6
Conclusão

Agentes e SMA's que atuam na Internet enfatizam uma nova geração de aplicações, baseada em tecnologias emergentes e num conjunto de padrões abertos para a Web. A necessidade de fácil adaptação às novas tecnologias e à dinâmica de mudanças em requisitos demanda novos conceitos e incorporação de arquiteturas abertas, flexíveis e adaptativas. SMA's que atuam na Internet estão se tornando mais complexos e passam hoje pela transição de arquiteturas monolíticas, baseadas em componentes de software passivos para arquiteturas flexíveis e abertas [99, 52], compostas por agentes dinâmicos e pró-ativos.

Grande parte dos atuais *frameworks* e plataformas para SMA's utilizam modelos de comunicação unilateralmente associados ao conceito de mensagem, e se fundamentam em uma infra-estrutura de comunicação síncrona e fortemente acoplada. Estas características podem exercer forte impacto nas propriedades de flexibilidade e adaptabilidade da arquitetura e aumentar a complexidade do desenvolvimento, conforme demonstrado na Seção 5.5.

Os mecanismos utilizados pela abordagem atuam em dois níveis de abstração: *ao nível da arquitetura*, abstraindo a complexidade de serviços de infraestrutura e *ao nível do projeto detalhado* dos agentes, provendo meios para facilitar a comunicação e troca de mensagens, promover o reuso de serviços distribuídos e dar suporte ao *workflow* dos agentes.

6.1
Contribuições da Tese

Acreditamos que o paradigma SOA pode facilitar o desenvolvimento de SMA's na Internet porque provê um estilo arquitetural flexível, adaptável e habilitado a evoluir. SOA exige um nível de coordenação muito mais simples do que as abordagens tradicionais focalizadas no modelo orientado a mensagens. Este trabalho de pesquisa procurou explorar as vantagens advindas da utilização do paradigma SOA para o desenvolvimento de SMA's, mostrando um *framework*

cuja uma arquitetura segue e estende o modelo de referência WSA. No decorrer do trabalho, e mais especificamente na Seção 5.5.4, pudemos demonstrar as vantagens provenientes da utilização da solução proposta para a redução da complexidade e do tempo de desenvolvimento.

Os conceitos e abstrações fornecidos por WSA facilitaram o entendimento da arquitetura e a obtenção de blocos de construção modulares, compreensíveis e coesos, construídos a partir de um modelo de papéis reutilizável. Agentes da arquitetura e das aplicações podem alterar dinamicamente o seu comportamento, tornando simples o processo de evolução e adaptação a novos requisitos.

As principais contribuições desta tese são as seguintes:

1. Construção de um *framework* com arquitetura centrada em agentes para os modelos de referência WSA, composta por duas estruturas principais: uma concreta, representada pelos *agentes intermediários* e uma abstrata, representada pelos *agentes de aplicações*.

A separação dos papéis dos agentes intermediários e dos agentes de aplicações possibilitou obter uma distinção clara das partes fixas e não-fixas do *framework*. Os *agentes intermediários*, que provêem serviços de infraestrutura compõem a parte fixa do *framework* e permanecem inalterados para as aplicações. Os *agentes de aplicações* compõem uma estrutura abstrata, cuja implementação representa os agentes provedores ou requisitantes de serviços localizados em aplicações. A definição das estruturas (Seção 3.4.1, Fig. 26 e 3.4.2, Fig. 36) tornou possível obter uma melhor visão da participação e do posicionamento dos agentes na arquitetura, das suas responsabilidades e das interações e colaborações necessárias para atingir os objetivos definidos pela solução.

2. *Extensão dos conceitos da arquitetura de referência WSA*: foram introduzidos o papel de agente abstrato e um *blackboard* para comunicação entre agentes.

A introdução conceito de agente abstrato (Seção 3.4.1.5, Fig. 35) estende a especificação da arquitetura de referência WSA, oferecendo um meio para fatorar em uma superclasse as propriedades comuns a todos os agentes. A classe abstrata regula o fluxo de interação entre agentes e a arquitetura, facilitando a coordenação

e a comunicação entre a camada lógica e as camadas de apresentação e de dados. A classe abstrata define os *hot-spots* para a implementação das particularidades específicas de aplicações (Seção 4.1.4.1, Fig. 42) e um conjunto de métodos para controlar o ciclo de vida dos agentes e regular o fluxo de interação entre agentes de aplicação e a arquitetura.

A introdução do *blackboard* (Seção 3.2.1.5 e 4.1.4.1, Fig. 43/44) supre uma deficiência presente em arquiteturas que utilizam o paradigma SOA: a falta de suporte a um modelo de comunicação ponto-a-ponto. O *blackboard* oferece um meio para simular comunicação ponto-a-ponto entre agentes, e possibilita a disseminação de mensagens através do mecanismo de *broadcast*. Ele pode funcionar também como um sensor para os agentes, capturando as mudanças ambientais e notificando os agentes. O Blackboard supre uma deficiência encontrada em arquiteturas orientadas a serviços. O paradigma SOA provê transparência de serviços, o que significa que o processo de requisitar e prover serviços acontece sem uma referência explícita do agente provedor. SOA não atende a situações onde há necessidade desta identificação, por exemplo, quando um agente deseja enviar uma mensagem para outro agente específico.

As duas primeiras contribuições estão relacionadas ao Objetivo 3, e foram demonstradas conceitualmente no Capítulo 3, e no protótipo Expert Committee, apresentado no Capítulo 4, cujas seções foram identificadas na descrição. Algumas propriedades ou combinação de propriedades e conceitos utilizados pela abordagem podem ser vistas como inovações tecnológicas, descritas a seguir.

3. *Inovações Tecnológicas*: caracterizam a combinação de um conjunto de técnicas, ferramentas e padrões, utilizados para garantir o cumprimento dos objetivos propostos. Dentre estas, podem ser citadas:

   3.1. *Redução da Complexidade e do Tempo de Desenvolvimento*: está diretamente relacionada aos objetivos 1 e 2 (Seção 1.3.2). Esta redução foi conseqüência direta dos seguintes mecanismos e técnicas utilizadas:

   - *Simplicidade no processo de criar, enviar e receber uma requisição de serviços*: o paradigma SOA exige nível de coordenação muito mais simples para a manipulação de requisições de serviços do que as abordagens tradicionais associadas ao conceito de mensagens (Seção

5.5.4). No JADE, por exemplo, para cada serviço de cada agente é necessário criar uma intra-classe de comportamento específico e descrever programaticamente a interface. No MIDAS, os serviços descritos em formato XML simplificam o processo de requisição. Os agentes são invocados por uma interface única, ao invés de terem que implementar classes diferentes para cada serviço.

- *Transparência total para manipular requisições remotas de serviços*: os serviços de transparência de local providos pelo *framework* simplificaram o desenvolvimento, abstraindo a complexidade de efetuar requisições locais e remotas de serviços (Seção 4.1.4.1). Para requisitar um serviço, não é necessário especificar o endereço e o recipiente provedor do serviço: apenas o nome do serviço e a organização onde ele está alocado são necessários.
- *Facilidade para a coordenação entre as camadas de lógica e de apresentação*: a utilização de uma abstração para representar o modelo orientado a serviços simplificou a coordenação entre os agentes e as camadas de apresentação (Seção 5.5.4) e de dados. O agente Proxy (Seção 3.2.1.2) efetua o processamento das mensagens eliminando a complexidade de re-direcionamentos e criação de classe de controle.
- *Suporte ao projeto detalhado dos agentes*: todas as características anteriores facilitam o projeto detalhado dos agentes. Entretanto, algumas outras propriedades descritas a seguir fornecem suporte adicional:
  - *Reuso de serviços distribuídos*: componentes e processos de software serem expostos como serviços de forma que eles possam ser facilmente acessíveis, promovendo fácil reuso (Seção 4.2). Os mecanismos para a visualização dos serviços providos pelo agente Manager auxiliam a procura e descoberta de serviços (Seção 4.1.5).
  - *Blackboard*: além do suporte ao modelo de comunicação, o *blackboard* facilita o *workflow* dos agentes mantendo uma área compartilhada de variáveis globais e simulando um sensor. Os agentes são notificados quando ocorrem mudanças nas variáveis ambientais (Seção 3.2.1.5 e 4.1.4.1).

- *Componentes wrapper*: facilitam o encapsulamento de objetos de acesso a dados (Seção 4.1.4.2) e funcionalidades de sistemas legados (Seção 4.2). Facilitam também a integração entre a plataforma e serviços Web, podendo ser usado para representar um serviço Web externo (Seção 4.2, Fig. 52).
- *Acesso a dados*: as facilidades providas pelo agente DBPool (Seção 3.2.1.6, Fig. 20, Seção 3.4.1.5, Fig. 35) para manipular dados em bases relacionais e integrar *frameworks* especialistas em mapeamento objeto-relacional (Seção 4.1.4.2, Fig. 49).

3.2. *Flexibilidade e Capacidade de Evoluir*: estão relacionadas com o objetivo 1, e podem ser sintetizadas como segue:

- *Flexibilidade*: a flexibilidade é uma propriedade implícita em abordagens que utilizam o conceito de SOA. Ela resulta principalmente da natureza fracamente acoplada do paradigma (Seção 1.1.2 e 1.3.2.1) que utiliza as facilidades providas por XML para descrever as interfaces de serviços.
- *Capacidade de evoluir*: a capacidade de evoluir pode ser analisada a partir de duas perspectivas:
  - *Da arquitetura*: blocos de construção modulares e coesos desempenham responsabilidades definidas através de um modelo de papéis. A possibilidade de reutilizar o modelo de papéis (Seção 3.2) pode contribuir para a capacidade de evolução da arquitetura. Além disso, a capacidade de configuração dinâmica da arquitetura (Seção 3.2.1.2 e 3.4.1.2) garante esta propriedade, já que a arquitetura consegue evoluir mudando o seu comportamento em tempo de execução.
  - *Dos SMA's*: a utilização de interfaces abstratas (Seção 3.4.1.5) exerce forte impacto sobre a capacidade de evolução dos SMA's gerados pelo *framework*. As interfaces possibilitam criar famílias de agentes e componentes, fornecendo um pré-requisito fundamental para a criação dinâmica de instâncias. O processo de criação das instâncias é abstraído pelo agente

Proxy (Seção 3.4.1.2), que utiliza o padrão de projeto Factory Method [48] para abstrair o processo de criação de instâncias.

3.3. *Interoperabilidade e Integração com Serviços Web*: focaliza a capacidade da plataforma de atuar integrada, e está relacionada com o objetivo 3.

- *Interoperabilidade bidirecional*: o componente Adapter, localizado no agente S-Broker oferece comunicação bidirecional entre os agentes MIDAS e serviços Web (Seção 4.2 e 4.3).
- *Geração automática das especificações WSDL e UDDI*: efetuada pela classe Generator (Seção 3.4.2), localizada no agente S-Broker. Ela utiliza a ferramenta Axis para gerá-las de forma automática.

A fácil integração entre os agentes MIDAS e serviços Web garante também a fácil interoperabilidade da plataforma com aplicações externas e sistemas legados. Esta capacidade também é ampliada com a presença do Component Container (Seção 3.4.3), que provê um ambiente de execução para a integração entre SMA's e aplicações externas. Os componentes podem ser utilizados para encapsular serviços Web (Seção 4.2) e podem colaborar com os agentes (Seção 4.1.4.1) para compor parte da resolução do problema.

6.2 Restrições

A adoção do paradigma SOA e as propriedades de flexibilidade e capacidade de evoluir providas pelo *framework* MIDAS resultam em algumas restrições. As restrições, entretanto, não afetam o bom funcionamento da plataforma e podem ser parcialmente resolvidas com a adoção de algumas técnicas, como descrito a seguir.

1. *Dependência do servidor*: o ponto fraco de qualquer sistema centralizado é a dependência do servidor central. Existem algumas maneiras de diminuir o impacto de eventuais falhas de comunicação com servidor. Os seguintes procedimentos foram adotados para diminuir o grau de dependência:

- *Containers trabalhando stand-alone*: um container na plataforma pode operar na modalidade *stand-alone*, e os efeitos de uma falha de comunicação remota podem ser minimizados trabalhando no nível de implementação. Por exemplo, todas as requisições remotas de serviços podem ser efetuadas de maneira assíncrona. Se o serviço não puder ser atendido no momento da solicitação, ele continua na fila de espera de serviços não executados, e novas tentativas podem então ser feitas posteriormente para capturar a requisição não atendida da fila.
- *Detecção de falhas*: na plataforma MIDAS, as falhas de comunicação com o servidor central são detectadas pelo agente *Broker*. Antes de enviar uma requisição de serviço ao servidor, ele verifica se o serviço está disponível. Se não estiver, ele retorna um aviso ao agente *Manager* que devolve uma mensagem de indisponibilidade ao agente requisitante do serviço.
- *Servidor espelho*: outra forma de contornar o problema de falha no servidor central é utilizar servidores espelho. Este procedimento envolve implicações que fogem ao escopo deste trabalho.

2. *Eficiência*: uma possível vantagem do modelo de comunicação utilizado por JADE em relação ao modelo centralizado usado por MIDAS é o tempo de resposta quando existe uma grande quantidade de requisições simultâneas. A comunicação direta entre containers realizada através de RMI em JADE tende a ser mais rápida do que aquela realizada de forma indireta. Entretanto, testes realizados para medir a eficiência não demonstram perda acentuada de performance no modelo centralizado para um número considerável de requisições. A eficiência no transporte HTTP é garantida pelo servidor Tomcat, e a perda resultante da comunicação indireta é compensada em parte pela boa eficiência da comunicação com soquetes.

3. *Comunicação remota ponto-a-ponto*: no paradigma SOA, conforme comentado anteriormente, não existe comunicação explícita entre entidades. A comunicação ocorre de forma anônima, através de requisições de serviços. A não existência de comunicação ponto-a-ponto no paradigma SOA é suprida pelo *blackboard*. Entretanto, este recurso não está disponível para

comunicação remota. Para obter uma comunicação remota ponto-a-ponto, o agente (ou desenvolvedor) precisa identificar o nome do container provedor e utilizar o *blackboard* remoto. Esta capacidade pode ser alcançada utilizando a mobilidade, discutida em trabalhos futuros.

6.3 Trabalhos Futuros

Os trabalhos futuros focalizam duas propriedades da arquitetura que estão em fase de projeto e implementação: mobilidade e segurança. Existe uma gama de aplicações para a Internet para as quais a mobilidade é um fator fundamental, como por exemplo, comércio eletrônico, telecomunicações [White1996], dentre outras. Segurança é outro aspecto fundamental para aplicações que atuam na Web. Nesta seção, são descritos os fundamentos que estão sendo projetados para implementar estas propriedades.

6.3.1 Mobilidade

Os *frameworks* JADE [9], *AgentScape* [89] e SOMA [23], descritos no Capítulo 5, provêem mobilidade. No MIDAS, a mobilidade está sendo implementada com a adição de um novo agente intermediário e um novo tipo de requisição. Este novo agente intermediário - denominado *Agentport* - atua em colaboração com o agente *Broker*, e pode ser capaz de enviar um agente a outros containers, encapsulando-os em requisições HTTP de um lado, e de recebê-los no outro. A nova requisição – denominada requisição de transporte é implementada pelo agente *Broker*.

A requisição trabalha como uma cápsula de transporte para o agente, permitindo seu embarque no container de origem, e desembarque no container de destino. Uma vez desembarcado em um container, o agente visitante pode manter interação com os banco de dados e *blackboard* locais. Tendo acesso ao *blackboard* de um container, o agente visitante poderá enviar suas mensagens ao *blackboard*, assim como recuperar mensagens de outros agentes. Estas capacidades exigem um tratamento mais robusto das requisições recebidas pelo

agente *Broker* e dispositivos de segurança e autorização de acesso, comentado a seguir.

6.3.2 Segurança

Segurança é um dos requisitos da arquitetura de referência WSA [13], definida pelo modelo de políticas. Embora ainda não tenham sido implementados dispositivos de segurança, esta tarefa já está em fase de execução. Um componente 'filtro' encarregado de autenticar as mensagens deverá atuar em colaboração com o agente *Proxy*. Desta forma, antes deste encaminhar a execução do serviço ao agente ou componente responsável, o componente deverá efetuar a autenticação da mensagem. Os dispositivos de segurança são descritos em um arquivo XML, que é interpretado dinamicamente. Além dos aspectos de segurança, regras podem ser usadas para evitar falhas, restringir o comportamento da arquitetura, etc.

As regras são descritas num arquivo XML especial, denominado *rules.xml*. Neste arquivo podem ser inseridas dinamicamente regras de filtragem, que são processadas para todas as requisições que chegam à arquitetura. Um filtro é uma sentença lógica composta de pelo menos dois *atributos* e um *operador*, que é ativado quando a sentença é verdadeira. A Figura 74 mostra a classe *Filter*, localizada no agente *Proxy*, que é responsável pela interpretação das regras descritas no arquivo XML.

Para esta classe, os seguintes métodos já foram definidos:

- *Object call*: método implementado da interface Callable que é disparado pelo agente *Proxy* e executa em uma *thread* paralela. Ele invoca continuamente o método *parseRules*.
- *parseRules*: método invocado pela *thread* de execução do método *call()*, que verifica a ocorrência de alterações no arquivo *rules.xml*. Caso positivo, incorpora as novas regras ao modelo em memória, para que possam ser utilizadas pelo filtro.
- *Boolean applyRules*: método invocado pelo agente *Proxy* no processo de validação de uma requisição de serviço. Ele verifica se existe em memória qualquer regra que impeça o atendimento do serviço num

dado momento e contexto. Retorna verdadeiro caso o serviço possa ser executado, e falso no caso contrário.

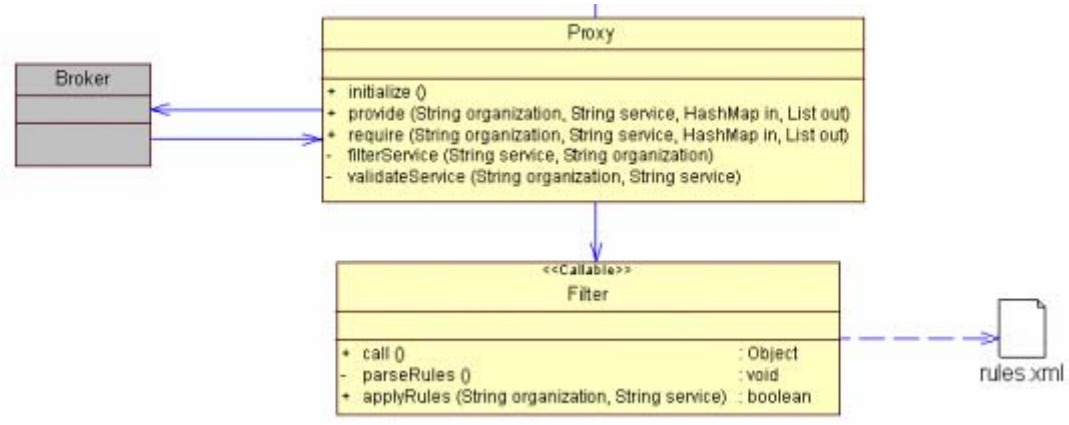

Figura 74 - O módulo de regras e segurança

Quando o foco é segurança, existem duas maneiras de operação para a filtragem:

- *allow all* – a forma de operação *default*, todas as requisições são naturalmente aceitas, com exceção daquelas que ativarem qualquer um dos filtros;
- *deny all* – todas as requisições são naturalmente rejeitadas, com exceção daquelas que ativarem qualquer um dos filtros.

O próximo passo é a escolha de uma linguagem para descrição das regras. Experiências anteriores [59, 60] indicam que as linguagens CCS [80] ou CSP [61] são adequadas para descrever formalmente as regras e restrições, pois suportam a representação de processos concorrentes.