3
A plataforma MIDAS

MIDAS é uma plataforma que provê um ambiente de execução e um *framework* para o desenvolvimento de SMA's. Sua arquitetura é orientada a serviços, e adere aos padrões definidos pela arquitetura de referência WSA. O ambiente de execução oferece facilidades para o gerenciamento e instanciação dos agentes, gerenciamento do ciclo de vida da plataforma, procedimentos de registro, configuração dinâmica, descoberta de serviços e outros. O *framework* provê um conjunto de serviços genéricos de infraestrutura que abstraem funcionalidades complexas e um modelo para facilitar o projeto detalhado e implementação dos agentes.

Os conceitos e modelos definidos por WSA fornecem uma base de referência sólida para a arquitetura. Entretanto, eles precisam ser estendidos para atender aos requisitos de desenvolvimento de aplicações baseadas em agentes. De acordo com Zhao [121], embora o conceito de agente esteja presente em todos os modelos, a atual versão de WSA não provê uma visão coerente dos agentes na arquitetura e os seus inter-relacionamentos.

Zhao propõe a inclusão de um modelo genérico denominado Agent Role Model (ARM) na atual arquitetura de referência WSA. O ARM encapsula o comportamento dinâmico e as propriedades dos agentes em papéis, fundamentadas por dois tipos de relacionamentos: (i) relacionamento entre os agentes e os seus papéis, implementado através de uma interface concreta de papéis e (ii) relacionamento entre papéis abstratos e concretos dos agentes, implementados através de uma especialização de uma interface genérica.

O processo para a definição dos componentes da arquitetura MIDAS possui alguma similaridade com a proposta de Zhao. Entretanto, existem diferenças fundamentais nas abordagens. Ao invés de um papel genérico, propomos diferentes representações para dois tipos de agentes que participam da arquitetura: agentes intermediários e agentes de aplicações. A diferenciação entre os modelos

de papéis dos agentes é um dos pontos fundamentais para obter um perfeito posicionamento dos agentes na arquitetura.

A Figura 10 mostra as fases para a definição dos componentes da arquitetura do MIDAS. Eles foram descritos através dos seus papéis e responsabilidades. Cada comportamento básico estabelecido pela arquitetura de referência foi separado e definido como uma responsabilidade específica de um papel, considerando o escopo do trabalho. O modelo de papéis forma a base para a etapa seguinte, a de definição das estruturas e do modelo de comunicação.

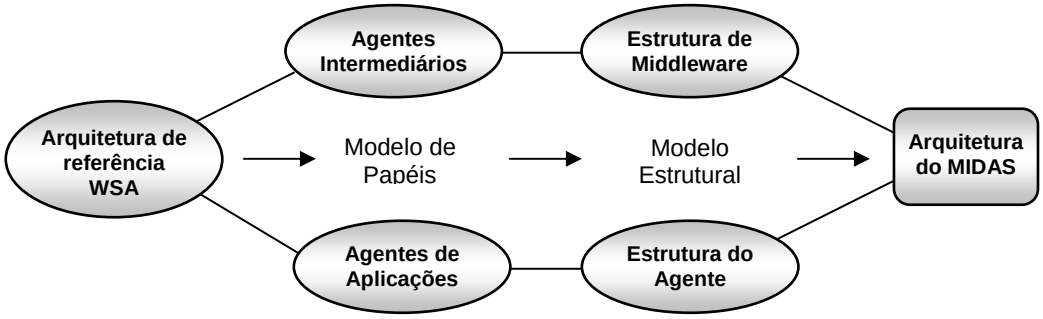

Figura 10 - Estratégia para a definição dos componentes da arquitetura

Na abordagem proposta, o papel de um agente intermediário pode ser visto como uma interface concreta, já que normalmente ele desempenha um conjunto de responsabilidades que pode ser definido a priori. O mesmo não acontece com os agentes de aplicações. Tipicamente, em uma aplicação, o papel concreto de um agente deve ser descrito através da especialização de uma classe abstrata.

A *estrutura de middleware* e a *estrutura de agentes* de aplicações são as duas estruturas básicas da arquitetura MIDAS. Elas foram definidas a partir dos modelos de papéis e das políticas para a infraestrutura de comunicação. A *estrutura de middleware* modela os agentes intermediários, que atuam de forma colaborativa e dinâmica. A *estrutura de agentes* é composta por uma classe abstrata que representa os agentes de aplicações. A classe abstrata define os *hot-spots* do *framework*, e os passos do algoritmo que regula o fluxo de interação entre o agente e a arquitetura. As estruturas são descritas na Seção 3.4.

O modelo de comunicação foi construído considerando dois conceitos básicos da arquitetura de referência: mensagens e serviços. Sob a ótica do MOM, a comunicação entre os agentes ocorre através de um caminho, que tem início no

remetente (sender) e termina no destinatário (receiver). Ao longo do caminho, a mensagem pode passar por vários intermediários. Sob a ótica do SOM, as mensagens são trocadas entre um agente requisitante (requester) e um agente provedor de serviços (provider). Isto significa que a comunicação entre um agente requisitante e um agente provedor no SOM é implementada ou traduzida na passagem da mensagem entre o remetente (sender) e o destinatário (receiver) no MOM [121]. O modelo de comunicação é descrito na Seção 3.3.

Um dos benefícios do modelo de papéis é o reuso: um agente pode evoluir com poucas modificações radicais porque papéis são reusáveis [121]. Um agente, tanto intermediário quanto de uma aplicação pode mudar dinamicamente o seu comportamento quando uma nova responsabilidade é atribuída a um papel. Esta propriedade é alcançada pela capacidade de re-configuração dinâmica provida pela arquitetura. Os papéis dos agentes são descritos na seção 3.2.

A abordagem estende as especificações da arquitetura de referência WSA, introduzindo o conceito de agente abstrato e separando as responsabilidades e os papéis de agentes intermediários do papel dos agentes que participam de aplicações. Combinado com agentes, o *blackboard*, utilizado na abordagem, supre uma deficiência do paradigma SOA: a falta de suporte ao modelo de comunicação dos agentes.

3.1 Conceitos Básicos

A introdução do conceito de agente para desempenhar os papéis definidos pela arquitetura de referência WSA atende às atuais expectativas para arquiteturas Web: a necessidade de serem flexíveis, dinâmicas, pró-ativas e adaptativas. A arquitetura básica do MIDAS é composta por um conjunto de agentes colaborativos de *middleware* (intermediários), agentes de aplicações e componentes. Os agentes de aplicações e componentes estão representados como subclasses das classes abstratas Agent e Component, respectivamente.

Na plataforma MIDAS, os elementos da arquitetura são baseados na coexistência de vários containers, cada um executando uma JVM (Java Virtual Machine). Cada máquina virtual é um container básico que provê um ambiente de execução completo, onde agentes podem executar concorrentemente no mesmo

host. A Figura 11 mostra a arquitetura genérica do MIDAS, detalhando a configuração servidor de front-*end* (FES), um container de agentes (AC) e um container de componentes (CC). O servidor e o container possuem configurações específicas. O servidor é responsável pela integração dos containers registrados, e da plataforma com aplicações externas. Ele possui duas portas: uma SOAP para interoperar com aplicações externas e uma HTTP para interoperar com os containers. O servidor não possui organizações, agentes de aplicações ou componentes; contém somente agentes intermediários.

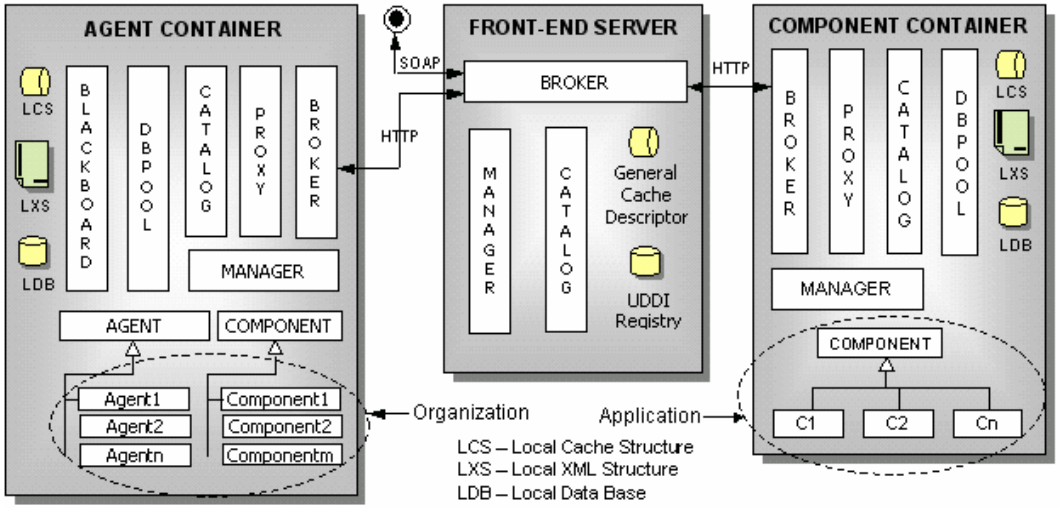

Figura 11 - A plataforma MIDAS e os principais conceitos

Os agentes intermediários provêem serviços de infraestrutura e separam o comportamento genérico da arquitetura do comportamento específico dos agentes de aplicações. São utilizados os seguintes tipos de agentes intermediários: Broker, que desempenha o papel definido pelo modelo orientado a mensagens (MOM); Proxy, que desempenha o papel definido pelo modelo orientado a serviços (SOM); Catalog, que desempenha o papel definido pelo modelo orientado a recursos (ROM) e Manager, que desempenha as funções definidas pelo modelo de gerenciamento. Além destes quatro agentes, foram utilizados mais outros dois agentes intermediários: o Blackboard e o DBPool. O Blackboard provê um poderoso mecanismo de comunicação e suporte ao *workflow* dos agentes, e o agente DBPool oferece facilidades para configurar fontes de acesso a dados e integração com *frameworks* especialistas em mapeamento objeto-relacional.

Agent é uma classe abstrata que representa as particularidades genéricas comuns a todos s agentes que participam das aplicações. Os agentes são implementados estendendo a classe abstrata, que provê as interfaces padrão que regulam o fluxo de interação entre o agente e a arquitetura e os procedimentos para gerenciamento do ciclo de vida. *Component* é uma classe abstrata que representa entidades puramente reativas, normalmente *utilizadas* para encapsular regras específicas do domínio da aplicação, tais como processos de negócios, e objetos de acesso a dados. *Organization* são entidades virtuais localizadas em AC's que podem conter agentes e componentes.

Em um container, o elemento LXS representa os recursos do container descritos em XML. O elemento LCS representa os recursos que são mantidos em memória *cachê* e o elemento LDB representa banco de dados locais, que podem ser manipulados pelos agentes. No servidor, o elemento GCD representa a estrutura consolidada de recursos mantida em memória *cache*. O servidor também é responsável pela criação automática e manutenção dos registros WSDL e UDDI.

Basicamente, os serviços de infraestrutura da plataforma são providos por um conjunto de agentes colaborativos que desempenham de maneira dinâmica e pró-ativa os papéis definidos pelos modelos da arquitetura. A Tabela 1 ilustra os agentes intermediários, identificando o modelo que eles atuam e as propriedades de agência que eles implementam. Os quatro primeiros desempenham as funções definidas pela arquitetura de referência WSA, e os dois últimos provêem suporte ao projeto detalhado dos agentes e pertencem ao Agent Model (AM).

	Modelo	Autonomia	Interação	Adaptação	Colaboração
Broker	MOM				
Proxy	SOM				
Catalog	ROM				
Manager	MGT				
Blackboard	AM				
DBPool	AM				

Tabela 1 - Tipos de agentes de *middleware* e as suas propriedades

As propriedades de agência foram definidas no Capítulo 2, (Seção 2.1.1) e foram utilizadas preferencialmente aquelas consideradas necessárias pela OMG para caracterizar agentes de software. A propriedade de mobilidade foi desconsiderada porque ela não se aplica neste contexto. Não foi considerada a propriedade de aprendizado, já que na atual versão, nenhum dos agentes

intermediários exercita esta propriedade, embora alguns possam ter potencial para desenvolvê-la. As propriedades de interação e colaboração são exercitadas por todos os agentes. Todos possuem características adaptativas: o seu comportamento pode ser alterado em tempo de execução. As propriedades são comentadas no decorrer da descrição dos papéis, apresentada na próxima seção.

A plataforma oferece facilidades para o desenvolvimento, resultante do modelo de arquitetura do *framework*. Dentre estas, podem ser citadas:

(i) assistentes GUI's para as tarefas de gerenciamento do ciclo de vida da plataforma e dos recursos (organizações, agentes, serviços, etc.);

(ii) suporte para visualização e descoberta de serviços em GUI's navegáveis;

(iii) serviços sistemáticos de monitoramento, tais como verificação de sincronismo, verificação de disponibilidade de serviços remotos;

(iv) capacidade de configuração dinâmica dos agentes da arquitetura e das aplicações, possibilitando alterar seu comportamento em tempo de execução;

(v) ferramentas embarcadas, tais como o Tomcat para servidor de rede, o Axis para geração WSDL/UDDI e opcionalmente o Hibernate para o mapeamento objeto-relacional;

(vi) classe abstrata que provê flexíveis *hot-spots* para facilitar a implementação das particularidades específicas dos agentes;

(vii) objetos *wrapper* de requisição que abstraem a complexidade de requisições de serviços;

(viii) agente Blackboard, que provê um poderoso mecanismo para o modelo de comunicação e suporte ao *workflow* dos agentes;

(ix) interoperabilidade bidirecional com sistemas externos e aplicações heterogêneas através de serviços Web;

(x) geração automática das especificações de serviços MIDAS para serviços Web e para o encapsulamento de serviços Web no MIDAS.

Os fundamentos utilizados para projetar e implementar estas características e propriedades estão descritos neste capítulo, que inicia detalhando os papéis dos agentes e finaliza mostrando as estruturas que implementam os papéis. Uma visão prática pode ser obtida no Capítulo 4, na descrição dos estudos de casos.

3.2
O Modelo de Papéis

A idéia de associar responsabilidades a um agente através de um modelo de papéis é parte de uma abordagem de projeto orientado a comportamento [Beck1989]. O primeiro princípio desta abordagem é que cada elemento (objeto, componente, agente, etc.) deve ter um único papel, claramente definido através de um bloco coerente de funções. O modelo de papéis tem origem na teoria dos papéis [11]. Um papel se refere a um padrão de comportamento esperado para um agente, sendo composto por obrigações e direitos. As obrigações especificam as atividades que um agente precisa executar para cumprir suas responsabilidades. Direitos são modelados como permissões que especificam quais recursos um agente pode dispor para cumprir as suas responsabilidades. Recurso é alguma coisa passivamente utilizada durante a execução da atividade [118].

Para facilitar a representação dos papéis, utilizamos cartões CRC (Class-Responsability-Collaborator) [7]. O cartão CRC pode simular de forma adequada a representação dos papéis, já que ele possui as responsabilidades descritas. As colaborações podem ajudar a visualizar os direitos ou permissões dos agentes. Na primeira parte desta seção são descritos os papéis dos agentes intermediários, e na segunda parte os papéis dos agentes de aplicações.

3.2.1
Os Papéis dos Agentes Intermediários

Os agentes intermediários são tipos de agentes que provêem serviços de infraestrutura, mapeados a partir das responsabilidades definidos pela arquitetura WSA. Os agentes intermediários podem estar localizados em um container ou no servidor. Para cada local, eles possuem responsabilidades específicas. Utilizamos o prefixo S para indicar que o agente atua no servidor e o prefixo C para indicar que ele atua em um container (p. ex., S-Broker e C-Broker).

3.2.1.1
O Agente Broker

O agente *Broker* desempenha o papel definido pelo MOM, focalizando os aspectos da arquitetura relacionados ao transporte de mensagens: atender e enviar

requisições, empacotar-desempacotar requisições e gerenciar exceções. Quando localizado em um container AC (ou CC), o agente C-Broker atua como uma porta de entrada e saída. Todas as mensagens remotas que entram ou saem do container passam por ele. As principais responsabilidades e colaborações do agente C-Broker podem ser vistas na Figura 12. Para desempenhar as responsabilidades, ele colabora com os agentes S-Broker, Proxy, C-Manager e C-Catalog.

Agente **C-Broker**	**Colaboradores**
Responsabilidades ▪ Atender/Enviar requisições de: – serviços – verificação de disponibilidade de serviço – verificação de sincronismo ▪ Enviar requisições de registro ▪ Empacotar/desempacotar requisições ▪ Recuperar exceções	▪ S-Broker ▪ Proxy ▪ C-Manager ▪ C-Catalog

Figura 12 - Responsabilidades e colaborações do agente C-Broker

As requisições remotas de serviços que chegam ao agente C-Broker são sempre direcionadas pelo servidor. O agente C-Broker desempacota as requisições colocando-a no formato do protocolo interno do MIDAS e as encaminha ao agente Proxy, que cria dinamicamente a instância e invoca o agente (ou componente) provedor do serviço. As requisições de verificação de disponibilidade checam a possibilidade de um serviço estar disponível ou não no container em um determinado momento. As requisições de verificação de sincronismo são enviadas para interrogar o container sobre o seu estado (sincronizado ou não sincronizado).

Da mesma forma que atende, o C-Broker pode também enviar os mesmos três tipos de requisição definidos acima. Sempre que uma requisição de serviço é encaminhada para o servidor, ela é empacotada como um objeto de requisição. O empacotamento facilita a manipulação de dados, abstraindo a ordem como os parâmetros da requisição são passados e evitando a necessidade de conversões (casting). Para enviar uma requisição de registro, o C-Broker estabelece e valida uma conexão com o servidor, e envia a estrutura de recursos, obtida com a colaboração do agente C-Catalog.

O agente C-Broker colabora com o agente C-Manager durante os procedimentos de verificação de sincronismo, mantendo-o informado sobre o estado do servidor. Quando ocorre uma situação de erro, ele captura as exceções e repassa a informação para o C-Manager.

Quando localizado no servidor, o agente Broker possui responsabilidades distintas daquelas desempenhadas em um container. No servidor, ele atende requisições de registro de containers, e é responsável pela interoperabilidade da plataforma com aplicações externas, via serviços Web. As responsabilidades e colaborações do agente S-Broker podem ser vistas na Figura 13.

Agente **S-Broker**	**Colaboradores**
Responsabilidades ▪ Atender requisições de: – registro – verificação de sincronismo ▪ Enviar requisições de verificação de sincronismo ▪ Direcionar requisições de: – serviços – verificação de disponibilidade de serviços ▪ Interoperabilidade – enviar requisições SOAP – atender requisições SOAP – gerar especificações WSDL-UDDI ▪ Recuperar exceções	▪ C-Broker ▪ S-Manager ▪ S-Catalog ▪ Adapter

Figura 13 - Responsabilidades e colaborações do agente S-Broker

Quando recebe uma requisição de registro enviada por um container, o S-Broker desempacota a requisição e a encaminha para o agente S-Catalog efetuar a atualização da estrutura de recursos. Quando recebe uma requisição de verificação de sincronismo, ele retorna um *boolean* confirmando o seu próprio estado.

O agente S-Broker tem também a responsabilidade de direcionar requisições de serviço e de verificação de disponibilidade de serviço encaminhadas pelos containers. Para direcionar uma requisição de serviço, o procedimento básico é localizar o container responsável (com a colaboração do agente S-Catalog) e encaminhar a requisição de serviço. Procedimento similar ocorre para verificar a disponibilidade de um serviço em um container.

Os mecanismos de interoperabilidade bidirecional com serviços Web são executados por um componente Adapter [21] que faz parte da estrutura interna do S-Broker. O Adapter é capaz de receber solicitações SOAP e convertê-las para o formato do protocolo MIDAS, assim como converter do MIDAS para SOAP.

O agente S-Broker possui certo grau de autonomia, pois quando ele percebe que um container não se encontra sincronizado, passa a rejeitar as requisições que seriam normalmente encaminhadas para ele. Ele também é responsável pela geração automática e manutenção das descrições WSDL e dos registros UDDI. Neste caso, ele possui características adaptativas porque antes de criar as especificações WSDL para os serviços Web, ele efetua um *look-up* sobre os registros existentes. Caso existam registros UDDI cadastrados, além de criar as especificações WSDL, ele publica os serviços Web nestes registros. Novos registros podem ser inseridos dinamicamente sob a contínua supervisão do agente, que se adapta às novas entradas.

3.2.1.2 O Agente Proxy

O agente *Proxy* desempenha o papel definido pelo SOM e focaliza os aspectos da arquitetura relacionados ao processamento das mensagens e execução dos serviços. Ele representa o objeto servidor remoto, invocado pelos clientes. Na plataforma MIDAS, o Proxy atua somente em containers, já que no servidor não existem provedores nem requisitantes de serviços. As principais responsabilidades e colaborações do agente Proxy podem ser vistas na Figura 14.

Agente **Proxy**	**Colaboradores**
Responsabilidades ▪ Atender requisições de serviços ▪ Processar as mensagens: – validar serviço – identificar provedor do serviço ▪ Encaminhar requisições de serviços remotos ▪ Configuração dinâmica ▪ Criação dinâmica de instâncias	▪ C-Broker ▪ C-Manager ▪ C-Catalog ▪ Agent

Figura 14 - Responsabilidades e colaborações do agente Proxy

As requisições de serviço atendidas pelo Proxy podem ter duas origens: local ou remota. No primeiro caso, elas têm origem em agentes locais, e no segundo caso elas representam requisições de agentes remotos, que são encaminhadas pelo agente C-Broker. Durante o processamento da mensagem, são executadas as seguintes atividades:

- *validar serviço*: verificar se a requisição para o serviço está correta, e se o serviço pode ser disponibilizado para o requisitante;
- *habilitar sincronização de mensagens*: acessos simultâneos a um componente precisam ser sincronizados; o agente C-Manager efetua os procedimentos de enfileiramento;
- *identificar entidade provedora do serviço*: efetuada com a colaboração do agente C-Catalog, que provê um método para recuperar o nome da entidade (agente/componente) responsável pela execução do serviço.

O agente Proxy atua como um representante do provedor. Quando um agente precisa de um serviço de outro agente local ou remoto, ele simplesmente passa o nome do serviço e os parâmetros para o agente Proxy. O Proxy exerce um papel fundamental também para a comunicação dos agentes com as camadas de apresentação e de acesso a dados. Quando um agente precisa requisitar um serviço provido por uma JSP, ele envia a requisição para o Proxy, que direciona a chamada para o *servlet* gerenciador. Este papel desempenhado pelo Proxy garante transparência de comunicação entre as camadas (lógica, apresentação, dados) e redução de linhas de código, como demonstrado na seção 5.6.

As atividades de configuração e criação dinâmica de instâncias envolvem diferentes mecanismos. A configuração dinâmica focaliza a capacidade de um agente que está executando na JVM poder ser substituído por outro modificado. Os procedimentos de configuração dinâmica são comandados pelo usuário através de uma GUI e efetuados com o auxílio da biblioteca *java.net.URLClassLoader*, que possibilita carregar e substituir classes que estão em execução na JVM.

A criação dinâmica de instâncias focaliza a capacidade de poder incluir novas classes e abstrair o processo de criação de instâncias sem precisar alterar o código fonte. A utilização dos padrões de criação Abstract Factory e Factory Method [48] provê o pré-requisito fundamental para abstrair o processo de criação de instâncias, desde que as classes não mudem em tempo de execução.

O agente Proxy desempenha papel fundamental para tornar a arquitetura flexível e adaptável. De acordo com Ferber [38], adaptação é uma propriedade que pode ser vista sob duas perspectivas: como uma característica individual de um agente ou como um processo coletivo. Como um processo coletivo, significa capacidade de evolução. Na plataforma, os mecanismos para a re-configuração dinâmica providos pelo Proxy possibilitam que o comportamento dos agentes possa ser modificado em tempo de execução, substituindo funcionalidades já existentes ou acrescentando novas funcionalidades.

3.2.1.3
O Agente Catalog

O agente Catalog desempenha o papel definido pelo modelo orientado a recursos (ROM), focalizando os aspectos da arquitetura relacionados com a descrição e a representação dos recursos. Quando localizado em um container, o agente C-Catalog tem a responsabilidade de manter a descrição XML dos recursos, assim como as representações em memória. Os procedimentos para manipular as informações XML são efetuados de forma automática pela classe Parser, que faz parte da estrutura do agente. A Figura 15 ilustra as principais responsabilidades e colaborações do agente C-Catalog.

Agente **C-Catalog**	**Colaboradores**
Responsabilidades ▪ Atender requisições de: – atualização da estrutura local de serviços – informações s/ a estrutura de recursos ▪ Manter a descrição e representação da estrutura de recursos: – em XML – em memória *cache* ▪ Prover mecanismos para: – localização de serviços – descoberta de serviços	▪ C-Manager ▪ C-Proxy ▪ Parser ▪ Agent

Figura 15 - Responsabilidades e colaborações do agente C-Catalog

Conforme pode ser visto na Figura 15, o agente C-Catalog pode atender a três tipos de requisição. As requisições para atualização da estrutura de recursos

têm origem no agente C-Manager, durante os procedimentos de registro e manutenção efetuados pelo desenvolvedor nas interfaces gráficas. Quando algum recurso novo é adicionado ou atualizado, as informações são persistidas em formato XML. Este procedimento é efetuado pela classe Parser. As requisições para localização e fornecimento de informações sobre a estrutura de recursos do container visam obter dados sobre as entidades, tais como nome, tipo, localização, dentre outras. Elas podem ter três origens: (i) agente C-Manager, que utiliza as informações para exibir nas interfaces gráficas habilitando a descoberta de serviços; (ii) agente Proxy, que solicita o nome e a localização de entidades provedoras de serviços e (iii) agentes de aplicações, que podem obter uma representação da estrutura para a descoberta de serviços.

Para tornar o processamento mais eficiente, as descrições XML são mantidas em memória *cache* em duas representações. A primeira delas é uma representação hierárquica onde o nó raiz é o container. A segunda é uma representação indexada pelo nome do serviço, que provê um meio de acesso rápido a um serviço, utilizando somente o nome como identificador, sem precisar percorrer a estrutura para localizar o serviço.

O agente C-Catalog possui características dinâmicas e pró-ativas, pois consegue se adaptar às novas configurações e manter as estruturas de recursos sempre atualizadas. Ele possui autonomia para excluir a estrutura de recursos de containers que perdem o sincronismo com o servidor. Ele desempenha funções similares àquelas definidas pelo agente DF (Directory Facilitator), utilizado na especificação FIPA, e por outros agentes citados na literatura tais como Matchmakers e Yellow Pages [29, 16].

Quando localizado no servidor, o agente Catalog possui algumas responsabilidades distintas daquelas desempenhadas em um container. No servidor, não existe descrição XML da estrutura de recursos porque no servidor não existem recursos. No servidor, ele consolida as representações dos recursos alocados nos containers, mantendo-as em memória *cachê* e em arquivos *serializados*. A representação mantida no servidor é uma descrição da hierarquia consolidada de recursos que reflete o estado de todos os elementos da plataforma em um determinado momento.

A Figura 16 mostra as responsabilidades e colaborações do agente S-Catalog. As requisições que chegam ao S-Catalog são encaminhadas pelo

agente S-Broker, e podem ser de dois tipos: de atualização da estrutura global ou para localização de containers provedores de serviços. As requisições de atualização são enviadas sempre que um container deseja se registrar na plataforma ou atualizar a sua estrutura de recursos.

Agente **S-Catalog**	**Colaboradores**
Responsabilidades ▪ Atender requisições de: – atualização da estrutura global de recursos – localização de containers ▪ Manter representação da estrutura global de recursos: – em memória cache – em arquivo serializado ▪ Prover mecanismos para: – localização de containers – descoberta de serviços	▪ S-Broker ▪ S-Manager ▪ Serializer

Figura 16 - Responsabilidades e colaborações do agente S-Catalog

As representações da estrutura consolidada de recursos são mantidas em memória *cache* e persistidas em um arquivo 'serializado'. Este procedimento é efetuado com o auxílio de uma classe Serializer, que faz parte da estrutura interna do S-Catalog. Quando um container perde o sincronismo, a sua estrutura mantida em memória é excluída, mas continua persistida no servidor. Isto possibilita que os recursos de um container possam ser visualizados e descobertos mesmo que num determinado momento ele não esteja sincronizado com o servidor.

3.2.1.4 O Agente Manager

O agente Manager focaliza os aspectos da arquitetura relacionados com as políticas de gerenciamento. Ele é o agente mais complexo da arquitetura, e executa as tarefas de gerenciamento do ciclo de vida, monitoramento e captura de métricas e estatísticas [14]. As principais responsabilidades e colaborações do agente C-Manager podem ser vistas na Figura 17.

Agente **C-Manager**	**Colaboradores**
Responsabilidades ▪ Gerenciamento do ciclo de vida – do container – dos agentes ▪ Gerenciamento dos recursos – visualização – construção ▪ Tarefas sistemáticas – monitoramento – captura de métricas ▪ Gerenciamento de requisições – *pool* de execução – síncronas e assíncronas – *wrapping* de requisições	▪ C-Broker ▪ C-Catalog ▪ Proxy ▪ Agent

Figura 17 - Responsabilidades e colaborações do agente C-Manager

As tarefas de gerenciamento do ciclo de vida envolvem um conjunto de atividades relacionadas ao controle do ciclo de vida do container e dos agentes, gerenciamento de transações, serviços, recursos e captura de métricas e estatísticas. O Agente C-Manager oferece duas perspectivas de gerenciamento: (i) uma de visualização, que fornece uma visão navegável da hierarquia de recursos e uma visão dinâmica do ambiente de execução do container e (ii) outra de construção, para auxiliar as atividades de registro e manutenção dos recursos.

O C-Manager executa de forma periódica e sistemática um conjunto de atividades de monitoramento. Elas são executadas com a colaboração do agente C-Broker, que envia requisições de verificação de sincronismo para questionar a disponibilidade do servidor e dos serviços. As atividades de captura de métricas e estatísticas envolvem um conjunto de procedimentos, tais como: manutenção de um arquivo *log* de transações, registro de falhas e exceções, e tempo de latência na comunicação remota.

A tarefa de gerenciamento de requisições envolve as atividades de gerenciamento de transações, assim como controle de concorrência e sincronismo das mensagens. O agente C-Manager utiliza um *pool* de execução, capaz de lidar com alta concorrência e manter ao mesmo tempo um *log* com todas as transações sendo executadas.

Embora similar, o papel do agente Manager no servidor possui diferenças daquele desempenhado em um container. No servidor não existe perspectiva de

construção, já que o registro dos recursos é feito somente nos containers. Da mesma forma, o gerenciamento dos recursos é diferente. No servidor, a perspectiva de visualização é mais abrangente, pois envolve a consolidação de todos os recursos da plataforma. A Figura 18 ilustra as principais colaborações e responsabilidades do agente S-Manager.

Agente **S-Manager**	**Colaboradores**
Responsabilidades ▪ Gerenciamento do ciclo de vida – da plataforma – dos containers ▪ Gerenciamento dos recursos ▪ Tarefas sistemáticas – monitoramento – captura de métricas	▪ S-Broker ▪ S-Catalog

Figura 18 - Responsabilidades e colaborações do agente S-Manager

As tarefas de gerenciamento do ciclo de vida da plataforma envolvem atividades de controle sobre o acesso e uso do servidor e dos containers, tais como iniciar, suspender e finalizar. Quando o servidor é iniciado, o agente S-Manager levanta o servidor de rede e inicializa os agentes da arquitetura. A partir deste momento, ele está habilitado a registrar containers, receber atualizações e atender requisições remotas de serviços.

Para efetuar as atividades de monitoramento o agente S-Manager conta com a colaboração do agente S-Broker, que envia aos containers mensagens de verificação de sincronismo. As tarefas de captura de métricas e estatísticas envolvem o monitoramento constante de um conjunto de propriedades, tais como as transações que estão sendo executadas, a quantidade de memória em uso, e a captura dos erros e exceções.

O agente S-Manager possui algum grau de autonomia como responsável pela inicialização e gerenciamento do ciclo de vida da plataforma. Durante o gerenciamento do ciclo de vida, ele tem autonomia para desarmar um container ou o próprio servidor, quando forem detectados erros muito graves ou não previstos. Ele pode também ser considerado adaptativo porque em colaboração com o agente S-Broker avalia continuamente o estado dos containers. Quando

algum container se encontra fora do ar, ele modifica o estado deste container para não-sincronizado, e reflete a mudança na interface gráfica e na estrutura de recursos mantida em memória.

3.2.1.5
O Agente Blackboard

O agente Blackboard desempenha papel fundamental na arquitetura provendo um poderoso meio de suporte ao modelo de comunicação e *workflow* dos agentes. A Figura 19 ilustra as principais responsabilidades do agente Blackboard, que estão distribuídas em três classes: Board, Message e Control, que possuem responsabilidades específicas na estrutura do Blackboard. As implementações concretas de Agent representam as fontes de conhecimento.

Agente **Blackboard**	**Colaboradores**
Responsabilidades ▪ Registrar ouvintes – grupos de agentes – agentes ▪ Enviar mensagens – para todos – para grupos – para um agente ▪ Manter mensagens: – rotear mensagens – recuperar mensagens – arquivar mensagens – manter arquivo log ▪ Manter variáveis globais ▪ Notificar agentes	▪ Agent (Knowledge source) ▪ Board ▪ Message ▪ Control

Figura 19 - Responsabilidades e colaborações do agente Blackboard

O Blackboard possibilita que os agentes possam trocar mensagens uns com os outros, com grupos de agentes ou com todos os agentes. Esta capacidade supre uma deficiência encontrada em arquiteturas orientadas a serviços, cuja estrutura não provê suporte a comunicação ponto-a-ponto. A manutenção de uma área de variáveis compartilhadas possibilita que os agentes possam capturar as mudanças ambientais e dirigir seu comportamento.A classe Board é responsável pelo registro e manutenção de ouvintes. Quando o *blackboard* é inicializado, ela

registra todos os agentes como potenciais ouvintes, utilizando o seu nome como identificador. Ela provê mecanismos para os agentes se registrarem em grupos de interesse e poderem enviar mensagens para todos, para grupos ou para um agente individual. Ela também possibilita manter uma estrutura de dados visível para todos os agentes. Durante a resolução de um problema, as ações executadas pelos agentes vão gradualmente modificando a estrutura de dados e o estado da solução.

A classe Message oferece um conjunto de serviços para manter mensagens que chegam ao *blackboard*. Os agentes podem ter acesso a este *log* de mensagens e recupera-las por grupo, por data, por assunto e assim por diante. A classe Control é responsável pelo monitoramento e notificação dos agentes ou grupos de agentes sobre as modificações no *blackboard*. Ela funciona como um sensor para os agentes, capturando as mudanças na estrutura de dados e notificando os agentes ou grupos de agentes interessados.

3.2.1.6 O Agente DBPool

O agente DBPool padroniza os procedimentos para acesso a dados possibilitando a fácil obtenção de objetos de conexão com banco de dados e integração com *frameworks* especialistas em mapeamento objeto-relacional. O objetivo é facilitar o projeto detalhado e implementação dos agentes. A Figura 20 ilustra as principais responsabilidades e colaborações do agente DBPool.

Agente **DBPool**	**Colaboradores**
Responsabilidades ▪ Prover objetos de conexão com banco de dados ▪ Prover métodos para efetuar *queries* ▪ Integrar *frameworks* especialistas em mapeamento objeto-relacional	▪ DBHandler ▪ Proxy ▪ Agent ▪ Component

Figura 20 - Responsabilidades e colaborações do agente DBPool

As descrições de fontes de dados são efetuadas no arquivo XML, e podem ser obtidas pelo agente DBPool com a colaboração do agente Proxy. A partir destas especificações, o agente DBPool pode obter e criar objetos de conexão com

banco de dados. Utilizando o objeto, é possível ao agente requisitar a criação de um *Statement* e utilizar os conhecidos métodos *getSq* e *setSql* para manipular informações. Para assegurar a integridade de acesso ao banco, o agente DBPool possui uma classe na sua estrutura chamada DBHandler que utiliza *pools* de execução. Ela controla a criação e a quantidade de conexões abertas, e quando o número de conexões é excessivo, forma uma fila de espera (*pooling*), não permitindo que haja *overflow* de requisições e garantindo a estabilidade do banco.

O DBPool possui características adaptativas porque consegue se adaptar dinamicamente a diferentes configurações de banco de dados. Novos bancos de dados podem ser inseridos dinamicamente, e ele se adapta às novas especificações.

3.2.2 Os Papéis dos Agentes de Aplicações

O Agent Model (AM) foi projetado para fornecer uma estrutura de suporte ao projeto detalhado dos agentes. Em sua estrutura básica, o modelo é composto por duas classes abstratas Agent e Component, pelas classes concretas que estendem as classes abstratas e pelos agentes intermediários Blackboard e DBPool, que provêem suporte ao *workflow* e modelo de comunicação dos agentes. A classe abstrata Agent define um esqueleto de algoritmo que representa um agente abstrato. A classe Component é um objeto *wrapper* cuja finalidade é facilitar a criação de componentes que encapsulam objetos de acesso a dados e funções específicas do domínio da aplicação. As classes Agent e Component fornecem os flexíveis hot-spots [37, 94] do *framework*.

3.2.2.1 A Classe Abstrata Agent

A classe Agent é uma representação abstrata de um agente que possui o seu próprio *thread* de execução, interfaces bem definidas para se comunicar com outros agentes e funções para interagir com o ambiente via *blackboard*. Ela define as interfaces de agentes que atuam em um modelo de arquitetura fracamente acoplado. As interfaces regulam o fluxo de interação entre os agentes e a arquitetura, tornando transparente o processo de enviar e receber requisições

locais, remotas ou de serviços Web. A classe Agent possui alguns métodos concretos e outros abstratos. Os métodos concretos são considerados *frozen-spots* [94], e os métodos abstratos são os *hot-spots* [94], e precisam ser completados pelo desenvolvedor. As principais responsabilidades e as colaborações da classe abstrata Agent podem ser vistas na Figura 21.

Agente **Abstract Agent**	**Colaboradores**
Responsabilidades ▪ Definir interfaces – provedora (*hot-spot*) – requerente (*frozen*) ▪ Ciclo de vida: – thread de execução (*frozen*) – lifecycle (*hot-spot*) ▪ Assinaturas para controle do ciclo de vida (iniciar...) ▪ Método para recuperar objetos de conexão com SGBD (*frozen*)	▪ Concrete-Agent ▪ Blackboard ▪ DBPool ▪ Proxy

Figura 21 - Responsabilidades e colaborações da classe abstrata Agent

A classe Agent oferece duas interfaces para comunicação externa: (i) uma interface *provedora* por onde o agente pode receber requisições de serviços, que provê um dos *hot-spots* do *framework* e precisa ser completada pelo desenvolvedor e (ii) uma interface *requerente*, por onde o agente pode requisitar serviços externos. A interface requerente implementa um método concreto, ou uma parte *frozen* da classe. Agent possui um método *call*, herdado da interface Java Callable que dispara o seu *thread* de execução. Este método é *frozen*, e é acionado toda vez que é criada uma nova instância do agente. O método *lifeCycle* dispara o *loop* de ciclo de vida do agente. Ele é *um hot-spot*, e precisa ser completado pelo desenvolvedor. Agent provê também assinaturas básicas para iniciar, suspender e finalizar a execução de um agente. Os eventos que disparam estes métodos têm origem no C-Manager, e podem ser de dois tipos: (i) comandados por interfaces gráficas ou (ii) comandados pelo próprio agente C-Manager, quando, por exemplo, ele inicializa os agentes da arquitetura.

Agentes podem possuir sensores para permitir a percepção de mudanças no ambiente, e um meio de reação a estas mudanças. A classe Agent define uma

interface (ou um sensor) que pode ser implementada pelo agente. A interface estabelece um canal de comunicação do agente com o ambiente através do Blackboard. Através da interface, o agente é notificado sempre que uma das variáveis (ou objetos) da estrutura de dados compartilhada é alterada ou incluída.

3.2.2.2 A Classe Abstrata Component

A classe abstrata Component atua como um *wrapper*, capaz de encapsular objetos de acesso a dados, funcionalidades específicas do domínio, aplicações legadas, serviços Web, partes de processos de negócios e outros procedimentos. Diferente de agentes, componentes são entidades puramente reativas, que não possuem *thread* de execução, embora possam executar concorrentemente. A Figura 22 ilustra as responsabilidades e colaborações da classe Component.

Agente **Abstract Component**	**Colaboradores**
Responsabilidades ▪ Definir interfaces – Provedora (*hot-spot*) – Requerente (*frozen*) ▪ Método para recuperar objetos de conexão com SGBD (*frozen*)	▪ Concrete-Component ▪ DBPool ▪ Proxy

Figura 22 - Responsabilidades e colaborações da classe abstrata Component

Component possui apenas duas interfaces, uma provedora e outra requerente, que funcionam de forma similar àquelas descritas para a classe Agent. A interface provedora é aquela por onde os componentes recebem requisições de serviços. Ela precisa ser implementada pelo desenvolvedor para efetuar o tratamento do serviço recebido e direcioná-lo para o método manipulador. A interface requerente é *frozen*, e define o fluxo de interação entre os componentes e a arquitetura. Através da interface provedora, os componentes podem requisitar serviços externo, inclusive serviços Web de forma transparente e simples.

Component provê ainda um caminho para facilitar a integração entre agentes e serviços Web. Para que serviços Web possam trabalhar de forma colaborativa para compor uma solução, é necessário agregar comportamento ao

serviço. Componentes, assim como agentes, podem ser utilizados para esta finalidade. A classe Component pode também ser utilizada para encapsular chamadas para aplicações heterogêneas, implementadas com CORBA ou JNI, auxiliando a integração da plataforma com aplicações heterogêneas.

3.3 O Modelo de Comunicação

A tarefa de requisitar um serviço é simplificada pelos mecanismos de transparência providos pela arquitetura. Quando um agente deseja requisitar um serviço local, remoto ou mesmo um serviço Web, ele utiliza o mesmo procedimento, como se todos os provedores estivessem localizados no mesmo container. Ele obtém da sua interface requerente um objeto *wrapper* de requisição, e pode invocar o serviço de forma síncrona ou assíncrona. A comunicação externa é mediada pelo agente Proxy, que regula o fluxo de interação entre os agentes provedores e requisitantes de serviços e a arquitetura. Utilizar comunicação assíncrona é uma maneira de otimizar o processamento do agente, que pode solicitar um serviço e continuar a executar suas tarefas, sem se preocupar com a resposta.

O modelo de comunicação define a forma como os agentes que participam da plataforma se comunicam, e como as mensagens trafegam entre os agentes provedores e requisitantes de serviços. Consideramos três abstrações para caracterizar o modelo de comunicação: (i) extra-plataforma, que define a comunicação entre a plataforma e o mundo exterior; (ii) intra-plataforma, que descreve a comunicação entre agentes localizados em containers diferentes no interior da plataforma e (iii) intra-container, que caracteriza o modelo de comunicação entre os agentes localizados em um mesmo container.

A Figura 23 mostra a primeira abstração, que define a comunicação entre a plataforma e aplicações externas. A comunicação é bi-direcional: Ela pode ocorrer de aplicações externas para a plataforma e da plataforma para aplicações externas. No primeiro caso, as requisições SOAP são recebidas pelo agente S-Broker, que as converte para o protocolo MIDAS, que utiliza http para o transporte da

mensagem. Neste caso, sistemas externos ou outras plataformas podem requisitar serviços publicados como serviços Web.

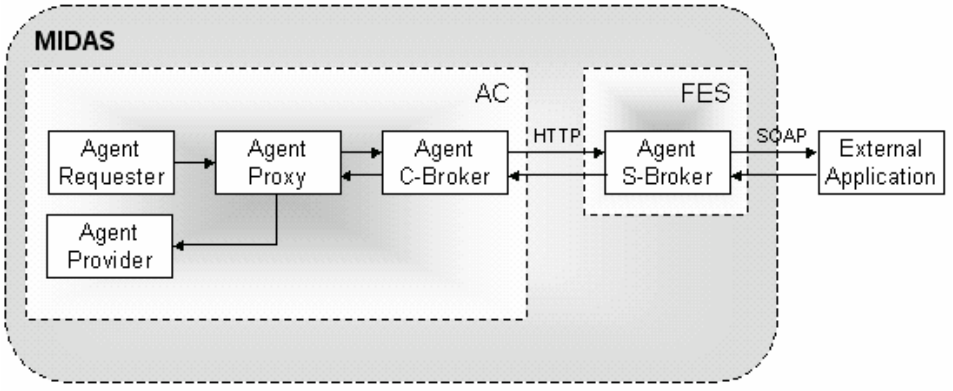

Figura 23 - Caminho da mensagem em uma requisição de serviço Web

No segundo caso, os agentes (ou componentes) inseridos na plataforma podem requisitar serviços Web oferecidos por sistemas externos. O agente que deseja requisitar o serviço Web efetua a requisição de forma transparente, como se estivesse requisitando um serviço local. A requisição é encaminhada pelos agentes Proxy e C-Broker até o servidor, onde o agente S-Broker converte a requisição MIDAS para o formato SOAP e invoca o serviço da aplicação externa.

Todas as funcionalidades para enviar e receber serviços Web e os mecanismos para a manipulação e geração das especificações WSDL e UDDI são efetuadas de forma automática pela plataforma. A requisição de um serviço Web por um agente é feita de forma totalmente transparente. Quando ele deseja requisitar um serviço Web, ele obtém da sua interface requerente um objeto *wrapper* de requisição que representa o serviço Web, e adiciona os parâmetros quando sem precisar seguir ordem de colocação pré-definida ou efetuar *castings*.

A segunda modalidade (inter-container) ocorre entre agentes localizados em containers remotos. Conforme pode ser visto na Figura 24, uma requisição remota transita entre vários intermediários no caminho entre o agente requisitante e o provedor do serviço. No MOM, a comunicação remota entre os agentes ocorre através do envio e recebimento de mensagens ao longo de um caminho, que inicia no remetente e termina no destinatário [121]. O modelo de distribuição segue as características de uma arquitetura orientada a serviços, e se fundamenta em uma estrutura fracamente acoplada. Todas as mensagens remotas são direcionadas para

um único servidor, que mantém uma lista de todos os containers registrados e os recursos existentes na plataforma. Assim, o container que solicita um serviço não precisa conhecer o endereço nem o nome do recipiente provedor do serviço.

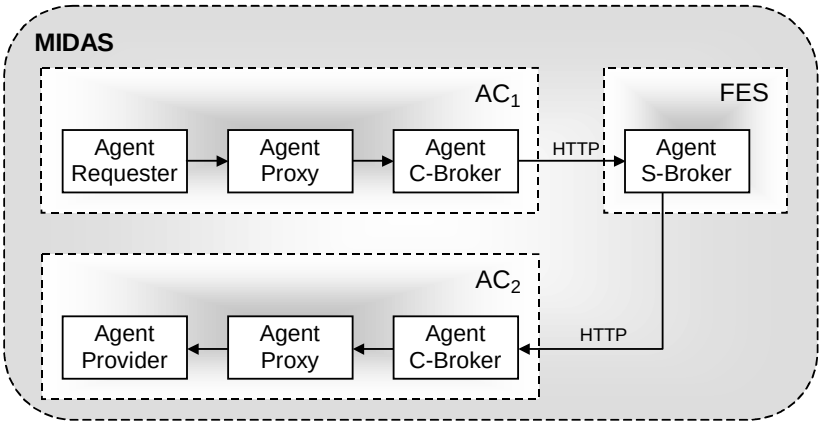

Figura 24 - Caminho da mensagem de uma requisição intra-plataforma

Quando um agente localizado no container AC_1 requisita um serviço remoto de outro agente localizado em um container remoto, o agente requisitante (*Requester*) envia a solicitação para o seu agente Proxy. O Proxy processa o serviço, verificando se ele pode ser atendido localmente. Se não puder (como é o caso da Figura 24), ele encaminha a mensagem para o agente C-Broker, que a direciona para o servidor. No servidor (FES), a mensagem é recebida pelo agente S-Broker, que identifica o container provedor do serviço (AC_2) e encaminha a mensagem. A mensagem percorre o caminho agenteCliente→proxy→cbroker→ sbroker→cbroker→proxy→agenteProvedor até chegar no provedor. O caminho entre um agente requisitante e um agente provedor é traduzido no MOM e no SOM. No MOM ele é representado pelos agentes C-Broker e S-Broker, e no SOM pelo agente *Proxy*. Então, o SOM é também um modelo orientado a mensagens [Zhao2004], já que o *Proxy* participa do *path* da mensagem.

A terceira modalidade de comunicação (intra-container) ocorre entre agentes localizados no mesmo container, como mostra a Figura 25. Quando localizados em um mesmo container, os agentes podem utilizar duas modalidades de comunicação: enviar mensagens de requisição de serviços e se comunicar através do *blackboard*. No primeiro caso, a comunicação é anônima. Quando um agente solicita um serviço, ele não sabe se este serviço é provido localmente ou se é remoto. As requisições são direcionadas para o agente Proxy, que se encarrega de

efetuar a verificação. Neste caso, o caminho da mensagem é AgentRequester→ Proxy→AgentProvider, que trafega internamente utilizando o protocolo interno MIP (Midas Internal Protocol) do MIDAS.

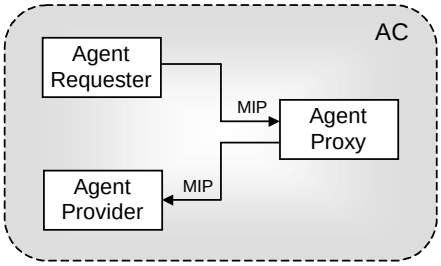

Figura 25 - Comunicação intra-container

Quando os agentes desejam efetuar comunicação ponto-a-ponto, eles podem utilizar o *blackboard*. O *blackboard* possibilita que os agentes possam enviar mensagens de forma assíncrona uns para os outros, simulando um modelo de comunicação ponto-a-ponto, ou envio de mensagens do tipo *broadcast*. Conforme descrito na seção 3.2.1.5, o *blackboard* provê uma interface que pode funcionar como um sensor, capturando e notificando as mudanças que ocorrem no ambiente.

3.4 O Modelo Estrutural

O modelo estrutural foi construído tomando por base as responsabilidades descritas na fase anterior, definidas pelos modelos de papéis. Ele foi projetado considerando a topologia de distribuição e as diferenças de configuração existentes nos containers e servidor (Seção 3.1). Nesta seção, são apresentadas as estruturas dos containers e do servidor.

3.4.1 A Arquitetura do Agent Container

A arquitetura do AC é composta por duas estruturas básicas: uma representada pelos agentes intermediários e outra representada pelos agentes de aplicações. Os agentes intermediários podem ser vistos como subsistemas colaborativos, que atuam de forma dinâmica e pró-ativa. A Figura 26 mostra a estrutura de um AC. As classes que representam as estruturas dos agentes

intermediários e as classes abstratas e agentes de aplicações estão organizadas em pacotes lógicos. Os relacionamentos resultam da colaboração entre os agentes, conforme descrito no modelo de papéis.

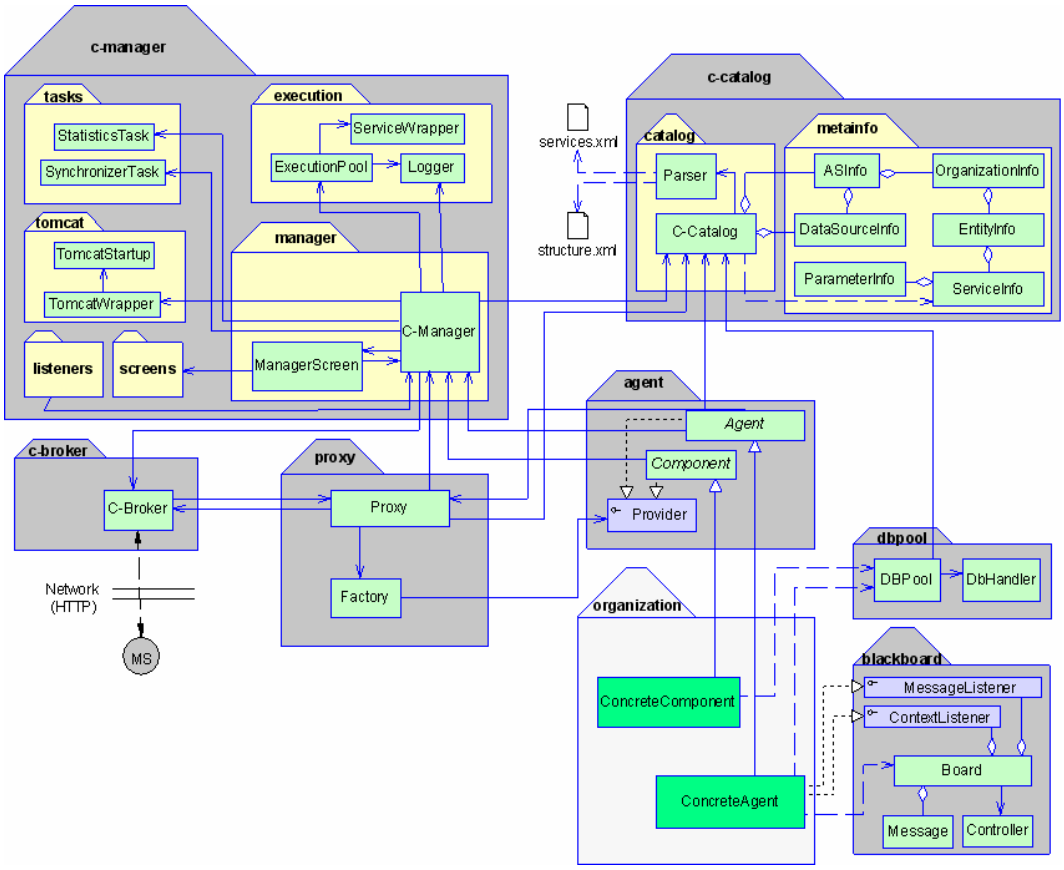

Figura 26 - Arquitetura do framework do Agent Container (AC)

A estrutura composta pelos agentes intermediários e pelas classes abstratas caracterizam uma parte *frozen* do *framework*, que permanece inalterada para as aplicações. As partes que variam a cada aplicação estão representadas pelas classes ConcreteAgent e ConcreteComponent, localizadas no pacote *organisation*, que representam a implementação concreta dos agentes e dos componentes. Os agentes intermediários Blackboard e DBPool participam do Agent Model e fornecem suporte ao projeto detalhado e *workflow* dos agentes.

O caminho básico de uma mensagem remota que chega a um AS é C-Broker→Proxy→Agent. No caminho, a mensagem é interceptada pelo agente C-Manager, que a coloca em um *pool* de execução. O agente C-Catalog também colabora no caminho, fornecendo ao Proxy o nome da entidade provedora do serviço. As mensagens locais transitam apenas entre os agentes locais e o Proxy.

3.4.1.1 O Agente C-Broker

O agente C-Broker é um *servlet*, cuja responsabilidade é manipular mensagens remotas que chegam ao container. Ele possui interfaces que servem como porta de entrada para requisições HTTP. A Figura 27 ilustra a estrutura do C-Broker, e os principais métodos implementados. Ele colabora com o agente C-Manager, atendendo mensagens para verificação de sincronismo e verificação de disponibilidade de serviços no servidor. Colabora também com o agente Proxy, nos procedimentos de enviar/receber mensagens.

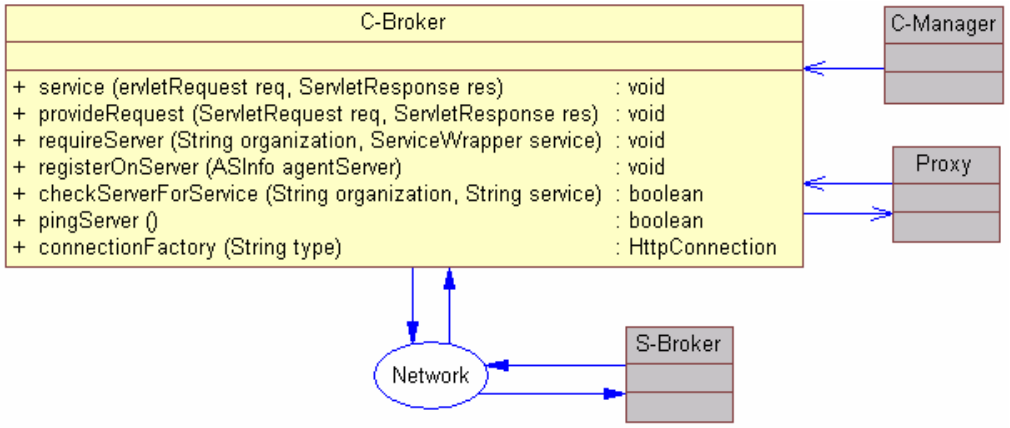

Figura 27 - A estrutura do agente C-Broker

O método *service* é invocado quando uma requisição de serviço é feita para um container AC. Ao receber uma mensagem, ele verifica qual o seu tipo, a fim de encaminhá-la a um dos seus métodos específico para o seu tratamento.

O método *provide* é invocado para atender requisições de serviço. Ele "desempacota" a requisição e solicita colaboração do agente Proxy para processar a requisição. Ele também se encarrega de empacotar a resposta da requisição para devolver o resultado do processamento do serviço, ou para devolver uma exceção.

O método *require* é invocado para re-direcionar um pedido de serviço para o servidor, a fim de alcançar um provedor de outro container. Ele empacota a requisição de serviço, envia para o servidor e procura obter a resposta para armazena-la na lista de saída.

O método *registerOnServer* é invocado quando o container deseja se registrar no servidor. Ele cria uma requisição de registro para o servidor e passa como parâmetro o objeto ASInfo, que contém a estrutura de recursos do container.

O método *checkServerForService* é invocado quando o container deseja verificar a disponibilidade de um serviço no servidor. Ele cria uma requisição de verificação para o servidor, informando a organização e o serviço desejado, retornado um *boolean*.

O método *pingServer* é invocado quando o container deseja interrogar o estado do servidor. Ele cria uma requisição de "*ping*" para o servidor e espera que a resposta da requisição seja a string "*pong*". Ele retorna verdadeiro caso isto aconteça e falso em qualquer outra condição.

O método *connectionFactory* efetua os procedimentos de empacotamento e desempacotamento de requisições. Ele manipula o objeto de requisição, encapsulando os parâmetros de entrada e saída pelo tipo e faz o caminho inverso, desempacotando as requisições para o agente Proxy.

3.4.1.2 O Agente Proxy

O agente Proxy atua como um representante da entidade provedora do serviço e é responsável pelo processamento das requisições de serviço. Este método desempenha um papel fundamental porque funciona como uma ponte entre os agentes e a camada de apresentação ou a camada de dados, reduzindo a complexidade do código e evitando a implementação de classes adicionais para mediar entre as duas camadas. A Figura 28 mostra a estrutura do agente Proxy, que é composta por duas classes: Proxy e Factory.

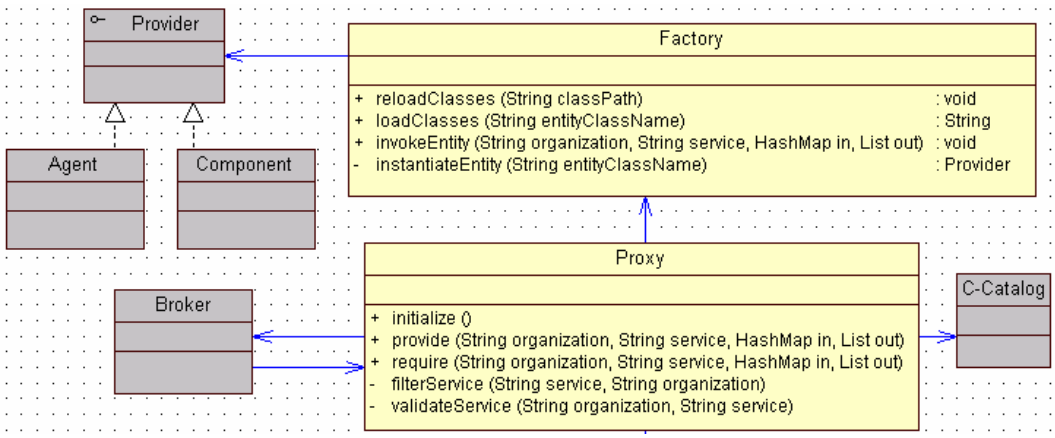

Figura 28 - A estrutura do agente Proxy

A classe Proxy é responsável pelo recebimento/envio e processamento de requisições dos agentes. Ela conta com a colaboração do agente C-Catalog para recuperar nome e localização de entidades provedoras de serviços. Na classe Proxy, o método *provide* é invocado pelo agente C-Broker quando uma requisição de serviços chega do servidor. O agente valida o serviço através dos métodos *validateService* e *filterService*, que verificam se a requisição é válida e se existe alguma regra que impeça o uso de serviço. O método *require* é invocado quando um agente ou componente dispara uma requisição de serviço. Ele verifica se o serviço desejado está disponível no próprio container ou se é um serviço remoto.

A classe Factory cumpre importante papel para tornar a arquitetura flexível e adaptável. Ela é responsável pelas atividades de re-configuração e criação dinâmica de instâncias. Os procedimentos de re-configuração possibilitam que uma classe executando na JVM possa ser substituída em tempo de execução. A criação dinâmica de instâncias é efetuada com a utilização do padrão de criação Factory Method [48], que abstrai o processo de criação de instâncias. Assim, novos agentes podem ser inseridos dinamicamente na plataforma sem precisar alterar o código da classe fábrica.

O método *reloadClasses* é invocado para efetuar a re-configuração do container. Ele utiliza a biblioteca *java.net.URLClassLoader* que provê um objeto do tipo URLClassLoader. O objeto possibilita atualizar os códigos que estão executando na JVM. O método *loadClasses* é invocado pelo método *reloadClasses* para efetuar o carregamento dinâmico das classes. O método *invokeEntity* é utilizado pelo agente Proxy para a efetivação de uma requisição de serviço. Ele invoca o método *instantiateEntity* para obter a interface provedora do agente, efetuar a instanciação dinâmica e chamar a execução do serviço.

3.4.1.3 O Agente C-Catalog

O agente C-Catalog é responsável pela manipulação dos recursos no container, e para isto ele possui um conjunto de responsabilidades relacionadas com a manutenção das descrições e das representações dos recursos. Para desempenhar estas responsabilidades, ele possui várias classes, que estão distribuídas em dois pacotes: *catalog* e *metainfo* (Figura 26).

A Figura 29 mostra uma visão mais detalhada das classes e colaborações que compõem a estrutura do agente C-Catalog. A classe C-Catalog funciona como uma porta de entrada, capaz de atender requisições de serviços encaminhadas pelo agentes Proxy, C-Manager e pelos agentes de aplicações. Considerando a quantidade de métodos, serão comentados apenas alguns. Grande parte deles pode ser entendida através dos nomes, auto-explicativos na maioria dos casos.

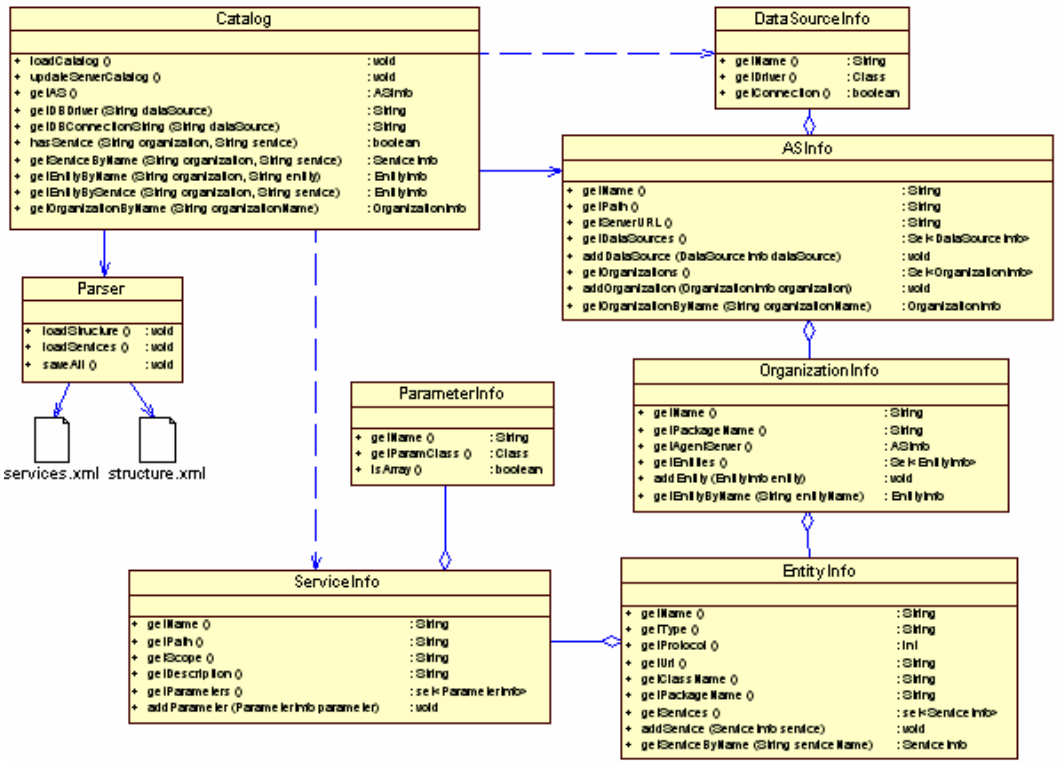

Figura 29 - Estrutura do agente C-Catalog

Na classe C-Catalog, o método *loadCatalog* é invocado pelo agente C-Manager quando a arquitetura é iniciada. Neste procedimento, com a colaboração da classe Parser ele recupera as informações dos arquivos XML para compor as estruturas de dados a serem mantidas em memória. O método *updateServerCatalog* é invocado sempre que acontece algum registro ou alteração em algum recurso. O método *getIAS* é invocado pelo agente C-Manager ou por agentes de aplicações, e retorna um objeto do tipo ASInfo que encapsula toda a estrutura de recursos. Os métodos *hasService* e *getEntityByName* são invocados pelo agente Proxy quando ele deseja saber se um serviço está disponível em um

container ou recuperar o nome de alguma entidade. Os demais métodos fornecem serviços similares, auto-compreensíveis.

A classe Parser é responsável pela manipulação da estrutura XML e pela manutenção das representações dos recursos em memória. Ela utiliza o utilitário DOM (Document Object Model) [107] para manipular as descrições XML, que são mantidas em dois arquivos XML: *structure.xml* e *services.xml*. A classe Parser executa os procedimentos para manter as descrições XML consistentes com as informações contidas nas estruturas mantidas em memória. Os serviços de persistência das informações XML são executados por uma intra-classe Generator.

A classe ASInfo é a raiz de uma hierarquia de classes agregadas que representam a estrutura de recursos do container. As classes encapsulam dados sobre os elementos da estrutura, e possuem um conjunto de métodos para obter informações sobre qualquer elemento. A estrutura provê duas modalidades de acesso: pela raiz da estrutura ou acesso direto ao serviço. A primeira modalidade pode ser utilizada em tarefas de descoberta de serviços. A segunda modalidade garante eficiência na medida que informações rápidas sobre serviços podem ser obtidas utilizando o serviço como índice sem precisar percorrer a estrutura.

3.4.1.4 O Agente C-Manager

O agente C-Manager é o mais complexo da arquitetura. Ele colabora com todos os outros agentes do container e as suas responsabilidades estão distribuídas em seis pacotes: *manager*, *tomcat*, *screens*, *listeners, tasks* e *execution* (ver Figura 26). A Figura 30 mostra um diagrama de classes que detalha a estrutura interna do agente C-Manager. Os relacionamentos com os outros agentes intermediários da arquitetura (conforme apresentado na Figura 26) foram suprimidos neste diagrama, para facilitar a visualização.

A classe C-Manager funciona como uma porta de entrada e saída, por onde ele recebe/envia requisições e colabora com os demais agentes da arquitetura. A classe C-Manager também é responsável pela inicialização do servidor de rede, dos agentes da arquitetura e dos agentes que participam das aplicações. Ela colabora com os agentes de aplicações, fornecendo *wrappers* de requisição.

Colabora também com o agente C-Catalog enviando especificações de recursos efetuadas nas interfaces gráficas. O C-Manager conta com a colaboração do agente C-Broker para encaminhar requisições de verificação de sincronismo, e colabora com o agente Proxy, atendendo requisições para enfileiramento de mensagens.

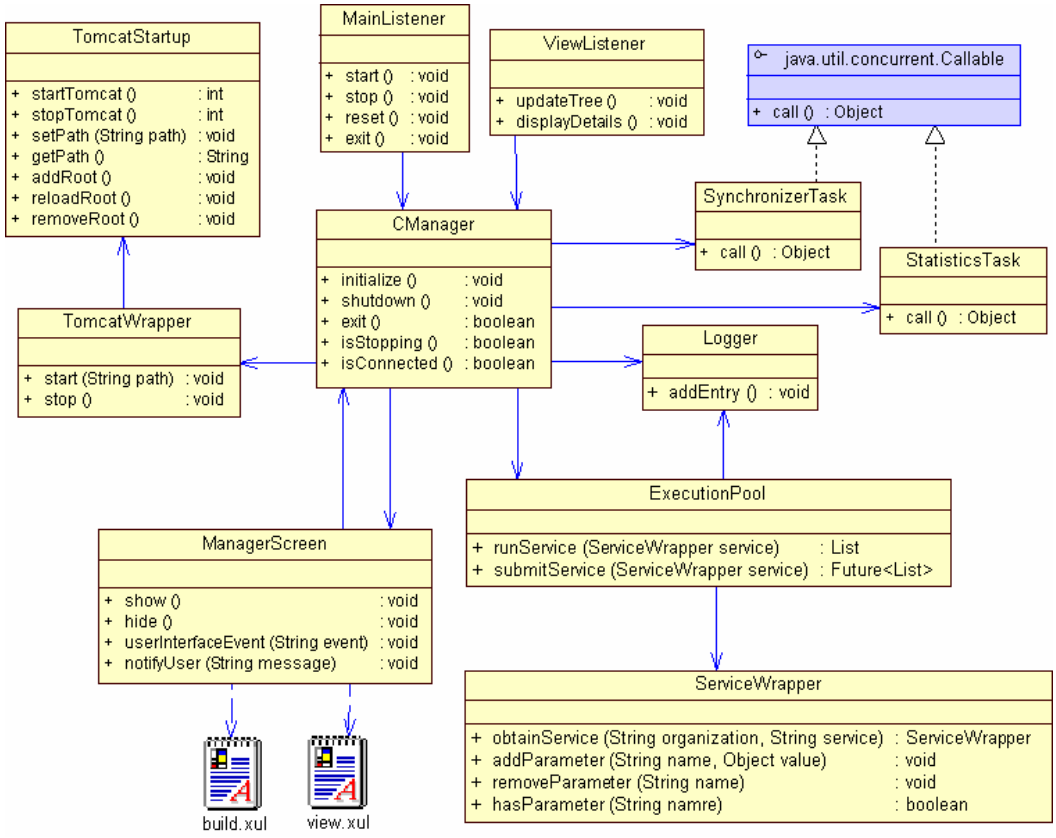

Figura 30 - Estrutura do agente C-Manager

A plataforma utiliza o Apache Tomcat embarcado para tornar transparente as atividades de configuração e conexão de rede. Quando o container entra em execução, a classe C-Manager levanta o servidor de rede invocando o método *start* da classe TomcatWrapper, que direciona a chamada para a classe TomcatStartup iniciar o servidor. Outros métodos estáticos da classe TomcatWrapper são oferecidos para o gerenciamento do ciclo de vida e registro da localização do servidor de rede.

A classe *SynchronizedTask* executa tarefas contínuas de verificação de sincronismo, checando continuamente os containers conectados e sincronizados com a plataforma. A classe *StatisticsTasks* tem por função atualizar

continuamente informações que são exibidas nas interfaces gráficas, tais como utilização de memória e número de *threads* ativos. Ela também coleta métricas e dados para estatísticas, tais como tempo de resposta das conexões, exceções ocorridas e falhas no sistema.

O objetivo da camada *execution*, formada pelas classes ServiceWrapper, ExecutionPool e Logger (ver Figuras 26/30) é permitir a realização de operações mais intuitivas e flexíveis no processo de construir uma requisição de serviço. A camada possui três classes em sua estrutura, descritas a seguir.

- Classe ServiceWrapper: possibilita a realização de operações mais intuitivas e flexíveis para construir uma requisição de serviço; ela possui métodos tais como *addParameter* e *removeParameter* através dos quais o agente pode adicionar parâmetros a uma requisição sem se preocupar com a sua ordem ou localização ou *casting* para a conversão de tipos, já que cada atributo representa um tipo.
- Classe Logger: implementa um arquivo *log* que é utilizado para armazenar informações sobre o processamento das requisições e estatísticas, possibilitando verificar se as requisições de serviço foram atendidas com sucesso, que erros ocorreram, tipo de erro, etc.
- Classe ExecutionPool: controla o *pool* de execução de chamadas síncronas e assíncronas, e para isto disponibiliza dois métodos: *runService*, utilizado para efetuar chamadas síncronas, e *submitService*, utilizado para chamadas assíncronas. O método *submitService* retorna um objeto do tipo *Future<List>* da API *java.lang.concurrency* de Java 5. Este objeto é uma conhecida entidade da programação concorrente, que representa um futuro que pode ou não acontecer.

A camada *execution* facilita o projeto detalhado dos agentes. Ela oferece um objeto *wrapper* de requisição, que facilita a manipulação de parâmetros durante a montagem da requisição, provendo transparência total e simplicidade no processo de requisitar serviços. Ela oferece duas modalidades de chamada: síncrona e assíncrona. O diagrama de seqüência apresentado na Figura 31 mostra as ações executas durante uma chamada síncrona. O primeiro passo é obter uma requisição de serviço através da interface requerente provida pela classe abstrata Agent. A

interface invoca o método *obtainService* da classe C-Manager, que cria e retorna um objeto do tipo *ServiceWrapper*. De posse deste objeto, o agente pode adicionar quaisquer parâmetros desejados e efetuar uma chamada síncrona utilizando o método *run*.

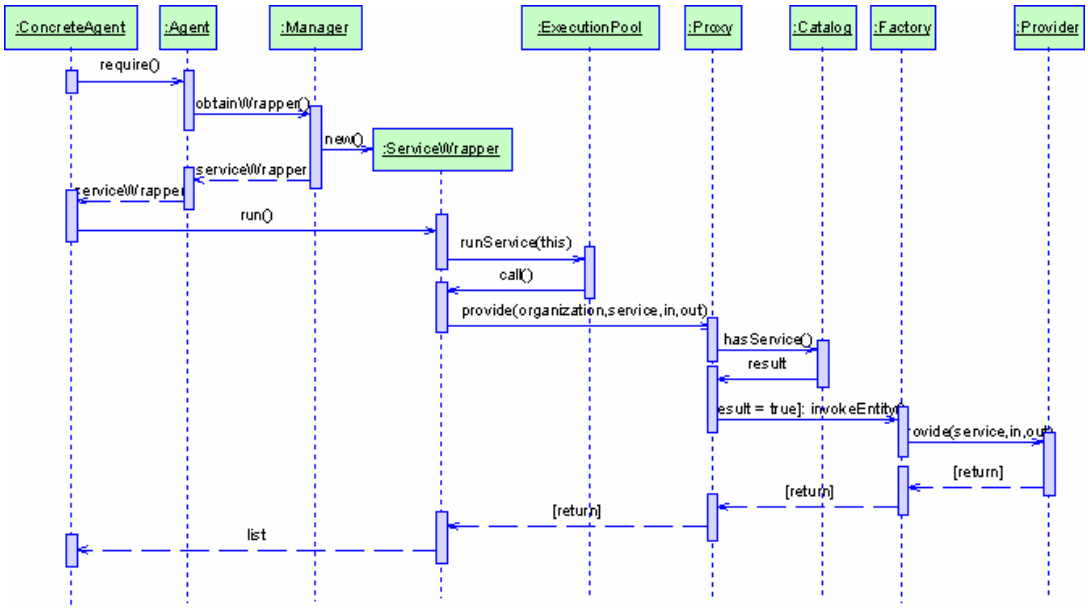

Figura 31 - Diagrama de seqüência para uma chamada local síncrona

Quando o método *run* é invocado, a requisição é repassada ao Proxy, que efetua o seu processamento. Durante o processamento, o agente Proxy conta com a colaboração dos agentes C-Catalog, que provê o nome da entidade provedora do serviço e com a colaboração da classe Factory, que cria dinamicamente a instância da entidade provedora e invoca a sua interface *provide*.

O diagrama de seqüência na Figura 32 ilustra o fluxo de comunicação entre o agente requisitante e o agente provedor numa chamada assíncrona. A chamada assíncrona é efetuada invocando o método *submit*, que imediatamente retorna um objeto do tipo Future<List>, criando uma nova *thread* para processar a requisição. Utilizando os tipos genéricos da nova API Java, é possível utilizar a parametrização <List> para indicar que este objeto Future retornará objetos do tipo List. Isto acaba com a necessidade de *casting* e evita erros de programação com trocas de tipo em tempo de compilação.

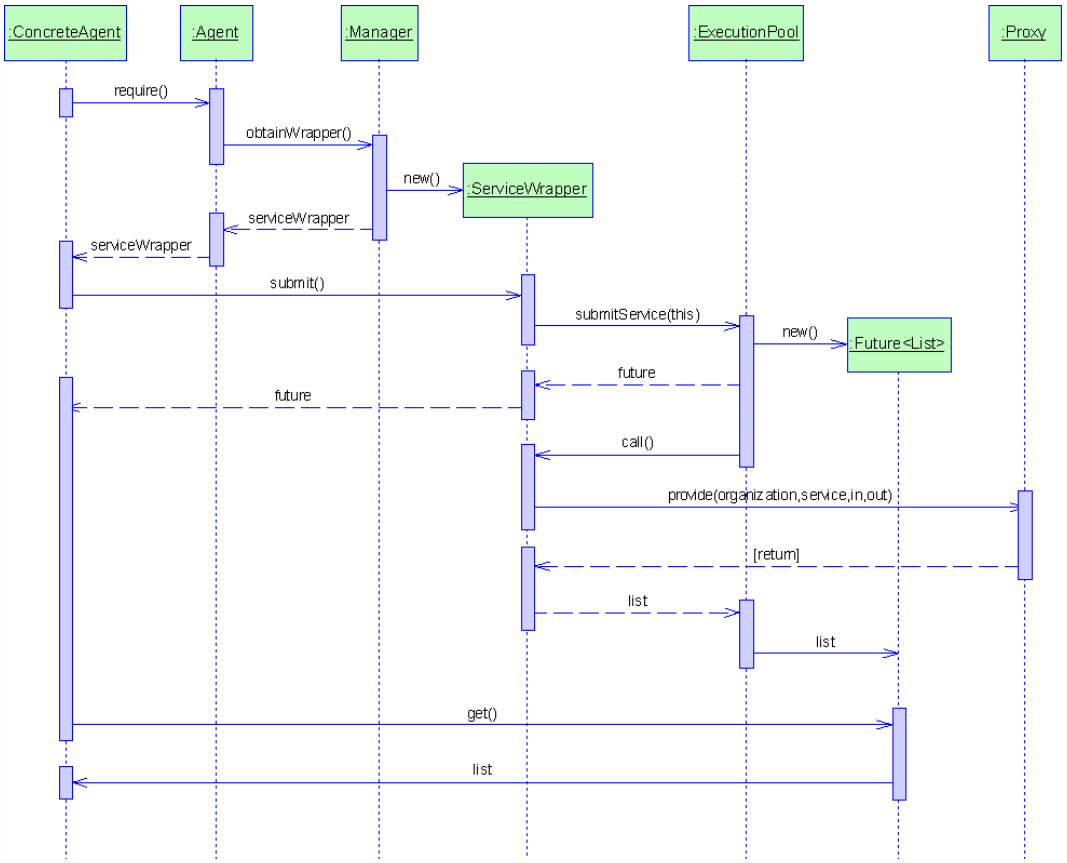

Figura 32 - Diagrama de seqüência para uma chamada local assíncrona

Os serviços de gerenciamento das interfaces gráficas que fazem parte da API do MIDAS são efetuados pela classe ManagerScreen. As especificações das GUI's são armazenados em formato XML nos arquivos *view.xul* e *build.xul*. A grande vantagem desta técnica é que as interfaces podem evoluir apenas alterando o arquivo XML sem necessidade de modificar o código. As interfaces gráficas que foram construídas utilizando o *toolkit* Thinlet.

Os assistentes GUI's oferecem duas perspectivas de visão: (i) a perspectiva *View*, utilizada nos containers oferece uma visão dinâmica do ambiente de execução e mecanismos para manipular os eventos de gerenciamento do ciclo de vida; e (ii) a perspectiva *Build* provê uma interface gráfica para o registro de agentes e recursos. A perspectiva de visão global da plataforma pode ser obtida do servidor de *front-end*. A Figura 33 mostra a perspectiva *View*, utilizada para visualização dos recursos e gerenciamento do ciclo de vida.

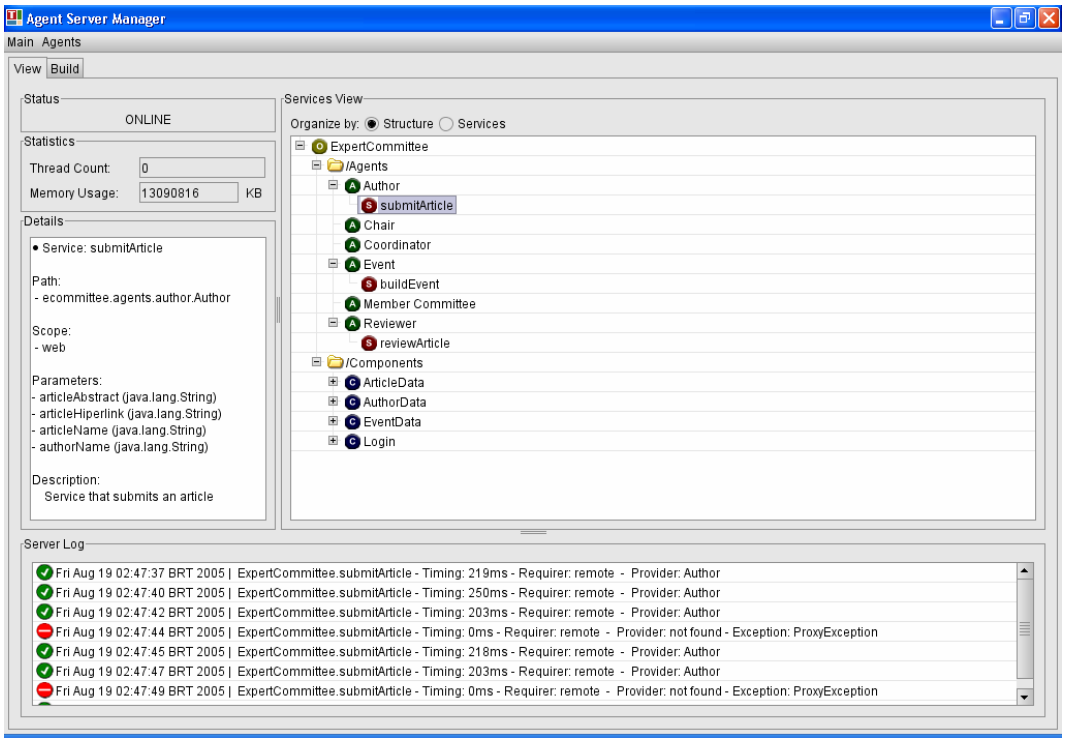

Figura 33 - Assistente GUI para a perspectiva View

O painel do lado direito mostra a estrutura de recursos do container. A estrutura é navegável contém uma hierarquia de entidades que compõem a estrutura de recursos do container. Ao navegar a estrutura, é possível visualizar os detalhes da entidade selecionada no painel *Detais* do lado esquerdo. Os painéis que aparecem do lado superior esquerdo exibem dados sobre o estado do servidor, os *threads* em execução e o espaço de memória alocado. Note que o assistente apresenta um ícone diferenciado para as transações com ocorrência de falhas. Na parte inferior da janela, o painel *ServerLog* mostra detalhes de todas as transações sendo executadas no servidor.

A segunda perspectiva, denominada Build é utilizada para a especificação, visualização e registro de recursos. A Figura 34 mostra a interface gráfica para a perspectiva de construção. O painel *Structure* do lado esquerdo mostra a hierarquia de recursos, identificando as organizações, agentes e componentes que fazem parte da estrutura. A árvore é navegável, e para cada entidade selecionada, são apresentados os detalhes nos *frames* localizados no lado direito da janela.

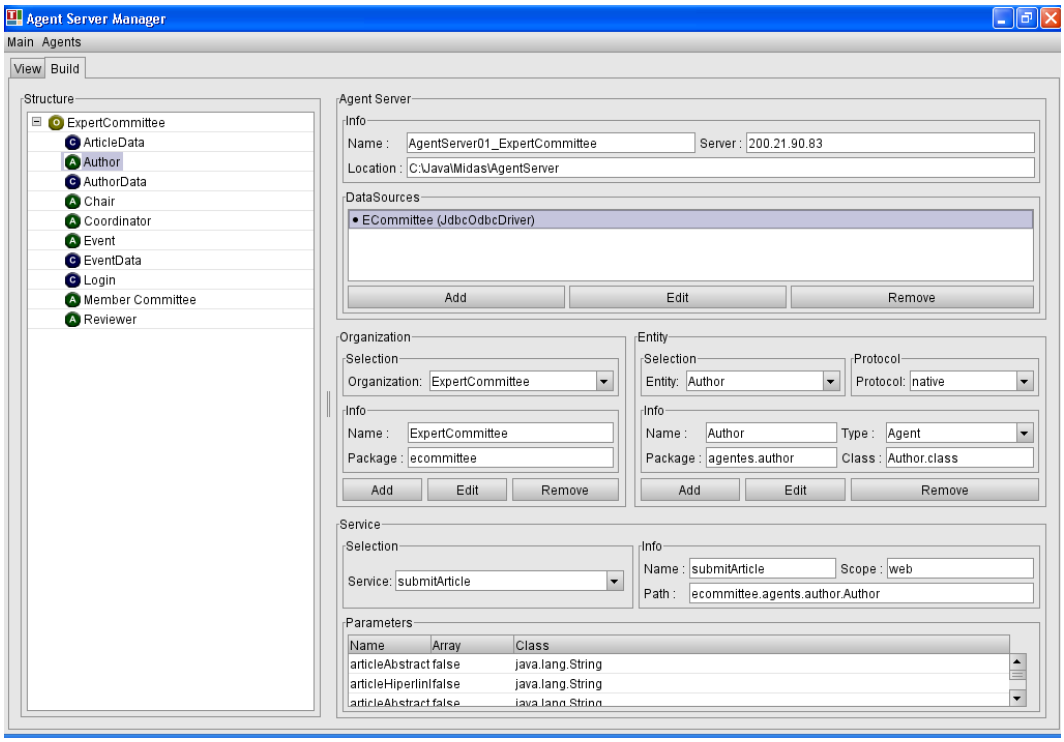

Figura 34 - Assistente GUI para a perspectiva Build

Os *frames* possibilitam efetuar operações de manutenção, utilizando os botões *Add*, *Edit* e *Remove*. No *frame Info* são informados metadados do container, tais como o seu nome, e o IP do servidor. No painel *DataSources* podem ser visualizados e configurados *drivers* de acesso a banco de dados; para incluir um novo *driver* basta acionar o botão do lado esquerdo do *mouse* no painel. Nos *frames* localizados abaixo deste painel, podem ser capturadas ou visualizadas especificações das entidades (organizações, agentes e componentes), serviços e parâmetros.

Os eventos que ocorrem nas interfaces gráficas são capturados pelas classes ouvintes, localizadas no pacote *listeners*: (i) ViewListener, que monitora os eventos de manutenção do ciclo de vida dos agentes e (ii) BuildListener, que monitora as inclusões e atualizações efetuadas na estrutura de recursos.

3.4.1.5 O Agent Model

O Agent Model (AM) define uma estrutura de suporte ao projeto detalhado dos agentes, composta pelas classes abstratas Agent e Component e pelos agentes

intermediários Blackboard e DBPool. A classe abstrata Agent provê as interfaces padrão por onde os agentes podem interagir com o ambiente, o procedimento que dispara o *thread* de execução do agente e as assinaturas para gerenciamento do ciclo de vida. A Figura 35 mostra o diagrama de classes do AM.

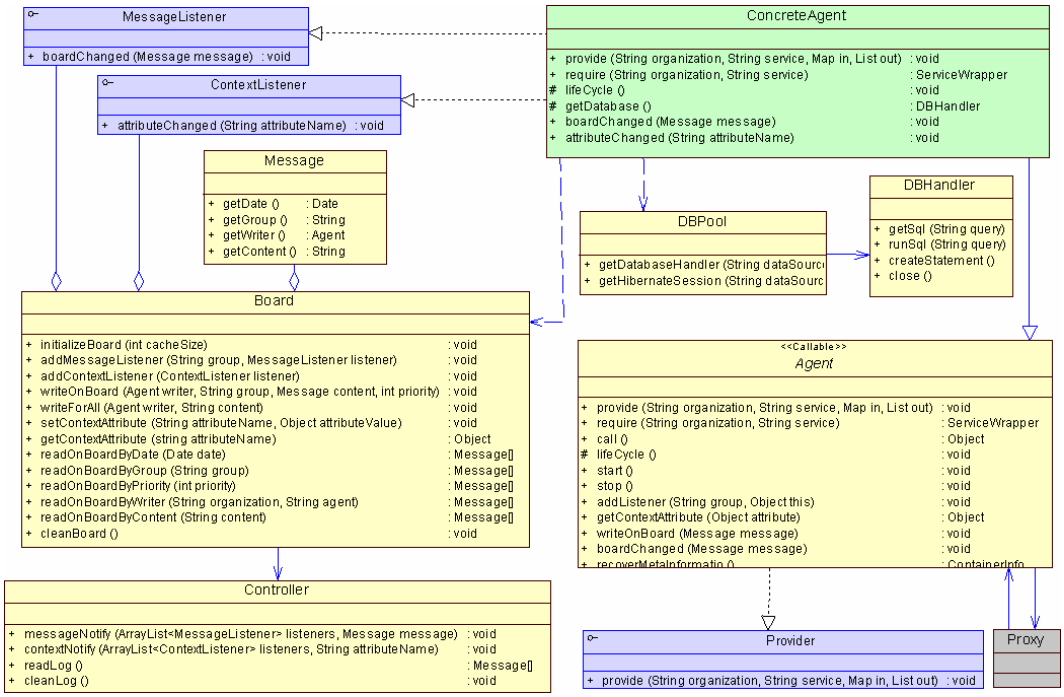

Figura 35 - Diagrama de classes do Agent Model

A classe abstrata Agent implementa a interface Provide, que define uma assinatura básica *provide* por onde os agentes recebem as requisições de serviços encaminhadas pelo Proxy. A interface fornece um dos requisitos básicos para a criação dinâmica de agentes, possibilitando obter famílias de agentes e componentes que respondem às requisições nesta interface abstrata. O método *require* representa a interface requerente do agente, onde ele pode obter os objetos *wrapper* de requisição e invocar serviços externos de forma transparente. Agent implementa também a interface Callable (melhoria da Runnable) que possibilita criar *threads* utilizando a nova API de concorrência de Java 5. O método *call* cria a *thread* de execução e invoca o método *lifeCycle* que dispara o ciclo de vida do agente.

A estrutura do agente Blackboard é composta pelas classes Board, Controller, Message e pelas interfaces MessageListener e ContextListener. Os agentes concretos devem implementar estas interfaces para ter acesso ao

Blackboard. A interface MessageListener define a assinatura *boardChanged*, utilizada para notificar os agentes sobre mensagens de interesse. A interface ContextListener define a assinatura *attributeChanged*, invocada pela classe Board para notificar os agentes sobre mudanças na estrutura de variáveis globais. A classe Control mantém a lista de ouvintes e grupos registrados. Ela provê os métodos *messageNotify* e *contextNotify* que são invocados pela classe Board para notificar os agentes sobre mensagens ou mudanças de variáveis globais.

O agente DBPool colabora com os agentes de aplicação provendo serviços para facilitar a manipulação de informações em banco de dados. As suas responsabilidades estão distribuídas em duas classes: DBPool e DBHandler. A classe DBPool provê os métodos *getDatabaseHandler*, para recuperar um objeto de conexão com banco de dados (especificado em XML) e *getHibernateSession*, que fornece um objeto de conexão com o *framework* Hibernate. A classe DBHandler encapsula um objeto de conexão com banco de dados. Utilizando o objeto DBHandler, é possível ao agente requisitar a criação de um *Statement* e utilizar os conhecidos métodos *getSq* e *setSql* para manipular dados. Ele provê um método *getFactory*, que cria um objeto do tipo *Session* para integração com o Hibernate.

A classe ConcreteAgent representa os agentes concretos instanciados nas aplicações. Ela implementa as interfaces MessageListener e ContextListener, para obter acesso a mecanismos providos pelo Blackboard.

3.4.2 A Arquitetura do Servidor de Front-End

O servidor de *front-end* (FES) tem quatro responsabilidades principais: (i) receber e direcionar requisições remotas de serviços; (ii) gerenciar o ciclo de vida da plataforma e manter a lista de containers registrados; (iii) manter uma representação consolidada da estrutura de recursos da plataforma e (iv) prover os mecanismos necessários para a integração com serviços Web. Para desempenhar estas responsabilidades ele utiliza três agentes intermediários: S-Broker, S-Catalog e S-Manager. No servidor não existem agentes de aplicações nem serviços, apenas representações de recursos. A Figura 36 mostra um diagrama de pacotes as classes e os relacionamentos que compõem a estrutura do servidor.

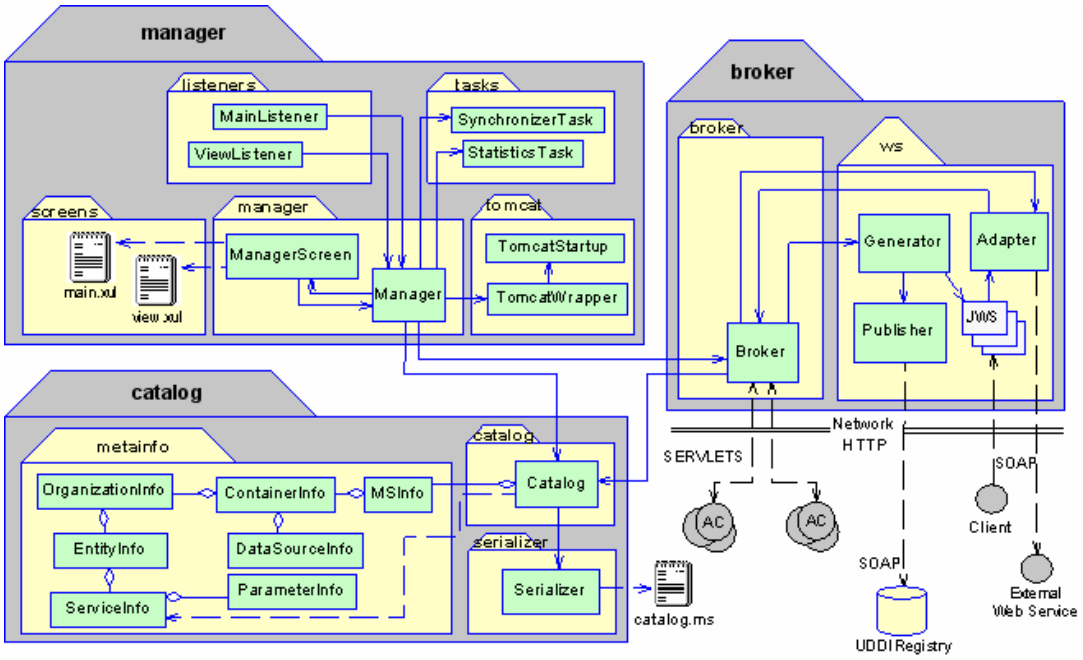

Figura 36 - Arquitetura do framework do servidor de front-end

Os agentes S-Manager e S-Catalog têm responsabilidades similares àquelas desempenhadas em um container AC, mas possuem algumas diferenças. No modelo estrutural, o agente S-Catalog possui as seguintes diferenças em relação ao C-Catalog:

- no servidor, ele não manipula descrições XML, como acontece em um container; ao invés disso, ele persiste as informações em um arquivo serializado, mantido pela classe Serializer;
- no servidor, ele trabalha e mantém em memória as representações consolidada de recursos, ao invés das representações locais.

O agente S-Manager também possui diferenças em relação ao agente C-Manager localizado no container. A principal é que no servidor ele não possui o pacote *execution*, responsável pelo fornecimento dos objetos *wrapper* de requisição, já que este procedimento é utilizado apenas por agentes de aplicações.

O agente mais diferenciado do servidor é o S-Broker, que possui diferenças significativas em relação ao C-Broker localizado em um container. O S-Broker é responsável pelas regras de integração entre os containers da plataforma e pela interoperabilidade com aplicações externas. A Figura 37 mostra um diagrama de classes do agente S-Broker, onde podem ser vistos os principais métodos de cada classe.

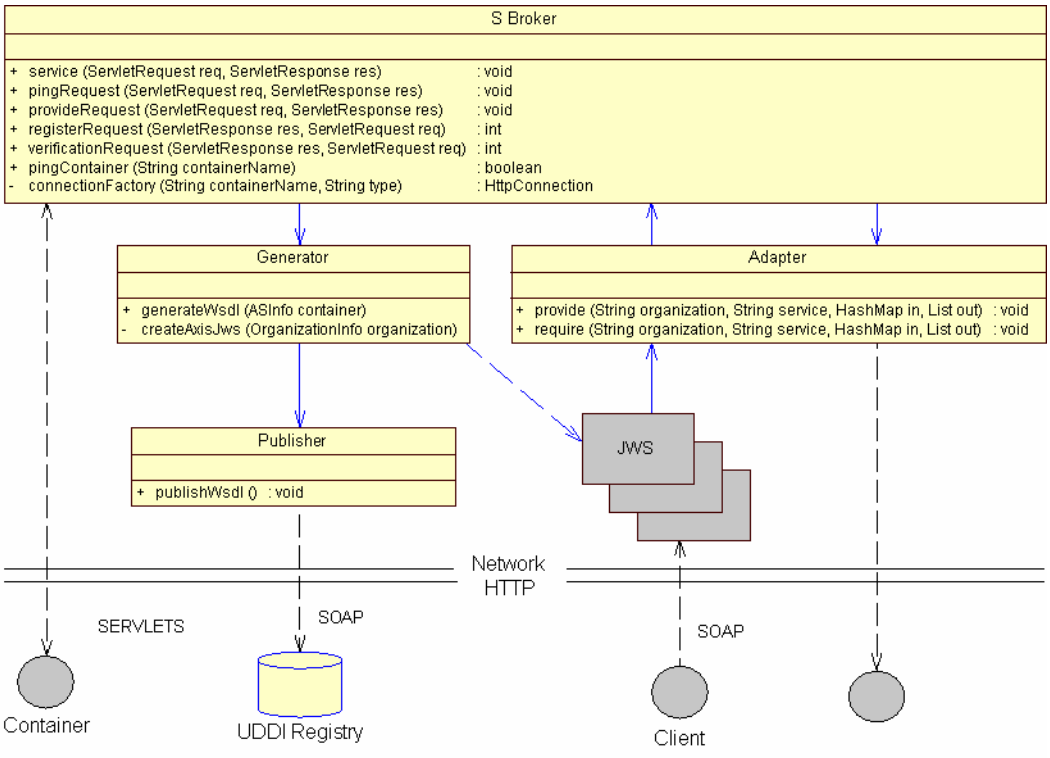

Figura 37 - Diagrama de classes do agente S-Broker

A classe S-Broker é um *servlet*, que pode receber requisições por uma porta HTTP. Quando uma requisição HTTP é enviada ao servidor, a classe S-Broker avalia o tipo da requisição antes de encaminhá-la a um método específico para o seu tratamento. Conforme definido no modelo de papéis, ele pode receber três tipos de requisição: solicitação de serviço, verificação de sincronismo e requisições de registro. Os métodos que implementam estas requisições são similares àqueles descritos para o agente C-Broker, na seção 3.4.1.1.

O agente S-Broker possui também uma porta SOAP, localizada na classe Adapter para receber requisições de serviços Web. A classe *Adapter* é responsável pela transformação de mensagens SOAP para o protocolo MIDAS e do protocolo MIDAS para SOAP. Para a primeira finalidade, ela possui o método *provide*, que é invocado pelos arquivos JWS. Isto permite que a classe Adapter possa repassar as chamadas transformadas para a classe S-Broker, e conseqüentemente, para um container provedor. No sentido contrário, a requisição nativa é adaptada para uma requisição SOAP e despachada ao provedor do serviço Web.

As classes Generator, Publisher e Adapter fazem parte do pacote *ws*, e desempenham as funções de integração da plataforma com serviços Web e a

automatização das atividades de geração das especificações WSDL e UDDI. A classe Generator é responsável pela geração das descrições WSDL para os serviços publicados como serviços Web na plataforma. Para criar uma descrição WSDL, ela utiliza a ferramenta Axis[1] para gerar os arquivos JWS (Java Web Services). O Axis fornece as descrições WSDL prontas para publicação e habilita estes arquivos JWS possam receber requisições SOAP. A classe Generator possui os seguintes métodos:

- *generateWSDL (ASInfo container)* – este método é invocado pela classe S-Broker quando um container se registra no servidor; ele recebe as informações sobre o container e recupera os objetos de informações das organizações, para poder gerar os arquivos do Axis.
- *createAxisJws (OrganizationInfo orgnization)* – este método é invocado pelo método *generateWSDL* para cada organização existente em um container; ele investiga os serviços da organização, e gera os arquivos JWS para receberem requisições HTTP.

A classe *Publisher* é responsável pela publicação das descrições WSDL em um registro UDDI. Para isto, ela possui o método *publishWsdl*, que utiliza o Axis para recuperar as descrições WSDL publicar os serviços no registro UDDI. As classes JWS são geradas pelo Axis e possuem a organização como nome e os serviços oferecidos como métodos. As requisições SOAP que chegam ao agente S-Broker são direcionadas para as classes JWS, como apresentado na Figura 37.

3.4.3 A Arquitetura do Component Container

O Component Container (CC) é um container mais leve do que o AC, que provê uma estrutura mínima para facilitar a integração entre a plataforma e aplicações de negócios e serviços Web. Como pode ser visto na Figura 38, a principal diferença entre um container AC e um container CC é que no CC não existem agentes de aplicações, apenas componentes. Desta forma, ele não possui a classe abstrata Agent e o Blakboard, que são para uso exclusivo de agentes. Os

[1] O Axis é uma ferramenta da Apache para que gera arquivos WSDL para serviços Web.

demais agentes intermediários do CC desempenham papéis e responsabilidades idênticas às dos agentes intermediários localizados em um AC.

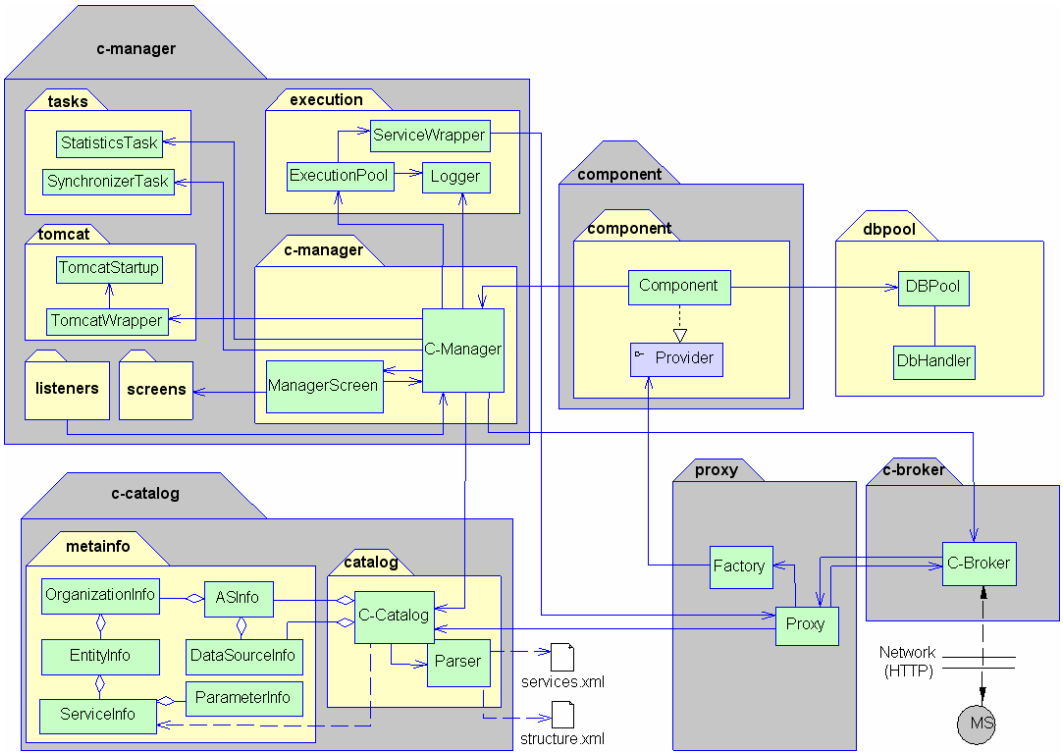

Figura 38 - Arquitetura do framework do Component Container

A idéia básica do container CC é a utilização de objetos *wrapper*. Os *wrappers* podem encapsular objetos de acesso a dados, funcionalidades de aplicações de negócios ou funcionalidades de sistemas legados. A vantagem desta abordagem é que ela promove simplicidade conceitual e transparência de linguagem. Os componentes podem acessar aplicações heterogêneas encaplusando objetos CORBA ou efetuando chamadas com a interface Java JNI. Eles também podem ser utilizados para encapsular serviços Web e formar composições de serviços. Os serviços encapsulados em um Component podem ser publicados de forma automática pela plataforma como serviços Web.

3.5 Discussão

De um modo geral, os modelos de referência são derivados de um estudo do domínio da aplicação. Contudo, eles não são normalmente um caminho para a

implementação. Em vez disso, um modelo de referência [101] fornece um vocabulário para comparação que atua como um padrão, em relação ao qual os sistemas podem ser avaliados. Eles representam uma arquitetura idealizada que inclui todas as características que o sistema pode incorporar.

A arquitetura de referência WSA estabelece um padrão construído sobre um conjunto de conceitos e propriedades para arquiteturas Web. A utilização do padrão WSA para projetar a arquitetura do MIDAS trouxe os seguintes benefícios:

- Possibilitou obter uma arquitetura flexível e extensível, aplicando os princípios de modularidade e baixo acoplamento, característicos de SOA.
- Facilitou a comunicação e o entendimento dos conceitos oferecendo abstrações, um vocabulário comum e padrões de avaliação.
- Tornou mais fácil o projeto e implementação, provendo blocos de construção de alto nível modulares e coesos.
- Facilitou a evolução da arquitetura porque a construção dos módulos em um modelo reutilizável de papéis.
- Tornou mais natural a integração da tecnologia de agentes com serviços Web e a interoperabilidade com aplicações externas.

As estruturas apresentadas nas seções 3.4.1, 3.4.2 e 3.4.3 resultaram do mapeamento dos conceitos definidos pela arquitetura WSA e da introdução de novos conceitos. Os novos conceitos estendem a especificação da arquitetura de referência introduzindo um modelo abstrato de agente e um *blackboard* para comunicação entre agentes. O modelo abstrato deve ser estendido pelos agentes concretos e provê os *hot-spots* onde desenvolvedores podem implementar as particularidades específicas das aplicações. A introdução do *blackboard* como um meio alternativo de comunicação supre uma deficiência apresentada por arquiteturas que utilizam o paradigma SOA: a falta de suporte para a comunicação ponto-a-ponto entre agentes.

O modelo de papéis (Seção 3.2) foi utilizado como base para projetar as estruturas. A partir das responsabilidades definidas pelos papéis, foram projetadas as estruturas dos agentes intermediários e do Agent Model. Os agentes intermediários executam serviços de infraestrutura, e separam totalmente o comportamento genérico da arquitetura do comportamento específico dos agentes.

A estrutura do AM, demonstrada nas seções 3.2.2 e 3.4.5.1, foi construída para dar suporte ao projeto detalhado dos agentes. O modelo abstrato Agent fatora em uma superclasse um conjunto de funções comuns a todos os agentes de aplicações. A classe abstrata Component funciona como um objeto wrapper, que pode ser usado para encapsular funcionalidades específicas do domínio. O agente Blackboard provê um poderoso meio para comunicação entre agentes, e disponibiliza uma interface que pode funcionar como um sensor para os agentes capturarem mudanças ambientais. O agente DBPool oferece facilidades para manipulação de dados em bases relacionais.

As propriedades de flexibilidade e capacidade de evoluir da arquitetura resultaram da adoção do princípio básico de baixo acoplamento, característico de arquiteturas orientadas a serviços e da utilização de padrões de projeto para projetar os módulos da estrutura. A capacidade de adaptação dos agentes da arquitetura e dos agentes que participam das aplicações pôde ser alcançada com os mecanismos de configuração dinâmica descritos na seção 3.2.1.2 e pelos padrões de criação Factory Method (Seção 3.4.1.2).

A plataforma provê interoperabilidade bi-direcional entre agentes e serviços Web e fácil integração com aplicações externas, conforme demonstrado na seção 3.4.2. O componente Adapter, que participa da estrutura do agente S-Broker, é capaz de converter requisições SOAP para o protocolo MIDAS, assim como converter requisições do protocolo MIDAS para SOAP. Todos os procedimentos para a geração das especificações WSDL e UDDI são automáticos.

O container de componentes, descrito na seção 3.4.3, facilita a integração entre a plataforma e aplicações empresariais e legadas. Ele provê uma estrutura baseada em componentes, oferecendo uma classe abstrata e uma interface que possibilita criar famílias de componentes para encapsular funções específicas do domínio, tais como objetos de acesso a dados ou processos de negócios. Os componentes podem ser utilizados também para efetuar composição e orquestração de serviços Web.