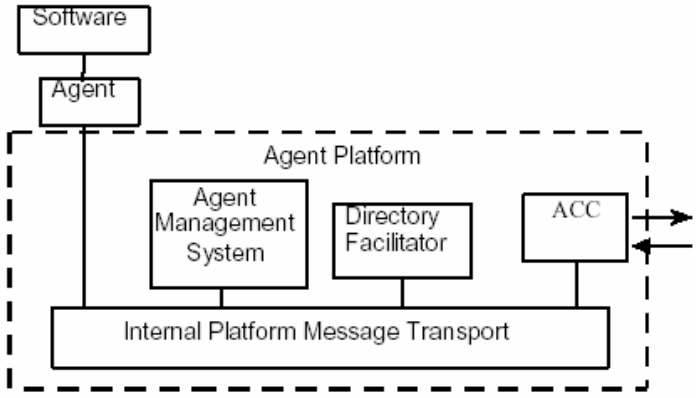

Figura 57 - Modelo de referência FIPA

Na plataforma JADE, não existe restrição quanto à tecnologia utilizada para implementação: plataforma baseada em e-mail, baseada em CORBA, baseada em aplicações *multithreading* Java, todos podem operar em conformidade com FIPA. A arquitetura de software é baseada na coexistência de várias máquinas virtuais Java e a comunicação entre diferentes máquinas virtuais é efetuada com Java RMI. Dentro de um container, a comunicação é efetuada de acordo com o protocolo padrão FIPA. Cada máquina virtual é um container básico de agentes, que provê um ambiente completo para a execução de agentes e permite vários agentes executarem concorrentemente no mesmo *host*.

A Figura 58 [9] mostra uma visão completa da plataforma e os diferentes tipos de agentes e containers utilizados em um ambiente. JADE possui um servidor de *front-end* denominado Agent Platform Front-End, que mantém um registro RMI, usado pelos outros containers de agentes para se registrarem. O

servidor de *front-end* mantém uma tabela com todos os containers e uma tabela chamada Agent Global Descriptor, que relaciona cada nome de agente com o seu container de referência. Quando o servidor de *front-end* começa a executar, ele cria um registro interno RMI no *host* corrente e os agentes FIPA ACC, AMS e DF são ativados.

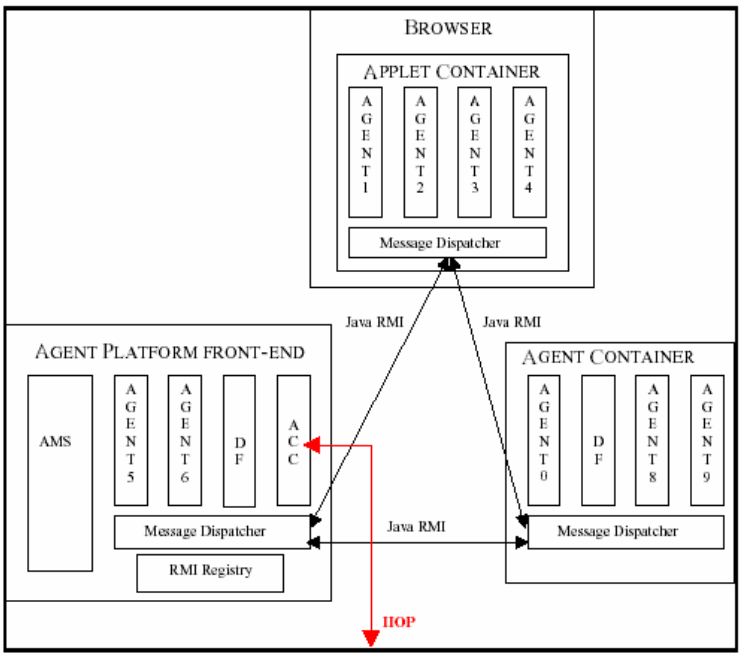

Figura 58 - Arquitetura de software de uma plataforma de agentes JADE

A plataforma apresenta uma interface com o lado externo do ambiente usando o agente padrão ACC. O agente utiliza a especificação CORBA sobre o protocolo IIOP para se comunicar com sistemas heterogêneos, e atua como um objeto servidor e um ouvinte de chamadas remotas. Cada vez que ele recebe uma mensagem encapsulada em uma *string* (usualmente enviada por agentes não JADE), ele processa a mensagem, convertendo-a em um objeto Java ACLMessage usado por todos os agentes JADE. Ele também pode efetuar a conversão contrária, quando um agente JADE envia uma mensagem para um agente não JADE. As versões mais recentes de JADE possuem extensões compatíveis com serviços Web para interoperar na Internet.

O Agent Container possibilita a JADE alcançar seu principal propósito, no qual os mecanismos de comunicação devem ser desconhecidos por todos os agentes. Cada *thread* de agente é uma classe completamente independente que

não conhece os detalhes do mecanismo de comunicação e a arquitetura da plataforma de agentes. JADE utiliza uma abstração de comportamento para modelar as tarefas que um agente está habilitado a executar. De um ponto de vista de programação concorrente, um agente é um objeto ativo, sendo executado dentro de um *thread* de controle. O desenvolvedor precisa estender a classe Agent e implementar as tarefas de comportamento específico do agente através de uma ou mais classes de comportamento.

Uma plataforma de agentes é composta de vários containers de agentes. As interfaces gráficas para gerenciamento são providas por um agente chamado RMA (Remote Monitoring Agent), que monitora e controla o status dos agentes em execução. Ele trabalha em colaboração com o agente AMS, que controla o ciclo de vida dos agentes, que possibilita criar, suspender, executar e excluir agentes.

JADE utiliza tecnologia idêntica a MIDAS para o tratamento de serviços Web. Ambas se baseiam no conceito WSA [13], que expressa claramente a noção de que, "... os agentes do software são os programas rodando que dirigem serviços Web – tanto para os executar quanto para os *acessar* como recursos computacionais que agem em nome de uma pessoa ou da organização". A solução JADE utiliza uma porta de integração com serviços Web (WSIG) [57], projetada para encapsular todas as funcionalidades requeridas para conectar as duas tecnologias, assegurando a mínima intervenção humana. A WSIG utiliza, da mesma forma que MIDAS, transformações bidirecionais entre serviços de agentes FIPA e serviços Web.

5.2
CASA – Collaborative Agent System Architecture

O *framework* CASA foi projetado pelo CAG (Collaborative Agent Group) da universidade de Calgary, Canadá, no final dos anos 90 e focaliza o desenvolvimento de aplicações baseadas em agentes colaborativos provendo facilidades para coordenação de serviços na Internet. O *framework* CASA tem sido utilizado com sucesso na área de manufatura em projetos para para o desenvolvimento de aplicações para controle de chão de fábrica. Os serviços providos pelo *framework* incluem trocas de mensagens, páginas amarelas (*lookup*, *search*) e suporte para comunicação remota.

Os conceitos básicos na arquitetura de CASA podem ser vistos na Figura 59. Uma Area é uma região delimitada que compreende recursos computacionais de propriedade de alguma instituição humana, que pode ser formada por recursos parciais de um computador, um computador inteiro ou um grupo de computadores. Os agentes se localizam em áreas que formam comunidades reguladas pelas políticas e restrições definidas pelos proprietários dos recursos. As ações de coordenação em uma área são desempenhadas por um agente denominado Local Area Coordinator (LAC). Ele atua ao mesmo tempo como um representante dos agentes locais e como uma porta para o lado externo, provendo uma interface por onde os agentes podem se comunicar com o mundo exterior. Os agentes colaboram entre si provendo e requisitando serviços.

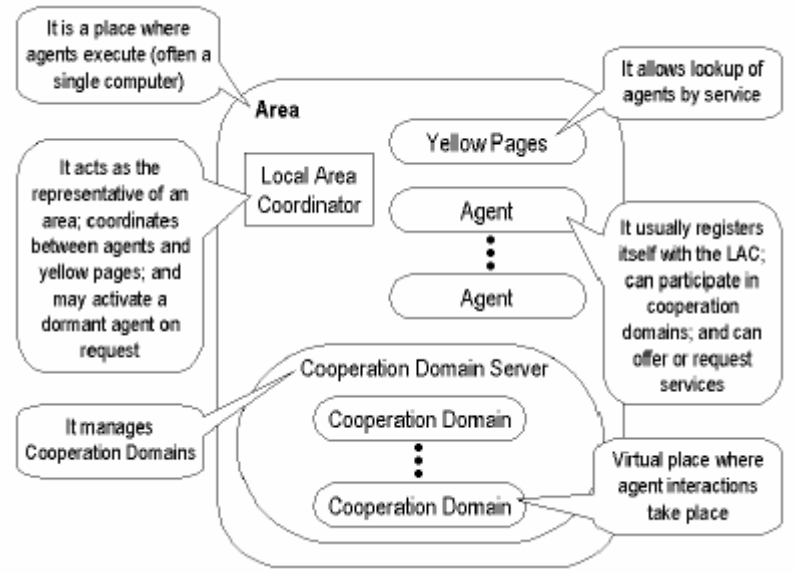

Figura 59 - Os conceitos básicos na arquitetura CASA

Os agentes que desejam manipular os recursos de uma área necessitam se registrar através do LAC, que também coordena o acesso dos agentes às páginas amarelas de serviços. O conceito de páginas amarelas (*yellow pages*) foi introduzido na abordagem para gerenciar as funções de registro e possibilitar a colaboração. Um agente do tipo Yellow Pages desempenha estas funções provendo os mecanismos para visualização e descoberta de serviços. Agentes podem fazer consultas nas páginas amarelas para conhecer o perfil dos demais agentes e determinar que agentes oferecem um determinado serviço, assim como recuperar descrições da localização deste agente e de serviços.

Cooperation Domain pode ser definido como um lugar virtual onde ocorrem interações entre agentes. Cada agente dentro de um domínio colaborativo envia mensagens através de um Cooperation Domain Server (CDS). O CDS funciona como um canal de comunicação entre os agentes distribuídos na área, que pode direcionar a mensagem a um agente específico (imitando a comunicação ponto-a-ponto) ou para todos os agentes no domínio colaborativo (imitando comunicação *broadcast*). Todas as mensagens remotas são direcionadas e recebidas por CDS, que fica ouvindo em uma determinada porta à chegada de novas conexões. Os agentes podem enviar mensagens para esta porta e a requisição é direcionada de forma transparente para o agente provedor do serviço.

Os benefícios de todas as mensagens (tanto internas quanto externas) serem manipuladas pelo servidor de domínio é que o servidor pode prover vários serviços e agentes podem endereçar papéis e serviços, não necessitando conhecer a identidade dos agentes que desempenham os papéis ou executam os serviços.As funcionalidades dos agentes são descritas através de um modelo de papéis, que especifica os serviços que o agente provê e os serviços que ele deverá requisitar. O *framework* CASA possui uma entidade denominada *minimal agent* que descreve a funcionalidade mínima de um agente. O *minimal agent* provê uma estrutura de código que define o fluxo de comunicação entre o agente e o seu LAC, habilitando-o a participar do domínio de cooperação.

5.3 O *Middleware* AgentScape

O *middleware AgentScape* foi projetado para suportar o desenvolvimento de aplicações baseadas em agentes para sistemas distribuídos na Internet. Os serviços são encapsulados e apresentados como serviços Web, tornando-se disponíveis no *middleware* para os agentes através de SOAP/WSDL. *AgentScape* utiliza a infraestrutura de *middleware* para suportar as tarefas de mobilidade, segurança e gerenciamento de comunicação remota.

A Figura 60 [89] ilustra a arquitetura em camadas, que contém um pequeno núcleo (AOS Kernel) implementando os mecanismos básicos e serviços de *middleware* de alto nível (*Agent Server*, *Host Manager*, *Location Manager*). No *framework* AgentScape, os agentes são entidades ativas, e serviços são entidades

externas (ao agente) que podem ser acessados pelos agentes hospedados no *middleware*. Agentes podem se comunicar com outros agentes e podem acessar serviços. Os serviços de alto nível podem interagir somente com o núcleo local de *middleware*. Todos os processos de comunicação entre servidores de agentes e portas de serviços Web é exclusivamente feito através do seu núcleo local AOS. O núcleo AOS pode manipular diretamente a transação (operação local) ou direcionar as mensagens de requisição para o núcleo AOS remoto (operação remota). *Agent Server* fornece uma API para acessar o *middleware*, e os agentes instanciados precisam aderir à interface definida pela API.

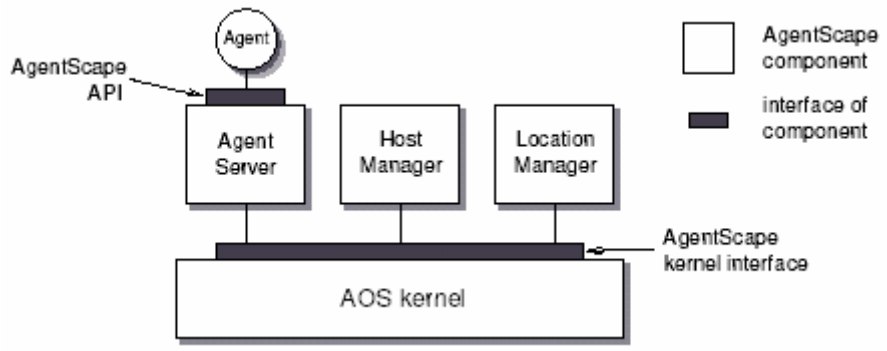

Figura 60 - A arquitetura de framework AgentScape

O *Host Manager* gerencia e coordena as atividades em um *host*; atua como um representante local do *Location Manager*, sendo também responsável pelo acesso aos recursos locais e gerenciamento destes recursos. As políticas e mecanismos da infraestrutura de gerenciamento e localização são baseados em negociação e adesão aos contratos de serviços.

Os serviços são registrados em um catálogo local, e podem ser requisitados pelos agentes, assim como formar composições de mais alto nível, que criam novos serviços. Extensibilidade e interoperabilidade com outros sistemas são realizadas através de serviços adicionais de *middleware*. A porta para serviços Web é um serviço de *middleware* que provê acesso controlado a serviços Web via SOAP e WSDL. Os agentes podem obter descrição dos serviços de várias maneiras, como, por exemplo, consultando o catálogo local ou, através de UDDI.

Um dos princípios básicos de *AgentScape* é produzir uma mínima, mas suficiente, plataforma aberta de agentes que possa ser estendida para incorporar novas funcionalidades. Esta noção adere ao conceito de container leve, cujo

objetivo é fornecer apenas as funcionalidades básicas de *middleware*, deixando tarefas tais como persistência e apresentação para outros *frameworks*. Apesar de apresentar conceitos similares aos utilizados em nossa abordagem, AgentScape focaliza um modelo de alto nível que prove pouco suporte direto para os agentes.

Interoperabilidade com sistemas externos é realizada somente através de uma porta para serviços Web, que provê acesso controlado a SOAP/WSDL. AgentScape não provê suporte adicional para integração com aplicações empresariais, e não provê mecanismos para manipulação de dados persistentes. Além do mais, o atual núcleo de AgentScape é implementado com a linguagem de programação Python. Conforme declarado pelo autor [89], as funcionalidades para manipular recursos e os serviços de descoberta e gerenciamento implementados em Pyton não são escaláveis e seguros.

5.4 SOMA – Secure and Open Mobile Agent

O foco principal do *framework* SOMA [23] é a mobilidade. Ele é baseado em uma hierarquia de abstrações de localidade capazes de modelar e descrever qualquer tipo do sistema distribuído aberto e global, variando de LANs simples à Internet [8]. Todo o nó possui ao menos um lugar que constitui o ambiente de execução do agente. Em cada domínio, um lugar *default* hospeda uma porta para executar a funcionalidade inter-domínio e para manter a informação específica do domínio em tempo de execução. SOMA é implementado em Java e fornece um conjunto de serviços tradicionais para agentes móveis e dois componentes responsáveis por garantir a segurança e a interoperabilidade com outras aplicações e serviços, independente do estilo de programação adotado.

A Figura 61 mostra os principais elementos da arquitetura do SOMA, que provê os seguintes serviços básicos:

- *Naming*: associar entidades com um identificador global único e organizar estes identificadores no sistema para possibilitar o rastreamento de agentes móveis;
- *Migration*: permitir a migração do agente móvel para diferentes ambientes de execução;

- *Communication*: provê ferramentas para coordenação e comunicação entre agentes móveis; no esmo local, os agentes interagem por meio de objetos compartilhados para cooperação;
- *Monitoring*: para observar o estado dos recursos e dos serviços locais e para fornecer esta informação ao nível da aplicação para permitir reações adaptativas dinâmicas;
- *Persistency*: para possibilitar que a execução do agente móvel possa ser suspensa temporariamente, armazenando-o em um meio persistente. Este serviço permite que os agentes evitem desperdiçar recursos de sistema enquanto esperarem eventos externos tais como o re-conexão de um usuário ou de um terminal.

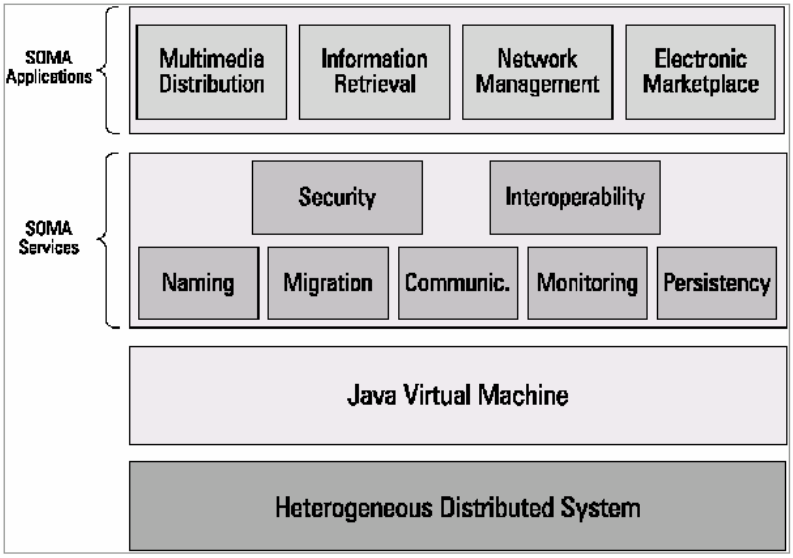

Figura 61 - A arquitetura do framework SOMA

Os dois componentes diferenciais de SOMA são o *Security* e o *Interoperability*. O *Security* provê um conjunto de serviços que possibilita a especificação e o reforço das políticas de segurança e mecanismos de autenticação de mensagens. Recursos de interoperabilidade via CORBA são providos pelo componente *Interoperability*. Não existe referência sobra a integração de SOMA com serviços Web.

5.5
Comparação entre as Abordagens

O propósito principal desta seção é mostrar as diferenças entre as várias abordagens para *frameworks* de *middleware* descritas neste Capítulo, e explicitar as vantagens da utilização da abordagem MIDAS em relação a outras abordagens. Para obter uma visão de alto nível das características dos *frameworks* analisados, utilizamos um conjunto de conceitos e propriedades, seguindo o mesmo critério adotado no Capítulo 4. A Tabela 5 mostra os *frameworks* relacionados frente aos conceitos e propriedades consideradas.

Características	Framework				
	MIDAS	JADE	CASA	AgentScape	SOMA
Infraestrutura					
MOM (Message-Oriented Model)	■	■	■	■	■
SOM (Service-Oriented Model)	■				
ROM (Resource-Oriented Model)	■	■	■	■	■
MGT (Management Model)	■	■	■	■	■
Configuração dinâmica	■				
Criação dinâmica de instâncias	■	■	■	■	■
Mobilidade		■		■	■
Suporte ao Projeto Detalhado					
Reuso de serviços distribuídos	■	■	■	■	
Wrapper de requisição	■	■	■	■	
Chamadas assíncronas	■	■	■	■	■
Componente *wrapper*	■				
Blackboard	■		■		
Acesso a dados	■		■		
Interoperabilidade					
Serviços Web	■	■		■	
FIPA		■			■
CORBA-RMI		■	■		■

Tabela 5 - Comparação entre características dos *frameworks*

A separação de conceitos é útil para identificar a existência ou não de uma determinada propriedade da arquitetura, ela não mostra as diferenças entre as implementações. Examinando a tabela por alto, pode ser visto que o MIDAS apresenta duas características importantes não encontradas nas outras abordagens: as abstrações providas pelo modelo orientado a serviços e a capacidade de configuração dinâmica dos agentes. Pode também ser observado que MIDAS provê um suporte maior ao projeto detalhado dos agentes, enquanto JADE provê maior capacidade de interoperabilidade.

A não existência de um modelo orientado a serviços pode exercer forte impacto sobre algumas propriedades da arquitetura, especialmente sobre as

propriedades de flexibilidade e adaptabilidade. Isto dificulta a implementação de mecanismos para configuração dinâmica[1] e torna mais difícil a evolução da arquitetura. A função do SOM é efetuar o processamento das mensagens e prover uma descrição e uma representação das interfaces em XML.

O foco da comparação está centralizado no *framework* JADE, por dois motivos principais. Primeiro, porque JADE utiliza os mesmos princípios básicos de MIDAS: é um *framework* de *middleware* mais conhecido e utilizado para o desenvolvimento de SMAs na Internet, e remete para o mesmo público alvo de MIDAS. Segundo porque as duas abordagens utilizam a plataforma Java e trabalham em conformidade com padrões estabelecidos. Os padrões FIPA usado por JADE e WSA adotado por MIDAS fornecem os fundamentos básicos para comparar as abordagens. Os demais *frameworks* CASA, AgentScape e SOMA possuem muitas características similares a JADE e MIDAS, e os principais fundamentos utilizados por estas abordagens são atendidos por JADE.

5.6 Discussão

Esta seção tem por propósito mostrar os possíveis ganhos e vantagens da abordagem proposta em relação às abordagens analisadas. O foco da comparação está centrado no *framework* JADE. A comparação foi feita utilizando uma implementação desenvolvida em JADE do sistema Expert Committee. Considerando as particularidades e circunstâncias das duas implementações, comparações globais foram evitadas. Ao invés disto, procuramos focalizar a análise em aspectos, mostrando alguns cenários e comentando como eles foram implementados.

Três cenários foram considerados para fazer a comparação. Os dois primeiros mostram como ocorre a troca de mensagens entre os agentes. O terceiro cenário mostra como a camada de lógica (onde se localizam os agentes) solicita um serviço para a camada de apresentação.

Cenário 1: ***como o Chair despacha uma mensagem para o Coordinator, pedindo novos revisores durante o processo de atribuição secundária.***

[1] JADE não menciona a existência de capacidade de configuração dinâmica. Conforme definido no Capítulo

Neste cenário, o processo de submissão de artigos ao workshop já foi concluído e a fase de revisão de artigos iniciada. Nesta fase, o processo de seleção de revisores já foi realizado, e os artigos já foram atribuídos aos revisores. Será mostrado inicialmente como este cenário foi implementado em JADE. Na implementação analisada, este cenário ocorre através de dois passos.

No primeiro passo o agente Chair deve recuperar a descrição da interface do agente Coordinator (AID - Agent Interface Description). Este procedimento é implementado no método *buscarCoordenador*, como mostra o fragmento de código da Figura 62.

```
74 private AID buscarCoordenador(int codigoEvento)
75 {
76     AID coordenador = null;
77
78     DFAgentDescription template = new DFAgentDescription();
79     ServiceDescription sd = new ServiceDescription();
80     sd.setType("coordenador");
81     sd.setName(Integer.toString(codigoEvento));
82     template.addServices(sd);
83     try
84     {
85         DFAgentDescription[] result = DFService.search(this, template);
86         if (result.length > 0)
87             coordenador = result[0].getName();
88     }
89     catch (FIPAException fe)
90     {
91         fe.printStackTrace();
92     }
93
94     return coordenador;
95 }
```

Figura 62 - O agente Chair cria uma instancia da interface de descrição

Nas linhas 78 e 79, o agente Chair cria uma instância (*template*) da descrição da interface de FIPA DFAgentDescription, e o nome do destinatário é adicionado ao *template*. Entre as linhas 83 e 94, o *template* é submetido ao agente DF de FIPA que através do método *search* (linha 85) procura pela especificação, retornando na linha 94 uma interface do tipo AID. De posse desta interface, o agente Chair pode então fazer a comunicação no passo seguinte.

No segundo passo, de posse da interface, o agente Chair cria uma mensagem, e a despacha utilizando o método *send* da interface AID. Este procedimento é mostrado no fragmento de código apresentado na Figura 63.

3, a configuração dinâmica possibilita substituir componentes na JVM em tempo e execução.

```
365    coordenador = buscarCoordenador(artigo.getEvento().getCodigo());
366
367    ACLMessage msgNovaLista = new ACLMessage(ACLMessage.REQUEST);
368    msgNovaLista.addReceiver(coordenador);
369    msgNovaLista.setContent(artigo.getTitulo());
370    msgNovaLista.setConversationId(ParametroMensagem.ATRIBUICAO_SECUN
371    msgNovaLista.addUserDefinedParameter(ParametroMensagem.NUMERO_REV
372    msgNovaLista.setReplyWith("rqt" + System.currentTimeMillis()); //
373
374    // Preparando o modelo de resposta
375    mtAtribuicaoSecundaria = MessageTemplate.and(MessageTemplate.Matc
376                                                 MessageTemplate.Matc
377    step++;
378
379    myAgent.send(msgNovaLista);
```

Figura 63 - Chair adiciona parâmetros à interface e despacha a requisição

No MIDAS, este procedimento pode ser feito em um único passo. O agente Chair utiliza o Blackboard para enviar a mensagem ao agente Coordinator, simulando uma comunicação ponto-a-ponto. A Figura 64 ilustra este procedimento.

```
82          int     priority = 1; // Very High
83          String target = "coordinator";
84          String message = "selectReviewers";
85
86          Board.writeOnBoard(priority,target,message,this);
```

g

Figura 64 - Chair enviando mensagem ao Coordinator

Entre as linhas 82 e 84 são efetuadas atribuições de valores aos atributos *priority*, *target* e *message*. O atributo *priority* torna possível definir um grau de prioridade à mensagem. O atributo *target* indica o agente destino e o atributo *message* identifica o nome do serviço. A seguir, na linha 86, o agente escreve a mensagem utilizando o método estático *Board.writeOnBoard* do Blackboard. Neste procedimento, a abordagem MIDAS prioriza a flexibilidade em oposto à *tipificação* utilizada para troca de mensagens entre os agentes em JADE.

Esforço de implementação deste Cenário

No JADE, a operação envolve três classes: o agente, o objeto de interface do agente alvo e o objeto de mensagem, todos instanciados manualmente e estáticos. No MIDAS, a operação envolve apenas a colaboração entre os agentes Chair e Blackboard, construída através de um método e atributos simples.

Custo de evolução deste Cenário

Adicionar mais um ouvinte ou criar uma nova comunicação em JADE envolveria o desenvolvimento de duas a cinco classes novas, além da alteração no código da classe que despacha a mensagem. No MIDAS, o custo para adicionar mais um ouvinte custa apenas uma chamada adicional ao Blackboard.

Cenário 2: *como o agente Coordinator recebe a mensagem*

Este cenário mostra a continuação do processo de troca de mensagens entre os agentes, ilustrando o lado do destinatário, isto é, como o agente Coordinator recebe a mensagem. O agente Coordinator estende a classe abstrata CyclicBehaviour, que é uma classe de comportamento cíclico que JADE possui em sua estrutura de classes comportamentais. O fragmento de código apresentado na Figura 65 ilustra este procedimento. A classe ReceberPedidoSecundário é implementada como uma intra-classe do agente Coordinator, e procura continuamente capturar a mensagem específica de solicitação de revisores, descobrindo se uma mensagem chegou quando o objeto *msg* é não nulo.

```
68  private class ReceberPedidoSecundarioRevisores extends CyclicBehaviour{
69
70      public void action() {
71          MessageTemplate mt = MessageTemplate.and(
72                                  MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
73                                  MessageTemplate.MatchConversationId(ParametroMensagem.ATRIBUICAO_SECUNDARIA));
74
75          ACLMessage msg = myAgent.receive(mt);
76
77          if (msg != null)
78          {
79              // Select Reviewers
```

Figura 65 - Intra-classe ReceberPedidoSecundario do agente Coordinator

No segundo passo, o método construtor do agente deve adicionar a classe ReceberPedidoSecundarioRevisores ao seu leque de comportamentos, como mostra o fragmento de código na Figura 66.

```
28  public class Coordenador extends Agent
29  {
30      private Evento evento;
31
32      protected void setup()
33      {
34          this.addBehaviour(new ReceberPedidoSecundarioRevisores());
```

Figura 66 - O Coordinator adiciona um novo comportamento ao seu leque

No MIDAS, esta atividade também ocorre em dois passos. No primeiro passo, o Coordinator deve implementar o método *boardChanged* da interface MessageListener. Ao implementar esta interface do Blackboard, ele se habilita como ouvinte. Este método é invocado apenas quando a mensagem é despachada, o Coordinator identifica a mensagem testando seu identificador (no caso, a String *selectReviewers*). A Figura 67 mostra este procedimento.

```
36    public void boardChanged(Message msg)
37    {
38        if (msg.getData().equals("selectReviewers"))
39        {
40            // Logging
41            System.out.println("[Coordinator] - Selecting Reviewers");
42
43            // Método que opera a seleção dos Revisores
44            selectReviewers();
45        }
46    }
```

Figura 67 - O Coordinator no MIDAS implementa o método boardChanged

No segundo passo, o agente Coordinator se registra no Blackboard durante a inicialização do seu ciclo de vida. A Figura 68 mostra este procedimento.

```
23    @Override
24    public void lifeCycle() throws LifeCycleException
25    {
26        // Logging
27        System.out.println("[Coordinator] - Registering on Board");
28
29        // Registrando no Board
30        Board.addMessageListener("coordinator",this);
31    }
```

Figura 68 - O agente Coordinator se registra como ouvinte do Blackboard

Esforço de implementação deste Cenário

O processo de escutar mensagens no JADE é simulado por uma contínua verificação sobre a ocorrência da mensagem. Quanto mais agentes existirem, e mais possíveis mensagens estes agentes receberem, mais *threads* estarão ativos durante o ciclo de vida do sistema, realizando sempre as mesmas verificações. É um custo de processamento muito alto para mensagens que podem ser muito escassas ou até mesmo únicas durante todo o ciclo de vida. No MIDAS, há uma interface única para o recebimento de mensagens, que trabalha junto ao Board. Assim que uma mensagem é despachada, o(s) seu(s) alvo(s) têm o método desta

interface invocada para comunicar ao agente a mensagem, sem necessidade de *lookups* contínuos.

Custo de evolução deste Cenário

No JADE há a necessidade de se instanciar (e programar) uma classe de comportamento (CyclicBehaviour) para cada possível mensagem que o agente possa receber. No MIDAS, para que o agente possa receber uma nova mensagem, basta inserir uma nova cláusula if para capturar o título da mensagem no método *boardChanged* (Figura 67).

Cenário 3: *como a camada de apresentação pode requisitar um serviço à camada lógica*

O terceiro cenário analisa como os dois *frameworks* se comportam no processo de interação entre as camadas de lógica e de apresentação. Desempenhando o papel de *frameworks* de *middleware*, as duas abordagens utilizam mecanismos diferentes para a comunicação entre as camadas. No JADE, as atividades envolvem dois passos: no primeiro passo, a aplicação implementa um agente especial que simula um *proxy* ou um mediador de serviços. Este agente possui um comportamento cíclico que procura continuamente capturar requisições de serviços de GUI, testando seus nomes para encaminha-las aos agentes que podem atendê-las. A Figura 69 ilustra este procedimento.

```
144    private class ReceberRequest extends CyclicBehaviour
145    {
146        public void action()
147        {
148            Interaction interaction = (Interaction)getO2AObject();
149            if(interaction != null)
150            {
151                // Verificando qual foi o request para dar o tratamento correto
152                if (interaction.getRequest().getParameter("comando").equalsIgnoreCase("submeterArtigo"))
153                    myAgent.addBehaviour(new TratarSubmeterArtigo(myAgent, interaction));
154
155                if (interaction.getRequest().getParameter("comando").equalsIgnoreCase("registrarRevisao"))
156                    myAgent.addBehaviour(new TratarRegistrarRevisao(myAgent, interaction));
157
158                if (interaction.getRequest().getParameter("comando").equalsIgnoreCase("registrarVoto"))
159                    myAgent.addBehaviour(new TratarReceberVoto(myAgent, interaction));
160
161                if (interaction.getRequest().getParameter("comando").equalsIgnoreCase("enviarCameraReady"))
162                    myAgent.addBehaviour(new TratarEnviarCameraReady(myAgent, interaction));
163
164            }
165            else
166                block();
167        }
168    }
```

Figura 69 - Classe da implementação JADE simulando o papel de um proxy

Na linha 148, o método *action* utiliza o objeto de requisição *interation* do tipo Interaction e verifica se existe uma requisição através do método *getO2AObject* do JADE. Ele testa o seu conteúdo na linha 149 e se não for nulo, ele testa o conteúdo da requisição e adiciona um novo comportamento ao agente.

No segundo passo, o *servlet* deve instanciar direta e manualmente o próprio agente controlador. Este procedimento está ilustrado no fragmento de código da Figura 70. O fragmento mostra o *servlet* SubmeterArtigo que invoca o método *iU.putO2AObject* do JADE para passar à *action* os objetos de requisição/resposta do comportamento do agente encapsulados no objeto Interaction. O serviço em si deve estar implementado em um dos comportamentos específicos que o agente controlador é capaz de redirecionar.

```
17 public class SubmeterArtigo extends Comando
18 {
19     public void executar(HttpServletRequest request, HttpServletResponse response)
20     {
21         // Procurando o agente de interface
22         AgentController iU = Repositorio.getInstancia().getInterfaceUsuario();
23 
24         // Create an interaction object wrapping the HTTP request and response
25         Interaction interaction = new Interaction(request, response);
26 
27         // Pass the interaction to the Agent and wait for the signal that the interaction was updated
28         try
29         {
30             iU.putO2AObject(interaction, AgentController.ASYNC);
31             interaction.waitChangedResponse();
32         }
33         catch(Exception e)
34         {
35             e.printStackTrace();
36         }
37     }
38 }
```

Figura 70 - Servlet do agente controlador de acesso a GUI's em JADE

No MIDAS, este procedimento também é executado em dois passos. No primeiro passo (Figura 71) o agente ou componente provedor do serviço deve estar registrado no arquivo descritor *strcture.xml*.

```
21     <organizations>
22         <organization>
23             <name>ExpertCommittee</name>
24             <package>ecommittee</package>
25 
26             <agents>
27                 <agent>
28                     <name>Event</name>
29                     <package>business.agents.event</package>
30                     <className>Event</className>
31                     <protocol>native</protocol>
32                 </agent>
```

Figura 71 - Descrição de um agente MIDAS na especificação structure.xml

Este procedimento é executado de forma automática pelo agente C-Manager, que captura as especificações passadas através do seu assistente GUI. Conforme apresentado no Capítulo 3, são mantidas duas descrições XML na plataforma: uma em formato de estrutura e a outra é uma descrição do serviço, que provê um mecanismo mais rápido e eficiente para localizar serviços. A Figura 72 mostra a especificação da descrição *serviços.xml*.

```
47    <service>
48        <name>buildEvent</name>
49        <organization>ExpertCommittee</organization>
50
51        <entity>Event</entity>
52
53        <scope>local</scope>
54
55        <parameters>
56            <parameter>
57                <name>event</name>
58                <class>ecommittee.business.valueobjects.EventBean</class>
59            </parameter>
60        </parameters>
61
62        <description>Builds an Event</description>
63    </service>
```

Figura 72 - Descrição de um serviço MIDAS na especificação services.xml

No segundo passo, o *servlet* requisita ao container local um *wrapper* de serviço, através do qual pode atribuir parâmetros e disparar a execução de forma síncrona ou assíncrona. O *proxying* é feito pela arquitetura; o *servlet* precisa apenas saber o nome do serviço, promovendo baixíssimo acoplamento entre as camadas. A Figura 73 ilustra este procedimento.

```
112         ServiceWrapper service;
113
114         service = AgentServer.require("ExpertCommittee","buildEvent");
115         service.addParameter("event",event);
116         service.run();
```

Figura 73 - Requisição de um serviço no MIDAS

Enquanto no JADE os agentes têm que manipular diretamente os objetos de requisição RMI, no MIDAS há uma abstração da requisição, uniformizando a manipulação dos parâmetros que são passados junto a uma requisição HTTP. Desta vez é o MIDAS que prefere a tipificação à flexibilidade.

Esforço de implementação deste Cenário

Como no JADE não há suporte para um modelo de serviços, um agente especial precisa ser programado para atuar como intermediário entre as camadas, desempenhando o papel de Proxy. Ainda, como todos os re-direcionamentos são efetuados manualmente, há grande chance de ocorrência de erros. Além disso, o *servlet* precisa instanciar programaticamente o agente re-direcionador. No MIDAS há um agente intermediário que desempenha o papel de Proxy. Este agente elimina as complexidades de re-direcionamento e *lookup* dos provedores dos serviços. Basta à camada de visão saber o nome do serviço que se deseja, e a arquitetura cuida do resto.

Custo de evolução deste Cenário

Requisitar novos serviços no JADE envolve a manipulação do agente re-direcionador, que com o passar do tempo, pode se tornar extremamente complexo e propenso a erros, dependendo do tamanho do sistema. Necessita da criação de comportamentos excessivos, para cada serviço, de cada agente, tornando o processo extremamente trabalhoso ao desenvolvedor. No MIDAS, é possível inserir novos serviços apenas registrando o agente nos arquivos descritores, sem necessidade de re-compilação do código da arquitetura. Os agentes são invocados através de uma interface única, ao invés de terem que implementar classes diferentes para cada serviço, suavizando o custo de crescimento do número de serviços.

O paradigma SOA utilizado por MIDAS pode trazer ganhos quando comparado ao modelo essencialmente orientado a mensagens utilizado por JADE. Conforme ficou demonstrado, a não existência de uma camada de abstração para tratar os serviços traz complexidade adicional, além de maior esforço e tempo de desenvolvimento. As vantagens de utilizar SOA foram demonstradas nos três cenários apresentados. Nos dois primeiros, o foco foi o modelo de comunicação entre agentes. A introdução do Blackboard trouxe considerável economia de código nos procedimentos de troca de mensagens entre agentes. e redução de complexidade e menor custo de evolução da aplicação. Isto ficou comprovado pelos fragmentos de códigos apresentados nas figuras 62, 63 e 64. O Blackboard atua ainda como um sensor, capturando as mudanças nas variáveis ambientais e notificando os agentes. Desta forma, os agentes não precisam ficar efetuando *loops* para recuperar mensagens, economizando processamento.

Pela análise apresentada no terceiro cenário, ficaram evidentes as vantagens providas pelo agente Proxy na comunicação entre as camadas do *middleware*. O Proxy promove transparência de local, recuperando o endereço do provedor da descrição XML e melhora a eficiência, como pode ser visto no cenário 3, onde a implementação JADE é "forçada" a simular o comportamento de um Proxy. Um agente especial precisa ser programado para atuar como intermediário entre as camadas, desempenhando o papel de Proxy.