Pos-Graduação em
Ciências da Computação

Dissertação de Mestrado

Hermes – Um *Middleware* Orientado a Mensagem para Ambientes Corporativos

Por

Eduardo Gonçalves Calabria

Universidade Federal
De Pernambuco
posgraduacao@cin.ufpe.br
http:ftp.cin.ufpe.br
ftp://ftp.cin.ufpe.br/pub/posgrad

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EDUARDO GONÇALVES CALABRIA

"Hermes – Um *Midddleware* Orientado a Mensagem para Ambientes Corporativos"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR(A): Prof. Nelson Souto Rosa

RECIFE, MARÇO/2004

RESUMO

Atualmente, a maioria dos sistemas de informação corporativos utiliza uma infra-estrutura de comunicação, conhecida como *middleware*, para a troca de mensagens com outros sistemas. O *middleware* fornece um conjunto de serviços (ex., segurança, transação e eventos) atuando como uma interface para que a aplicação seja construída sem que o desenvolvedor tenha que tratar diretamente com a complexidade dos mecanismos de comunicação de baixo nível.

Os sistemas de *middleware* são normalmente categorizados de acordo com o tipo de primitiva fornecida para interação entre as aplicações: *middleware* procedural (chamada remota de procedimento), *middleware* orientado a mensagem (passagem de mensagem), *middleware* transacional (transação distribuída) e *middleware* orientado a objetos (invocação de método remoto). Dentre estas categorias, os sistemas de *middleware* orientado a mensagem (MOM) são os mais amplamente utilizados como infra-estrutura de comunicação de aplicações corporativas.

Os requisitos de troca de mensagens são cada vez mais sofisticados e complexos, exigindo que os MOMs utilizados atendam a requisitos como: aumento no volume de dados, concorrência, escalabilidade, disponibilidade, garantias de entrega das mensagens, controle de assincronismo, tolerância a falhas, balanceamento de carga e transparência de localização.

Neste contexto, este trabalho propõe um *middleware* orientado a mensagem chamado *Hermes*. O *Hermes* implementa todas as funcionalidades exigidas para um MOM, e incorpora características adicionais, otimizando algumas implementações relativas às escalabilidade e disponibilidade, e adicionando elementos funcionais que tornam seu uso mais fácil e abrangente. O *Hermes* implementa ainda o padrão JMS (Java *Message Service*), que propõe a implementação de um conjunto de interfaces e de características comuns a qualquer *middleware* orientado a mensagem.

Podem ser enumeradas as seguintes contribuições deste trabalho: a apresentação detalhada das características e das formas de implementação de um MOM, e a concepção de uma arquitetura modular e componentizada para o MOM.

Palavras-Chave: *Middleware*, MOM, Escalabilidade, Balanceamento de Carga, Transparência de Localização.

ABSTRACT

Currently, most of the corporative information systems use a communication infra-structure known as middleware, in order to exchange messages with other systems. The middleware provides a set of services (e.g., security, transaction and events), allowing application programmers to avoid dealing with the complexity of underlying communication mechanisms. Middleware systems are considerated interfaces or APIs, which encapsulates such mechanisms.

Middleware systems are commonly categorized according to the type of primitive provided to allow the interactions between applications: RPC middleware (remote procedure call), message oriented middleware (message exchange), transaction middleware (supports distribuited transactions) and oriented object middleware (remote method invocation). Among these categories, systems of message oriented middleware (MOM) are the most widely used to provide the communication infra-structure support for corporative applications.

Message exchange requirements have become increasingly sophisticated and complex, whereas demands from the MOMs the satisfaction of requirements such as: concurrency, scalability, availability, guaranty on delivering messages, asynchronous proccessing control, fault torelance, load balancing and location transparency.

This work proposes a message oriented middleware named *Hermes*, which implements all the required functionalities for a MOM. *Hermes* embodies additional features, improving some coding aspects related to scalability / availability, and adding functional elements, which make *Hermes* usage easier and more available to a wider set of applications. *Hermes* also implements JMS (Java Message Service) framework, which defines a set of interfaces and common MOM features.

The contributions of this work are defined as follows: detailed design and code features associated to MOMs, and the conception of a modular and componentized architecture for a MOM.

Keywords: Middleware, MOM, Scalability, Load Balancing, Location Transparency.

GLOSSÁRIO

.NET	*Framework* da Microsoft para desenvolvimento de sistemas.
ASP	*Active Server Pages*. Linguagem da Microsoft para construção de páginas HTML dinâmicas.
CORBA	*Common Object Request Broker Architecture*. Conjunto de protocolos e serviços para interoperabilidade de aplicações baseadas em objetos.
COTS	*Components Off-The-Shelf*. Termo usado para designar componentes de software prontos e de uso geral.
Cupom de venda	Comprovante emitido por um PDV no ato da venda de produtos. Contém os dados da venda (ex. preço por produto vendido, tributação, valor total da venda)
Cupom Eletrônico	Representação eletrônica (em forma de registros ou de objetos) de um cupom de venda.
Download	Operação de copiar um arquivo a partir de um servidor FTP (ou http) para uma máquina cliente.
Endereço IP	Endereço físico de uma máquina em uma rede local, em Intranets ou na Internet.
FIFO	*First In First Out*. Mecanismo básico de funcionamento de uma fila: o primeiro que entra é o primeiro que sai.
Fila	Estrutura de dados que implementa o mecanismo FIFO.
FTP	*File Transfer Protocol*. Protocolo de rede para troca de arquivos. É baseado em *socket*.
Handler	Elemento de software que, no contexto de MOM, é responsável pelo processamento das mensagens.
HTTP	*Hyper Text Transfer Protocol*. Protocolo de comunicação da Internet.
IPX/SPX	Protocolo de comunicação de rede usado no padrão *Netware*.
J2EE	Java 2 *Enterprise Edition*. *Framework* de funcionalidades e padrões para o Java, destinados ao desenvolvimento de aplicações servidoras e corporativas.
JDBC	Java *Database Connectivity*. Padrão do Java para acesso a bancos de dados relacionais que suportam SQL padrão.
JMS	Java Message Service. Padrão de implementação de um MOM definido pela Sun Microsystems para a Linguagem de Programação Java.
JNDI	Java *Naming Directory Interface*. Padrão do Java para implementação de um serviço de nomes.
JTA	Java *Transaction* API. Conjunto de funções especificadas para programas Java que realizam controle de transações distribuídas.
JTS	Java *Transaction Service*. Padrão do Java para controle de transações distribuídas.

JVM	Java *Virtual Machine*. Interpretador de byte code Java, necessário para executar qualquer programa neste linguagem.
Listener	Estrutura lógica de um servidor de comunicação responsável por aguardar e atender às solicitações de conexão.
Middleware	Infra-estrutura de comunicação para troca de mensagens com outros sistemas. Provê abstração às aplicações dos mecanismos de comunicação.
MOM	*Message Oriented Middleware*. Categoria de *middleware* que realiza troca de mensagens com transmissão assíncrona utilizando filas.
PDV	Ponto de Venda. Designação dos softwares que fazem o papel dos caixas registradores em lojas.
Porta lógica	Estrutura usada pelo *socket* para estabelecer uma conexão cliente-servidor
Set	Estrutura de dados que representa um conjunto, cujos elementos estão associados a uma chave. Não fornece a ordem de entrada dos elementos na estrutura.
Sistema de Retaguarda	No contexto do varejo, designa um conjunto de funcionalidades complementares às desempenhadas pelo PDV, e necessárias à automação de uma loja.
SOAP	*Simple Object Access Protocol*. Protocolo para troca de objetos entre aplicações que usa XML. Base dos *Web Services*.
Socket	Protocolo de comunicação utilizado em redes locais, Intranets e Internet.
SQL	*Structured Query Language*. Linguagem padrão de consulta a informações em bancos de dados relacionais.
Tag	Unidade representativa de comandos e de dados das linguagens de marcação, como HTML e XML.
TCP	Protocolo de comunicação de rede. É orientado a conexões.
Thread	Designa um processo de execução de código em linguagens de programação.
Timeout	Tempo de espera de uma transação ou conexão. Depois de expirado este tempo, a conexão é quebrada, a transação é desfeita ou a operação em curso é interrompida.
UDP	Protocolo de comunicação de rede. É orientado a pacotes ou datagramas.
UML	*Unified Modeling Language*. Linguagem usada para elaboração de modelos e projetos de sistemas.
X-25	Protocolo de comunicação, muito utilizado entre sistemas corporativos de grande porte (*mainframes*).
XML	*Extended Mark-Up Language*. Linguagem de representação de dados baseada em *tags*. É considerada uma extensão do HTML.

CONTEÚDO

ÍNDICE DE FIGURAS

ÍNDICE DE QUADROS

ÍNDICE DE TABELAS

Capítulo 1

Introdução

Este capítulo descreve inicialmente o contexto de desenvolvimento de sistemas de middleware e o estado atual de middleware orientado a mensagem. Em seguida, são apresentados os objetivos da dissertação. Finalmente, na última parte do capítulo é mostrada a estruturação do resto da dissertação.

1.1 Motivação

O desenvolvimento de aplicações corporativas tem sido uma tarefa cada vez mais complexa. Estas aplicações são normalmente compostas por componentes independentes que se encontram distribuídos em diversos ambientes heterogêneos e que requerem uma sofisticada infra-estrutura de comunicação. Além disto, estas aplicações têm demandado a implementação de requisitos funcionais e não funcionais cada vez mais elaborados, tais como necessidade de transações distribuídas, tolerância a falhas, segurança, transparência na comunicação, dentre outros.

A infra-estrutura de comunicação mencionada é genericamente chamada de *middleware* [5][34][44]. O *middleware* tem o papel básico de esconder detalhes de comunicação de baixo nível e de tratar a heterogeneidade de software e hardware dos ambientes onde as aplicações corporativas executam. Além disto, o *middleware* provê serviços (ex., segurança, transação, eventos, etc) que agregam valor à interação entre as partes da aplicação. Em termos práticos, o *middleware* é o elemento responsável por permitir que as partes da aplicação interajam sem que o desenvolvedor tenha que tratar diretamente detalhes de comunicação de baixo nível.

Muitos problemas relativos à comunicação entre aplicações corporativas surgiram e diversas categorias de *middleware* têm surgido [50]: *middleware* transacional, *middleware* procedural, *middleware* orientado a objetos (MOO) e *middleware* orientado a mensagem (MOM). Cada uma destas categorias essencialmente resolve um conjunto de problemas distintos e comuns às aplicações corporativas, tais como comunicação assíncrona, processamento de transações e assim por diante. Dentre estas categorias, os MOMs [16] são os mais utilizados em ambientes corporativos.

Entretanto, o aumento de complexidade das aplicações corporativas e o surgimento de novas tecnologias de desenvolvimento de sistemas, tais como orientação a objetos, fazem com que características adicionais, facilidades de utilização e otimizações dentro de MOMs sejam necessidades constantes. Além disto, à medida que novos requisitos vão sendo adicionados às aplicações, domínios específicos que antes não utilizavam sistemas de *middleware* passam a utilizá-los de forma extensiva. Naturalmente, surgem também novas necessidades a serem incorporadas aos MOMs já existentes.

Um exemplo desta adesão de domínios de aplicação à utilização de MOMs é a rápida evolução dos sistemas embarcados, que passaram de soluções pontuais a aplicações que necessitam se conectar com outros dispositivos embarcados e acessar a Internet. Estes novos requisitos levam à necessidade de uso de uma infra-estrutura de comunicação, normalmente já existente, mas que precisa incorporar características adicionais inerentes ao domínio de aplicação.

Como será mostrado nos capítulos seguintes, a incorporação de características adicionais e de otimizações a um MOM resolve alguns problemas relativos à comunicação entre as

aplicações. A simplificação na forma de uso de um MOM também contribui para uma maior facilidade de implementação e de manutenção das aplicações.

1.2 Problemas com o Estado da Arte

Diversos MOMs são normalmente utilizados para comunicação entre aplicações corporativas. Muitos deles possuem um vasto conjunto de funcionalidades e potencialidades que se propõem a atender às necessidades da maioria das aplicações, mas alguns problemas podem ser observados:

- **Complexidade de uso dos MOMs**: apesar de existir um padrão para implementação de *middleware* orientado a mensagem, programadores usuários de MOMs têm certa dificuldade para implementar funções que utilizam um MOM;

- **Complexidade de administração**: normalmente, a utilização de um MOM é vinculada à montagem de um ambiente de serviços, onde diversos parâmetros têm que ser administrados. Muitas vezes, o MOM é utilizado, mas o custo de sua utilização é a criação de uma infra-estrutura nova para administrar o ambiente criado;

- **Flexibilidade de utilização de alguns mecanismos de comunicação**: normalmente, os MOMs trabalham sempre com um único mecanismo para troca de informações. No entanto, muitas aplicações demandam a utilização de formas diversas para troca de informações;

- **Escalabilidade**: com a crescente dependência de processamento distribuído, as aplicações demandam cada vez mais disponibilidade de serviços e processamento em escala

Embora os MOMs existentes forneçam uma série de funcionalidades capazes de satisfazer à maioria dos requisitos de comunicação entre aplicações corporativas, os problemas citados persistem.

1.3 Objetivos

O principal objetivo desta dissertação é projetar e implementar um MOM, o *Hermes*, o Deus grego da comunicação, que possui as características funcionais necessárias às aplicações corporativas (ex. balanceamento de carga e transparência de localização), e que incorpora melhorias e soluções para os problemas apresentados na seção anterior. Para resolver e ou minimizar estes problemas, alguns objetivos incluem:

- Estabelecer padrões de componentes que podem ser reutilizados em outros MOMs e até mesmo em outros domínios de aplicação;

- Propor e implementar melhorias referentes a algumas características dos MOMs, tais como: balanceamento de carga, disponibilidade, formas de processamento de mensagens.

- Implementar características não encontradas em MOMs, tais como: criação e gerência dinâmica de filas, combinações de tipos de transmissão e algumas otimizações na implementação do servidor de comunicação.

Com o objetivo de mostrar a efetividade do *Hermes*, este trabalho apresenta um estudo de caso, onde as características e otimizações do *Hermes* são utilizadas para atender a requisitos de uma aplicação corporativa, resolvendo problemas relativos à performance, à disponibilidade de serviços e ao processamento em escala.

1.4 Estrutura da Dissertação

A abordagem utilizada para estruturar esta dissertação é a de um estudo *top-down*: o conhecimento dos conceitos mais genéricos é apresentado, com ênfase no objetivo principal de estudo, os MOMs; um padrão de especificação para MOMs é apresentado, as características de três MOMs são criticamente analisadas; o *Hermes* é descrito, seguindo uma sistemática de apresentação das suas características (requisitos), arquitetura / módulos (análise) e detalhamento dos módulos (projeto e implementação); finalmente, é apresentado um estudo de caso, que mostra as características do *Hermes* como solução para os problemas mencionados na seção anterior.

Seguindo esta estruturação os capítulos estão organizados como segue:

Capítulo 2 – Conceitos Básicos de *Middleware*: este capítulo define *middleware* e apresenta os requisitos que estes atendem. A partir destes requisitos, é apresentada uma classificação dos sistemas de *middleware*, adotada como referência neste trabalho. Cada categoria desta classificação é descrita e o nível de atendimento dos requisitos apresentados é mostrado. Uma maior ênfase é dada à categoria de *middleware* orientado a mensagens, objeto principal de estudo deste trabalho.

Capítulo 3 – Trabalhos Relacionados: este capítulo apresenta um padrão definido para MOMs, o JMS (Java *Message Service*). Em seguida, o capítulo detalha as características de três MOMs, associando estas aos requisitos estabelecidos no Capítulo 2, e descrevendo as características gerais de MOMs naqueles observadas. Para cada um destes MOMs, um breve resumo comparativo com o *Hermes* é apresentado.

Capítulo 4 – MOM Hermes: este capítulo descreve o MOM *Hermes,* a principal contribuição deste trabalho. Primeiro, são apresentadas as características gerais dos MOMs observadas no *Hermes*. Depois, é mostrada a arquitetura do mesmo, os seus módulos e componentes, e as relações entre estes. Por fim, os módulos e componentes são descritos e detalhados. Durante a descrição e detalhamento do *Hermes*, as otimizações e inovações são apresentadas e descritas.

Capítulo 5 – Estudo de Caso: esse capítulo mostra um estudo de caso real com o objetivo de validar o *Hermes*. Para isto, é apresentado um estudo de caso onde o *Hermes* atende a requisitos funcionais e não funcionais dentro de cenários observados em um domínio de aplicação específico. Neste estudo, são mostradas as vantagens obtidas com as otimizações e características adicionais presentes no *Hermes*.

Capítulo 6 – Conclusão: este capítulo mostra as conclusões obtidas durante o desenvolvimento desse trabalho, assim como as principais contribuições que ele fornece para área de Sistemas Distribuídos. Serão mostrados alguns possíveis trabalhos futuros, assim como algumas extensões previstas para o *Hermes.*

Capítulo 2

Middleware

Este capítulo apresenta os conceitos básicos de middleware. Padrões, protocolos e funcionalidades de comunicação entre sistemas são detalhados. Uma parte deste detalhamento diz respeito ao tipo específico de middleware abordado nesta dissertação, o middleware orientado a mensagens, cuja abreviatura é MOM (Messaging Oriented Middleware).

2.1 Introdução

Este capítulo apresenta inicialmente o conceito de *middleware*, mostrando suas características gerais e os principais requisitos não-funcionais. Finalmente, apresenta uma classificação, adotada como a referência para o estudo desenvolvido neste trabalho, e detalhes de cada uma das classes de *middleware*.

Na classificação adotada, os principais tipos de sistemas de *middleware* são brevemente apresentados e as suas funcionalidades e características básicas são associadas aos principais requisitos não-funcionais descritos. O que se nota é que a maioria das categorias apresenta deficiências no atendimento pleno destes requisitos. Por fim, a categoria de sistemas de *middleware* orientados a mensagens é descrita com mais detalhes, sendo enfocada também a aderência de suas características aos requisitos não-funcionais elencados.

2.2 Conceito de Middleware

O conceito atual de *middleware* é bastante abrangente, por conta deste termo ser usado em diversos contextos de desenvolvimento de softwares e ser aplicado a diversos tipos de programas, desenvolvidos em contextos diferentes e com propósitos diferentes. O que existe em comum entre estes diversos componentes denominados de *middleware* é o fato deles todos resolverem serem utilizados como software de comunicação.

Um *middleware* apresenta serviços comuns de infra-estrutura de software, necessários à grande maioria das soluções de sistema atualmente desenvolvidas no contexto moderno das aplicações comerciais e científicas [34]. Através de sistemas de *middleware*, diferentes aplicações, normalmente com características distintas e muitas vezes desenvolvidas em linguagens diferentes e sendo executadas em plataformas operacionais de software e de hardware distintas, trocam informações entre si de diversas maneiras.

Para muitos autores, o termo *middleware* está associado a componentes que disponibilizam serviços de comunicação entre componentes de softwares. De acordo com [50], *middleware* representa uma classe de tecnologias de software projetadas para ajudar a gerenciar as complexidades inerentes a sistemas distribuídos. Nesta visão, sistemas de *middleware* dão uma abstração aos programadores das funcionalidades dos sistemas distribuídos de como as questões referentes a transmissão e a recepção de informações são resolvidas.

Além das questões referentes aos protocolos de transmissão e de recepção de dados, sistemas de *middleware* também se preocupam em abstrair os programadores dos sistemas distribuídos de como os problemas de heterogeneidade são resolvidos, permitindo que aplicações com características diversas se comuniquem entre si.

Existem diversos padrões e protocolos comuns, destinados a permitir que aplicações diferentes possam interagir entre si. Estes padrões têm evoluído e convergido para que seja cada vez mais fácil integrar aplicativos e desenvolver sistemas distribuídos. A Figura 2.1 mostra o papel geral de um *middleware* dentro do cenário de sistemas distribuídos.

Na Figura 2.1, pode-se observar que o *middleware* é uma camada intermediária entre a aplicação e o sistema operacional, abstraindo aquela de como os mecanismos específicos de comunicação implementados nos sistemas operacionais são ativados.

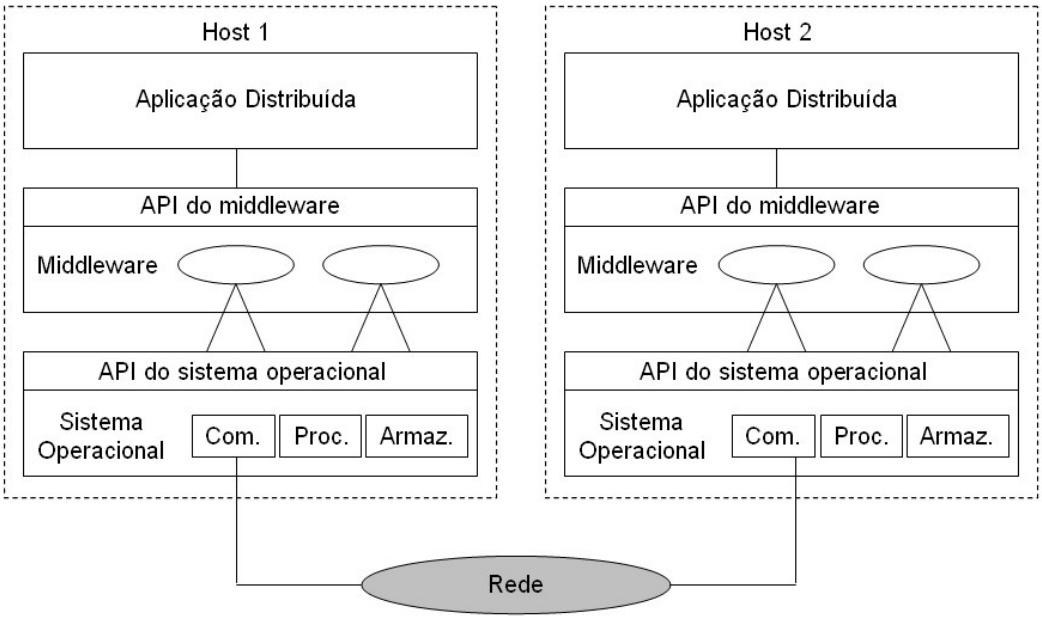

Figura 2.1 – Camada correspondente ao *Middleware* para Aplicações Distribuídas

Os sistemas de *middleware* não se destinam apenas a abstrair o programador dos processos e protocolos de comunicação e das questões de heterogeneidade. Existem diversos outros requisitos de sistemas distribuídos que são atendidos por sistemas de *middleware*. Por conta da diversidade e aplicabilidade, estes sistemas podem ser sub-categorizados em diversas classes, cada uma das quais com características e propósitos bem definidos.

Existem diversas classificações para os sistemas de *middleware*, sendo todas concordantes quanto ao conjunto de características que um dado sistema deve apresentar a fim de se enquadrar em uma das classificações apresentadas. A sistemática a ser abordada neste estudo seleciona uma classificação para sistemas de *middleware*, apresenta uma lista de requisitos não-funcionais que os sistemas de *middleware* devem prover e descreve cada uma das categorias consideradas na classificação utilizada como referência de estudo. Tal classificação é a proposta pelo autor [50], que considera um *middleware* como tendo características de uma camada entre as aplicações distribuídas e os sistemas operacionais, interagindo em última instância com as funcionalidades de comunicação dos tais sistemas. Com base nesta premissa, o citado autor define uma lista básica de requisitos não-funcionais requeridos por qualquer sistema distribuído e

apresenta a classificação dos sistemas de *middleware* atualmente aceita e citada por diversos outros autores como [34], sendo ela usada e considerada em diversos estudos sobre sistemas de *middleware*.

Os principais requisitos não funcionais observados nos sistemas de *middleware* estudados e implementados serão analisados no contexto de cada tipo de *middleware* existente, e especificamente relacionados com o objeto de estudo deste trabalho – o *middleware* orientados a mensagens – MOM.

2.3 Requisitos Não-Funcionais para Middleware

Qualquer componente de software que se destina a atender a necessidades específicas de comunicação e de troca de informações entre sistemas deve prover uma série de características comuns, cujos níveis de implementação e de complexidade variam com os diferentes tipos de sistemas de *middleware* existentes. Emmerich [50] define uma lista abrangente destas características que contempla um conjunto de requisitos não-funcionais normalmente providos por componentes de comunicação. Estes requisitos são discutidos a seguir.

2.3.1 Comunicação de Rede

É a premissa básica para qualquer *middleware* existir. Permite que dois aplicativos troquem informações e dados de controle através de uma rede física. Normalmente, os sistemas distribuídos são construídos no topo da camada de transporte, notadamente TCP e UDP, pois as camadas mais baixas são providas pelos sistemas operacionais.

Assim, pode-se concluir que os sistemas de *middleware* normalmente usam a camada de transporte (hoje, a quase que totalidade usa realmente TCP e UDP) para implementar a troca de dados via rede. Cabe aos sistemas de *middleware* proverem complexos mecanismos de conversão de estruturas de dados de alto nível, recebidas dos aplicativos, em bytes a serem enviados através da rede para um outro aplicativo. Este processo é denominado de *marshalling*. Da mesma forma, cabe ao *middleware* receber bytes através da rede e convertê-los também em estruturas de alto nível a serem retornadas às aplicações. Este processo é chamado de *unmarshalling*.

2.3.2 Coordenação

Este conceito abrange as formas de sincronização em um processo de comunicação: síncrona ou assíncrona. A primeira forma bloqueia um componente que espera o processamento de uma requisição por outro componente. A segunda forma não bloqueia um componente que requisita o processamento de outro componente, cabendo a este último coordenação e sinalização de fim de processamento.

Um outro conceito de coordenação diz respeito a um conjunto de atividades relacionadas à gerência dos recursos instalados em máquinas que hospedam serviços distribuídos: reinicialização de componentes por conta de desligamento dos seus respectivos *hosts*, gerência de sobrecarga e gerência de recursos ociosos. A coordenação ainda abrange o conceito de ativação, que exige a inicialização e parada de um componente de processamento independente das aplicações que eles executam. Por fim, coordenação exige também que se controle a concorrência de requisições que são executadas simultaneamente em um mesmo *host*.

2.3.3 Confiabilidade

Este é um requisito não funcional exigido em qualquer contexto de troca de informações entre aplicações. Normalmente, confiabilidade e performance são requisitos conflitantes e os desenvolvedores de sistemas de *middleware* têm que equilibrar o atendimento destes dois fatores.

Muitas vezes, confiabilidade é sinônimo de redundância e de mecanismos mais complexos, exigindo um uso maior de memória física e mais capacidade de processamento. A literatura apresenta quatro níveis distintos de confiabilidade para sistemas distribuídos:

- O nível de melhor esforço diz que requisições de serviços com este nível de confiabilidade não dão nenhuma garantia de que a tal requisição será executada;

- O nível de no máximo uma vez diz que requisições de serviços com este nível de confiabilidade garantem que uma requisição será executada uma única vez, e pode não ser executada. Neste caso, o elemento que realizou a requisição é notificado sobre a não execução da referida requisição;

- O nível de no mínimo uma vez diz que requisições de serviços com este nível de confiabilidade dão a garantia de que tal requisição será executada ao menos uma vez e pode ser executada mais de uma vez; e

- O nível de exatamente uma vez é o mais alto grau de confiabilidade. Garante que uma requisição de serviço é executada uma e apenas uma única vez.

Outros aspectos relacionados com confiabilidade dizem respeito a questões de *timeout*, de execução ordenada de requisições e de replicação de instâncias de execução. Estes pontos, quando devidamente abordados, garantem uma maior disponibilidade dos serviços oferecidos.

Outra questão fundamental ligada à confiabilidade diz respeito ao conceito de transação. Uma transação define a atomicidade de um fluxo de execução. Isto significa que, se um passo do fluxo de execução não for realizado com sucesso, o fluxo deve ser interrompido e toda atualização realizada nos passos anteriores do fluxo deve ser desfeita.

2.3.4 Escalabilidade

Este termo define a capacidade de uma solução de se adaptar a um futuro aumento de carga sem grandes impactos na arquitetura física e lógica da solução. Uma das abordagens mais usadas para implementação de escalabilidade em sistemas distribuídos é a de transparência, que consiste em permitir que um serviço seja requisitado sem que o processo requisitor precise especificar qual processo executor irá realizar o processamento. Existem várias modalidades de transparência a serem usadas para garantir escalabilidade.

A transparência de acesso garante que componentes requisitores de serviços acessem os mesmos, estando eles em máquinas locais ou remotas. A transparência de localização garante que componentes requisitores de serviços não precisem saber a localização física dos serviços para acessá-los. A transparência de migração pode permitir facilmente o balanceamento de carga, fazendo com que requisições sejam redirecionadas para locais distintos de acordo com a disponibilidade de processamento destas. Na transparência de replicação, a transparência de migração é estendida para permitir que componentes de processamento sejam dinamicamente replicados em máquinas com mais disponibilidade de processamento.

2.3.5 Heterogeneidade

A troca de dados entre sistemas que são, de alguma forma, heterogêneos sempre foi um desafio para quem implementa soluções de comunicação entre sistemas. A heterogeneidade entre sistemas pode existir por conta de três elementos críticos relacionados à estrutura básica necessária ao desenvolvimento de sistemas:

- Sistemas operacionais. Programas diferentes que são executados em sistemas operacionais diferentes normalmente trocam dados com formatos diferentes e através de protocolos diferentes;

- Plataformas de hardware. Diferença de plataformas de hardware significa que problemas relacionados com sistemas de codificação de caracteres diferentes têm que ser resolvidos; e

- Linguagens de programação. Muitas vezes, programas desenvolvidos em linguagens diferentes apresentam padrões de representação para tipos de dados diferentes.

Estas questões citadas precisam ser tratadas e consideradas no projeto de um *middleware*, a fim de que o mesmo possa ser usado em contextos diferentes e garanta a interoperabilidade entre aplicativos com características diversas. As diversas necessidades existentes em sistemas distribuídos fizeram com que muitos sistemas de *middleware* fossem desenvolvidos desde que o conceito de computação distribuída foi criado. Estes diversos produtos podem ser categorizados por conta deles apresentarem características comuns e resolverem problemas específicos de sistemas distribuídos.

2.4 Middleware Transacional

Esta categoria de *middleware* suporta transações envolvendo componentes que são executados em *hosts* distribuídos. Estes sistemas de *middleware* usam o protocolo de controle para transações distribuídas denominado *two-phase commit* e têm como finalidade principal atender as necessidades que aplicativos têm de realizar controle para transações distribuídas. Alguns exemplos de produtos nesta categoria são: CICS (IBM), Tuxedo (BEA) e Encina (*Transarc*). A arquitetura típica destes sistemas de *middleware* está mostrada na Figura 2.2 e mostra que sistemas de *middleware* enquadrados nesta categoria têm a responsabilidade de gerenciar processos transacionais distribuídos entre diversas aplicações e bancos de dados diferentes.

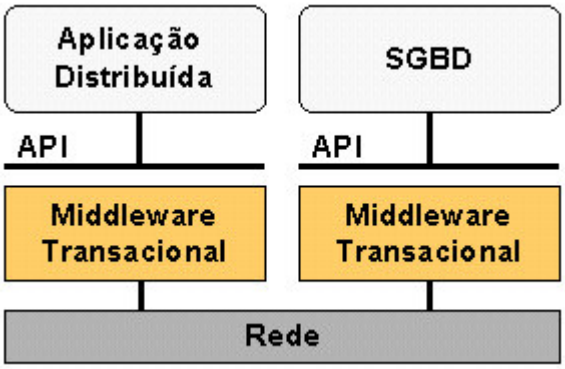

Figura 2.2 – Papel do *Middleware* Transacional

Um *middleware* transacional normalmente é utilizado em processos de sincronização de dados entre aplicações e instâncias de SGBDs diferentes, que interagem através de uma mesma rede física. Os processos de comunicação envolvidos nos procedimentos de sincronização usam, em níveis mais baixos de comunicação, protocolos comuns de mercado, notadamente TCP e UPD, e em níveis mais altos protocolos definidos por organismos internacionais para distribuição e replicação remota de dados. Os requisitos não funcionais nos sistemas de *middleware* transacionais têm as seguintes características:

Comunicação de rede
Os sistemas de *middleware* transacionais garantem a transparência da complexidade dos mecanismos de transmissão de dados e de informações de controle através da rede, permitindo de forma absolutamente natural que clientes e servidores sejam hospedados em máquinas físicas diferentes, ligadas através de uma rede.

Coordenação
Em sistemas de *middleware* transacionais, clientes podem usar comunicação síncrona ou assíncrona para requisição de processamento. Mecanismos de ativação podem ser disponibilizados, e, dependendo do produto, diversas políticas de ativação são permitidas.

Confiabilidade
O maior problema relativo a este requisito para sistemas de *middleware* transacionais é o de garantir a integridade das transações a serem gerenciadas. Muitas vezes, as implementações destes sistemas de *middleware* se valem do fato de que SGBDs já implementam o protocolo *two-phase commit*. A própria natureza dos sistemas de *middleware* transacionais faz com que estes tenham naturalmente um nível de confiança alto, pois sem isto ele não seria capaz de desempenhar seu principal papel.

Escalabilidade
Muitos sistemas de *middleware* transacionais implementam balanceamento de carga e replicação, normalmente associados a mecanismos proprietários dos SGBDs com os quais estes sistemas interagem.

Heterogeneidade
É bem suportada por sistemas de *middleware* transacionais, muito por conta do fato de que seus mecanismos estarem intimamente ligados a SGBDs que implementam protocolos de transações distribuídas.

Embora os sistemas de *middleware* transacionais atendam bem aos requisitos não funcionais básicos associados a sistemas de *middleware*, existem alguns pontos que eles não contemplam de forma satisfatória:

- A introdução de um *overhead* em soluções que não necessitam de controle transacional, caso um *middleware* transacional seja usado nesta situação. È comum isto acontecer por conta do fato de que um *middleware* transacional quase sempre contempla uma série de outras características, necessárias em contextos bem diversos do controle de transações distribuídas; e

- A realização dos processos de *marshalling* e de *unmarshalling* que convertem as estruturas de dados manipuladas pelos clientes em bytes muitas vezes têm que ser feitos manualmente e fora da implementação do *middleware*

2.5 Middleware Procedural

Mecanismos de RPC (*Remote Procedure Call*) constituem este tipo de *middleware*. A idéia de RCP surgiu no início dos anos 80 e desde então várias fabricantes de software vêm disponibilizando tal mecanismo como parte dos seus sistemas operacionais. Então, padrões de RPC foram estabelecidos e a maioria das implementações de sistemas operacionais, notadamente as famílias *Unix* e *Windows*, disponibilizam formas de RPC para acessar processos e programas que neles são executados.

Na Figura 2.3, é mostrado claramente que o sistema operacional faz o papel de interpretar e de encaminhar as chamadas remotas de um determinado programa para outro através da rede. Por conta disto, sistemas baseados em RPC são dependentes de recursos internos do

sistema operacional, necessitando muitas vezes de alterações para se tornarem portáveis de um sistema operacional para outro.

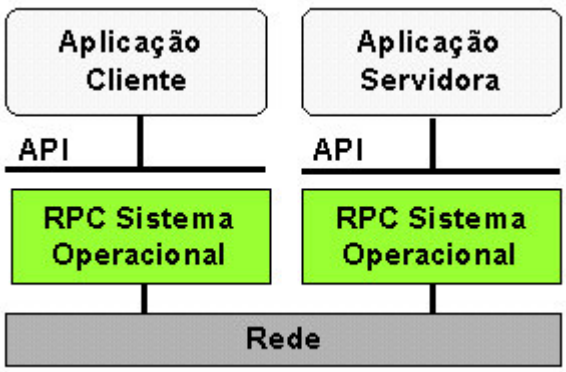

Figura 2.3 – Visão Geral de um *Middleware* Procedural

Os requisitos não funcionais nos sistemas de *middleware* procedurais têm as seguintes características:

Comunicação de rede
Os sistemas de *middleware* procedurais suportam a definição de componentes servidores como programas que podem ser chamados através de RPC. Sendo assim, é necessário que os mecanismos de comunicação de rede suportem codificação / decodificação de chamadas de rotinas, tratamentos de parâmetros passados nas chamadas e de retorno de valor resultante da execução.

Coordenação
Mecanismos de RPC são tipicamente interações síncronas entre um único cliente e um único servidor. Comunicação assíncrona e *multicasting* não são suportados diretamente por mecanismos de RPC e políticas de ativação podem ser implementadas para decidir se uma *procedure* de um dado programa estará sempre disponível ou será inicializada sob demanda.

Confiabilidade
Mecanismos de RPC são executados com nível de confiabilidade normalmente situados na categoria “no máximo uma vez”, retornando exceções se uma chamada remota a uma *procedure* falhar.

Escalabilidade
Em mecanismos de RPC, esta característica é bem limitada. As famílias de sistemas operacionais mais usados não possuem mecanismos nativos de replicação que possam ser usados para escalar programas que suportam RPC.

Heterogeneidade
Por conta de sua própria natureza e do suporte nativo de diferentes sistemas operacionais a RPCs, estes mecanismos atendem naturalmente ao requisito de heterogeneidade. Assim,

sistemas de *middleware* procedurais podem ser usados com diferentes linguagens de programação e em sistemas e plataformas operacionais distintos também.

Sistemas de *middleware* procedurais não são tolerantes a falhas nem escaláveis por natureza. A vantagem de sistemas de *middleware* procedurais é que eles realizam *marshalling* e *unmarshalling* de chamadas de funções, permitindo que interações remotas entre programas distintos sejam realizadas de forma transparente para quem está chamando as *procedures* associadas a tais interações, definindo suas interfaces associadas. Uma outra limitação de sistemas de *middleware* procedurais é que eles não são reflexivos, ou seja, não é possível um programa que suporta RPC retornar, em uma chamada remota, outro programa RPC.

2.6 Middleware Orientado a Objetos (MOO)

Este tipo de *middleware* é uma evolução dos sistemas de *middleware* procedurais. A idéia de chamada remota de *procedures* persiste, só que com mecanismos mais sofisticados associados aos conceitos de orientação a objetos (OO). Assim, nos mecanismos de chamada a *procedures*, denominadas em OO de métodos, são contemplados os conceitos de herança e de referências, aplicados em linguagens de programação que usam o paradigma OO.

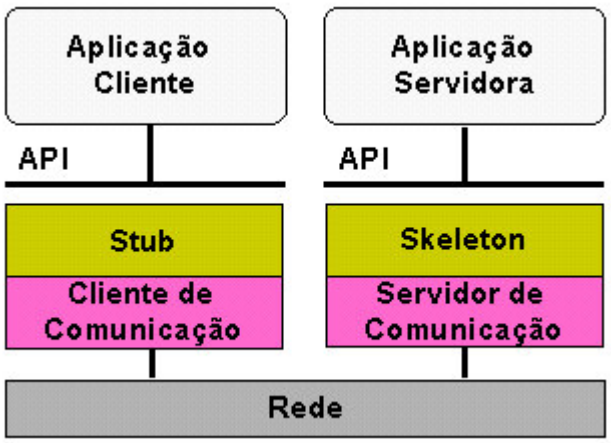

Figura 2.4 – Visão Geral de um *Midleware* de Objetos

A Figura 2.4 mostra uma visão geral das partes componentes de um *middleware* orientado a objetos. Neste contexto surge um componente de conversão de chamadas e de objetos em dados a serem enviados pela rede – o *stub*. Além do *stub*, o *skeleton* aparece como o elemento que converte dados da rede em chamadas e objetos e converte o retorno de objetos e de exceções em dados a serem retornados para a aplicação cliente através da rede. Neste contexto, surgem elementos responsáveis por fazer com que as aplicações possam realizar chamadas remotas de métodos com parâmetros e retornos definidos como objetos de forma transparente e simples. Os mecanismos de transmissão e de

recepção de dados são os usuais, normalmente utilizados por outros sistemas de *middleware*.

Alguns produtos e padrões incluídos na classificação de *middleware* orientado a objetos incluem o CORBA (*Common Object Request Broker Architecture*) [21][30], o COM (*Component Object Model*) [20][21] e o padrão EJB (*Enterprise Java Beans*) [26]. Os requisitos não funcionais nos sistemas de *middleware* procedurais têm as seguintes características:

Comunicação de rede
Mecanismos de comunicação de rede suportam codificação / decodificação de chamadas de rotinas, tratamentos de parâmetros passados nas chamadas e de retorno de valor resultante da execução dos métodos invocados. Estes mecanismos têm que ser mais sofisticados que os desenvolvidos em sistemas de *middleware* procedurais, pois aqueles devem suportar tipos complexos de dados (definidos normalmente através de classes), gerenciamento de referências remotas e herança.

Coordenação
Como os mecanismos de RPC, sistemas de *middleware* orientado a objetos realizam interações síncronas entre um único cliente e um único servidor, embora algumas implementações suportem assincronismo. Políticas de ativação similares às adotadas para os sistemas de *middleware* procedurais são também usadas em sistemas de *middleware* orientado a objetos, tratando da instanciação de objetos servidores e do tempo de sobrevida dos mesmos. Algumas implementações tratam de forma relativamente abrangente o controle de concorrência, através dos seus próprios modelos de sincronização de execução.

Confiabilidade
O nível padrão de confiabilidade para sistemas de *middleware* orientado a objetos é o "no máximo uma vez", suportando modelos complexos de exceções baseados em objetos. Algumas implementações suportam o conceito de transações.

Escalabilidade
Esta é uma característica limitada nos sistemas de *middleware* orientado a objetos, embora alguns produtos implementem suporte a balanceamento de carga. De qualquer forma, os mecanismos de replicação encontrados são bastante limitados e pouco eficientes.

Heterogeneidade
Este requisito é suportado pelos diferentes sistemas de *middleware* orientado a objetos de diferentes maneiras. O COM permite que programas escritos em linguagens diferentes interajam entre si, desde que eles sejam executados em uma mesma família de sistemas operacionais. O CORBA permite uma interação mais abrangente, estabelecendo padrões que permitem a comunicação entre programas escritos em linguagens diferentes e sendo executados em sistemas operacionais diferentes. O RMI (*Remote Method Invocation*)

[26] permite que programas escritos em uma mesma linguagem interajam entre si, mesmo que eles sejam executados em sistemas operacionais distintos.

O ponto forte dos sistemas de *middleware* orientado a objetos é que eles normalmente provêem um modelo bastante poderoso de interação remota entre objetos. No entanto, as limitações referentes à escalabilidade fazem com que o uso desta categoria de *middleware* seja restrito quando é feito em aplicações de larga escala.

2.7 Middleware Orientado a Mensagens

Um *middleware* orientado a mensagens, cuja abreviatura é, MOM, suporta um tipo de comunicação entre sistemas que é baseado fundamentalmente na troca de mensagens. Esta troca normalmente se dá de forma assíncrona e quase sempre ordenada. Por conta disso, as mensagens são convenientemente processadas e os resultados destes processamentos disponibilizados para verificação futura.

Mensagens podem ser tratadas por aplicações como eventos, em última instância. Estes eventos são ordenadamente tratados e processados. Estas mensagens podem ser priorizadas nas filas de processamento, descartadas caso estejam obsoletas, ou mesmo mantidas naquelas se houver motivo funcional para que isto aconteça. Os MOMs representam a categoria de sistemas de *middleware* mais versátil e difundida comercialmente, pois as suas características funcionais e sua robustez os tornam aptos a serem usados em uma ampla gama de aplicações e de funcionalidades.

Atualmente, o uso de MOMs vem se estendendo às aplicações móveis, pois este tipo de *middleware* se mostrou o mais adequado para prover conectividade a tais dispositivos. Suas características de garantia de entrega das mensagens, mesmo com uma alta taxa de interrupção nos processos de transmissão – fato bastante comum no mundo *wireless* - e o baixo acoplamento entre as aplicações clientes e servidoras fazem com que o MOM seja o mecanismo ideal para troca de dados envolvendo tais dispositivos.

Os MOMs também são largamente utilizados em diversos contextos de aplicações, provendo soluções robustas e simples para problemas de conectividade e de processamento distribuído, operando sob condições de alto volume de dados e com alta performance.

2.7.1 Arquitetura

Os MOMs normalmente têm uma arquitetura geral que contempla as funcionalidades básicas associadas a sistemas de *middleware* orientados a mensagens. Esta arquitetura contempla alguns pontos básicos associados às funcionalidades e aos requisitos que um MOM normalmente contempla. A Figura 2.5 mostra a arquitetura genérica de um MOM e seus elementos principais.

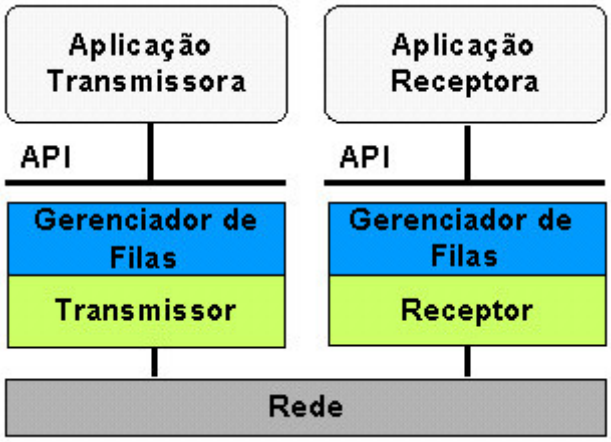

Figura 2.5 – Visão Geral dos Elementos de um *Middleware* Orientado a Mensagens

O gerenciador de filas no lado cliente é responsável por inserir convenientemente as mensagens nas filas, considerando que cada mensagem está associada a um tópico específico e este a uma fila. Ele também garante o assincronismo de transmissão, onde um processo diferente do que colocou a mensagem na fila tenta transmitir a mesma ao seu destino, usando o transmissor.

Em caso de sucesso no processo de transmissão, a mensagem é retirada da fila. Se o processo de transmissão não for realizado com sucesso, a mensagem é mantida na fila até que o receptor esteja disponível. O gerenciador de filas no lado cliente ainda garante a persistência das mensagens e a tentativa de transmissão das mesmas mesmo que um desligamento ocorra na máquina onde o cliente está sendo executado.

No lado servidor, o gerenciador de filas tem o papel de inserir uma mensagem entregue pelo receptor na sua respectiva fila, garantindo o assincronismo de processamento em relação ao processo que o colocou na fila. Se o processamento for realizado com sucesso, a mensagem é retirada da fila. Caso contrário, a mensagem permanece na fila por um tempo determinado ou indefinidamente. O gerenciador de filas no lado servidor ainda garante a persistência das mensagens, permitindo que mensagens recebidas sejam processadas convenientemente após um eventual *power-off* da máquina onde o servidor está sendo executado. A complexidade e a forma de implementação destas características básicas do gerenciador de filas é que vai determinar uma maior eficiência e confiabilidade do MOM.

O elemento transmissor é responsável por realizar o processo de comunicação entre o lado cliente e o lado servidor, normalmente usando um protocolo de comunicação bem conhecido e consagrado. Normalmente, o transmissor implementa uma interface a ser usada e esta implementação pode ser trocada se for necessário trocar o protocolo de transmissão de dados a ser usado entre o cliente e o servidor. Esta separação entre o transmissor e o gerenciador de filas garante que o impacto da troca de protocolo de comunicação não influencia o mecanismo de gerência das filas de um MOM. Em muitas implementações, a troca do protocolo de comunicação do transmissor é feita através de parametrização da aplicação. Outras implementações permitem que o protocolo de

transmissão seja determinado na hora de enviar uma mensagem ou na hora de cadastrar uma fila a ser usada no processo de transmissão.

O elemento receptor faz o papel típico de um servidor de comunicação, que pode ser implementado em diversos protocolos de troca de dados. Este servidor entrega a mensagem recebida ao gerenciador de filas que a encaminha para a fila de entrada conveniente no lado servidor e notifica, de forma assíncrona, um processo que irá ler a mensagem da fila e processá-la ou tratá-la de forma conveniente.

O receptor, além de implementar um servidor de comunicação, tem o papel de controlar a concorrência de mensagens recebidas para processamento. Normalmente, este receptor é multi-processo e tem a capacidade de alocar *threads* diferentes e simultâneas para atender a chegada de mensagens simultâneas.

2.7.2 Características

Os MOMs possuem características específicas que estão associadas aos seus propósitos funcionais. Estas características permitem que sistemas de *middleware* enquadrados nesta categoria se diferenciem entre si em termos de funcionalidades disponíveis e em termos de qualidade de serviço. As principais características funcionais que normalmente são encontradas em um MOM estão descritas abaixo.

Enfileiramento e sincronismo das mensagens
Um *middleware* orientado a mensagens implementa um esquema de enfileiramento das mesmas, tanto no lado cliente quanto no lado servidor. A especificação desta característica se resume a uma sigla – FIFO (*First In First Out*) – que implica em uma transmissão ordenada das mensagens nas filas de saída (filas do cliente), e em um processamento ordenado das mensagens nas filas de entrada (filas do servidor). Esta característica permite que a transmissão e o processamento das mensagens sejam sincronizados.

Assincronismo de transmissão e de processamento das mensagens
Os MOMs implementam transmissão assíncrona das mensagens enfileiradas nas filas do lado cliente em relação ao processo que inclui a mensagem na fila. No lado servidor, o processamento de uma mensagem é assíncrono em relação ao processo que inclui a dada mensagem na fila. Esta característica permite que aplicações clientes estejam totalmente desacopladas do processo de transmissão e de processamento de mensagens em um servidor.

Persistência das mensagens enfileiradas
Os MOMs normalmente implementam um mecanismo de persistência das mensagens a serem colocadas nas filas de saída e nas filas de entrada. A persistência é necessária para garantir que um *power-off*, tanto no lado cliente quanto no lado servidor, não implique em perda de mensagens já colocadas nas filas. Os mecanismos de persistência das mensagens nas filas variam e normalmente suportam mais de um tipo de mecanismo de

persistência, cabendo ao usuário do dado produto definir qual mecanismo será utilizado. Em alguns MOMs, existem opções para desabilitar a persistência das mensagens no lado cliente e/ou no lado servidor, que permite mais agilidade no processo de transmissão mas reduz a robustez do processo em casos de contingência, aumentando a possibilidade de perda de mensagens.

Tolerância a falhas no cliente
Os MOMs possuem mecanismos específicos de tolerância a falhas no cliente. Impossibilidade de transmissão de uma mensagem implica que a mesma deve ser retransmitida posteriormente, logo quando a conexão com o servidor destino da mensagem a ser transmitida for normalizada. *Power-offs* das aplicações clientes não devem causar perdas de mensagens que já foram enfileiradas para transmissão. Para garantir este último tópico, usa-se um mecanismo de persistência das mensagens nas filas de saída, comentados no parágrafo anterior.

Filas e tópicos
Um MOM não trabalha com uma única fila de transmissão e de recepção. As filas dos MOMs são normalmente categorizadas por tópicos, onde cada tópico representa uma fila de saída e uma fila de entrada. Existem casos em que os tópicos são considerados o tipo e o endereço de destino da mensagem. Em outras situações, apenas um tema define o tópico relacionado à fila.

Formas de processamento das mensagens
Uma vez que um MOM recebe uma mensagem e a coloca na fila de entrada, tal mensagem deve ser processada pela aplicação onde o MOM está sendo executado. As formas como a aplicação recebe a notificação de que existe uma mensagem em uma determinada fila de entrada são variadas. Uma das maneiras de comportamento do MOM é permitir que a aplicação tenha um processo "em espera" enquanto uma dada fila estiver vazia, e este processo só se mantém acordado enquanto existirem mensagens na tal fila. Uma outra forma de processar mensagens é através do mecanismo de gatilho, onde um programa externo é executado pelo MOM toda vez que uma mensagem é colocada na fila de entrada. Outro mecanismo de processamento de mensagens consiste em usar o modelo de eventos, onde o MOM chama uma função específica associada à fila de saída toda vez que uma mensagem for colocada nesta. Nos três casos, a responsabilidade de remover a mensagem da fila de saída após o processamento cabe à aplicação que está tratando tal mensagem.

Tolerância a falhas na rede
O mecanismo de transmissão de mensagens deve ser robusto o suficiente para detectar falhas de rede, problemas de *timeout* (tempo de espera excedido) e outras situações de contingência associadas ao processo de transmissão da mensagem do lado cliente para o lado servidor. Esta tolerância deve estar em consonância com a implementação

Tolerância a falhas no servidor
Os MOMs devem garantir que toda mensagem enfileirada e persistida seja entregue ao seu destino e a aplicação seja notificada da chegada da mensagem. O lado servidor, que

abriga as filas de entrada, deve ser tolerante a problemas de *power-off*, garantindo que as mensagens enfileiradas não se percam nesta ocasião. A implementação do MOM deve garantir ainda que uma mensagem só seja retirada da fila de saída quando a fila de entrada for populada com a mesma. Outra garantia que deve existir é a de que nunca uma mesma mensagem será gravada em duplicidade na fila de entrada.

Balanceamento de carga

Em muitas ocasiões, MOMs são usados para transferir informações a serem processadas por servidores "anônimos". Em outras palavras, não interessa ao transmissor da mensagem saber quem vai processá-la, e sim simplesmente saber que ela será processada. Este tipo de cenário implica que um MOM pode fazer a opção de transmitir uma dada mensagem para um endereço que esteja menos carregado de processamento naquele momento. Este mecanismo existe em alguns produtos caracterizados como MOM e é designado balanceamento de carga. O elemento transmissor e a estrutura do MOM devem ser capazes de avaliar, dentre os possíveis destinatários de uma mensagem, qual o que está mais disponível para processar tal mensagem. Existem diversos mecanismos de balanceamento de carga, e muitos deles são vendidos como produtos separados, podendo fazer parte de um contexto mais genérico de transmissão de dados. Isto abre a possibilidade de um MOM se valer do uso de um destes mecanismos para prover balanceamento de carga.

Transparência de localização

Quem transmite a mensagem não deve saber o endereço físico e exato do destinatário, mas apenas um identificador do mesmo, ou o nome do serviço. Quem efetivamente provê o endereço físico do destinatário é um mecanismo denominado de serviço de nomes, que é consultado toda vez que uma mensagem é enviada para um servidor de processamento. Este serviço, no caso do MOM prover balanceamento de carga, deve ser capaz de identificar qual o destinatário com mais disponibilidade de processamento.

Transparência de migração

Quando um servidor apto a processar mensagens entra em indisponibilidade, o MOM deve ser capaz de migrar tal serviço para outra máquina física, deixando a aplicação independente destas questões de contingência. Se for necessária tal migração, a aplicação continuará a enviar mensagens para um serviço com um nome pré-definido. A questão de alteração da tabela de endereços disponíveis para um dado serviço deve ficar a cargo do MOM.

Transparência de replicação

Muitas vezes se faz necessária a replicação dinâmica para outra máquina de um serviço que esta sendo executado em uma máquina, a fim de permitir uma maior disponibilidade de processamento de um dado serviço. Mais uma vez, a aplicação transmissora de mensagens deve ser abstraída desta questão, continuando a enviar dados para um identificador de serviço. A questão de alteração da tabela de endereços disponíveis para um dado serviço deve ficar a cargo do MOM.

Segurança – autenticação e criptografia
O mecanismo de transmissão de mensagens de um MOM deve prover serviços de autenticação e de criptografia das mensagens a serem transmitidas e recebidas. O serviço de autenticação deve garantir que uma mensagem recebida realmente seja originária de um cliente real, evitando que programas transmissores intrusos que conheçam o formato da mensagem possam enviar informações que não deveriam ser efetivamente processadas pelo serviço. O serviço de criptografia faz com que os dados a serem transmitidos sejam cifrados, a fim de não permitir o entendimento do conteúdo dos mesmos, caso eles sejam interceptados por programas de terceiros. Estes mecanismos muitas vezes são opcionais, pois representam um *overhead* no processo de transmissão e de recepção de mensagens.

Heterogeneidade
Os MOMs normalmente suportam interação entre programas desenvolvidos em linguagens de programação diferentes (heterogeneidade de linguagem), entre programas sendo executados em sistemas operacionais diferentes e entre programas que se enquadram nos dois casos supracitados. Esta característica implica no uso de padrões de tecnologia para comunicação via rede e para o formato das mensagens. Os MOMs heterogêneos clássicos suportam diversos padrões de comunicação de rede e mensagens com formato de seqüência de bytes. MOMs mais modernos implementam seus protocolos baseados em SOAP [1], que define um formato padrão de alto nível para as mensagens, que são representadas por objetos, e um protocolo comum de transmissão de dados, o http. Embora a heterogeneidade seja um diferencial e muitas de suas questões técnicas estejam bem resolvidas, problemas referentes a tipos de dados e a performance ainda persistem.

2.7.3 Atendimento aos Requisitos Não-Funcionais Básicos

Os requisitos não-funcionais básicos associados aos sistemas de *middleware* são atendidos pelos MOMs de forma bastante abrangente. Por isso, MOM é a categoria de *middleware* mais difundida comercialmente e a que mais atende aos requisitos de aplicações que requerem distribuição e troca de dados.

Comunicação de rede
Os MOMs garantem a transparência da complexidade dos mecanismos de transmissão de dados e de informações de controle através da rede, permitindo de forma absolutamente natural que clientes e servidores sejam hospedados em máquinas físicas diferentes, ligadas através de uma rede. Os MOMs comumente suportam os protocolos de comunicação mais difundidos no mercado, e mais recentemente SOAP e http. É comum um mesmo MOM suportar mais de um protocolo de comunicação.

Coordenação
O mecanismo de transmissão de dados de MOMs é assíncrono, com controle de seqüência e garantia de entrega das mensagens. MOMs mais complexos possuem mecanismos nativos de ativação, de replicação de serviços (transparência de replicação) e de balanceamento de carga.

Confiabilidade
As características de garantia de entrega das mensagens e de tolerância a falhas no cliente, no servidor e na rede fazem com que MOMs possuam uma robustez bastante alta e um nível de atendimento bastante elevado.

Escalabilidade: Os mecanismos de balanceamento de carga, as transparências de localização, de migração e de replicação fazem com que os MOMs possuam características fortes de escalabilidade. A possibilidade de replicação e de migração de serviços faz com que o uso de MOMs seja naturalmente escalável, pois é possível colocar mais poder de processamento acoplado ao conjunto de serviços já existente de uma forma simples e transparente para os usuários do MOM.

Heterogeneidade
A maioria dos MOMs suporta troca de mensagens entre aplicativos desenvolvidos em linguagens de programação diferentes e que são executados em sistemas operacionais diferentes, pois eles usam protocolos de comunicação comuns e formas padronizadas de representar mensagens e dados. Atualmente, é possível um MOM transmitir e receber objetos (mensagens de alto nível) trocados entre aplicações que foram desenvolvidas em linguagens de programação diferentes e que são executadas em sistemas operacionais diferentes. Os MOMs apresentam algumas limitações em relação aos requisitos mostrados, embora seja possível um produto desta categoria atender à maioria dos requisitos e implementar a maioria das características descritas nos tópicos anteriores. Nem todos os MOMs implementam todas as características listadas. Assim, em alguns casos, é necessário incorporar ao uso dos MOMs produtos que complementam a lista de requisitos dos usuários do dado produto. Por exemplo, um MOM pode não implementar balanceamento de carga, sendo esta funcionalidade provida por um outro software. Algumas características aparecem em um número reduzido de MOMs, como transparência de migração e de replicação.

2.7.4 Tipos de Transmissão

Esta seção descreve uma importante característica dos MOMs, os possíveis tipos de transmissão a serem utilizados nos contextos das aplicações. Na maioria dos MOMs, o projeto de assincronismo considera a existência de duas filas físicas de mensagens. Uma fila de saída, existente no transmissor, e uma fila de entrada, existente no receptor, que desempenham papéis bem definidos dentro do contexto de tratamento de mensagens.

Os MOMs típicos trabalham com assincronismo no transmissor e no receptor e implementam um dos dois mecanismos de contingência de retransmissão descritos. Qualquer forma ou necessidade diferente de implementação deve ser provida pela aplicação do MOM.

O assincronismo no transmissor e no receptor atende à maioria das necessidades funcionais associadas a trocas de mensagens, principalmente em cenários onde a mensagem assume um papel meramente informativo, e geralmente é simplesmente lida

ou gravada pelos destinatários. Alguns exemplos de aplicações desta natureza podem ser citados: sistemas de envio e de recepção de *emails*, sistemas de distribuição de notícias relacionadas por tópicos, sistemas de *messaging*, presentes em redes de telefonia celular, e uma gama de soluções que usam mensagens como uma simples portadora de dados, cujas complexidades de processamento sejam mínimas. No entanto, algumas aplicações comerciais usam MOMs para transportar dados e para solicitar processamento complexo no lado do servidor.

Sistemas de mensagens usados neste contexto normalmente necessitam de maior flexibilidade na forma de tratar mensagens, tanto no transmissor quanto no receptor. Existem casos em que o assincronismo dos dois lados não atende aos requisitos funcionais de aplicações de MOMs, e se faz necessário incorporar outros cenários de utilização do sistema de troca de mensagens no contexto de diversas aplicações comerciais.

Em algumas situações, é possível que o assincronismo no transmissor, tal qual ele foi descrito anteriormente, exista, mas que a mensagem só seja retirada da fila de saída no transmissor quando o receptor processar efetivamente a mensagem. Neste caso, ela não é colocada na fila de entrada no lado do receptor, e é encaminhada diretamente para processamento pela aplicação. Quando a mensagem é efetivamente processada, o receptor envia ao transmissor a confirmação de recebimento e este último retira a mensagem da fila de saída. Esta situação pode ser designada como assincronismo no transmissor e sincronismo no receptor.

Em outras situações, é possível que o assincronismo no receptor, tal qual ele foi descrito, exista, mas que a aplicação no lado transmissor tenha que receber uma confirmação efetiva de que a mensagem foi colocada na fila de entrada no receptor. Neste caso, a aplicação no lado transmissor aguarda a transmissão da mensagem pelo MOM, a inserção da mesma na fila de entrada no receptor e a confirmação de recepção pelo receptor da inserção da mensagem na fila de entrada. Não há filas de saída, pois a aplicação transmissora só retoma seu fluxo de controle normal quando a mensagem for efetivamente transmitida e gravada na fila de entrada do lado receptor. Esta situação pode ser designada como sincronismo no transmissor e assincronismo no receptor.

Por fim, existem casos em que deve existir sincronismo tanto no transmissor quanto no receptor. Quando ocorre esta situação, não há filas de saída nem de entrada. A aplicação transmissora solicita o envio da mensagem ao receptor, a mesma é enviada ao receptor, que a encaminha diretamente para processamento. Após o processamento ocorrer, o receptor confirma o recebimento ao transmissor, que finalmente libera o fluxo de processamento da aplicação. Esta situação pode ser designada como sincronismo no transmissor e sincronismo no receptor.

Desta forma, pode-se prever alguns cenários em que o assincronismo deve ser parcial, ou seja, apenas implementado em um dos lados transmissor ou receptor, ou até mesmo não existir. Quando isto se faz necessário, a aplicação de um MOM típico normalmente tem que prover um mecanismo que emule o assincronismo parcial ou inexistente. O ideal é

que diferentes combinações destas modalidades de transmissão de mensagens sejam providas pelo próprio MOM a ser utilizado nestes contextos.

Aplicações que utilizam *MOMs* normalmente executam funções disponibilizadas em suas *APIs* para o envio de mensagens. O primeiro passo deste processo é a inserção da mensagem na fila de saída e o retorno imediato do controle de processamento para a aplicação que solicitou o envio. Um outro processo, independente do processo que colocou a mensagem na fila de entrada, envia as mensagens da fila para os seus respectivos destinos. Assim, podemos concluir que o processo da aplicação não aguarda a transmissão da mensagem ao seu destino, apenas a inserção desta na fila de saída. Este processo de envio no transmissor pode ser designado como um assincronismo no mesmo, pois a aplicação que usa o processo de transmissão não espera que este ocorra.

O processo no transmissor que transmite as mensagens lidas da fila de saída, lê as mensagens desta fila, envia as mesmas para o receptor de destino e, se confirmada a recepção de uma dada mensagem pelo servidor, retira a mesma da fila de saída. A confirmação de chegada da mensagem no servidor implica que o mesmo deve ter recebido a mensagem, inserido a mesma em uma fila de entrada, e respondido ao cliente que conseguiu colocar a mensagem na referida fila . Em suma, a condição para que uma mensagem seja retirada da fila de saída em um transmissor é a confirmação do receptor de que ela foi inserida na fila de entrada.

O receptor possui um outro processo, independente do processo que recebe uma mensagem de um transmissor e a insere na fila de entrada, que lê uma mensagem da fila de entrada, entrega a mesma para processamento por uma aplicação que interage com o MOM, e aguarda que a aplicação confirme o efetivo processamento da mensagem para que ela seja retirada da fila de entrada. Este processo de recepção de mensagens no receptor descrito pode ser designado como um assincronismo no receptor.

Neste intervalo, é possível que algumas mensagens sejam retransmitidas pelo transmissor, mesmo já estando na fila de entrada no receptor. Para que isto ocorra, basta que o receptor não consiga enviar ao transmissor a confirmação de que a mensagem foi inserida na fila de entrada. Neste caso, pode haver duas possibilidades:

- O receptor identifica que a confirmação não chegou ao transmissor e há a garantia de que o mesmo detectou a não chegada da tal confirmação. Neste caso, a mensagem é retirada da fila de entrada, pois ela permanecerá na fila de saída no lado transmissor e será retransmitida posteriormente; e

- Cada mensagem possui um identificador único, e toda vez que o receptor vai inserir uma mensagem na fila de entrada, verifica se existe alguma outra com o mesmo identificador da nova mensagem. Em caso positivo, a nova mensagem é descartada. No caso de um reenvio desnecessário pelo transmissor, a mensagem reenviada não é inserida na fila de entrada do receptor, porque já existe na tal fila uma outra mensagem com mesmo identificador único da mensagem reenviada.

2.8 Considerações Finais

Neste capítulo, foram apresentadas as principais características atribuídas a diversos tipos de sistemas de *middleware* e os principais requisitos não-funcionais associados a tais categorias. A classificação de [50] foi adotada como referência para o desenvolvimento do estudo aqui apresentado.

A análise de como os requisitos não-funcionais apresentados são atendidos pelas categorias de sistemas de *middleware* apresentadas por [50] mostra que cada uma delas tem pontos fortes e fracos associados a cada um destes requisitos, e cada uma delas se propõe a resolver uma gama específica de problemas relacionados à interoperabilidade entre sistemas de software.

Dentre as categorias estudadas, mereceu ênfase a análise do MOM (*middleware* orientado a mensagens), objeto de estudo deste trabalho. Através da análise das características de um MOM, constatou-se que ele se presta a resolver um espectro amplo de problemas relacionados a processos de transmissão e de troca de dados entre aplicações.

As características gerais de um MOM apresentadas neste capítulo servem como base para o capítulo posterior, que vai se basear nelas para apresentar um estudo sobre um importante padrão para implementação de um MOM e para analisar quatro MOMs utilizados em aplicações comerciais sob a ótica das tais características apresentadas.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta inicialmente o padrão JMS (Java Message Service), que tem sido amplamente utilizado para a construção de MOMs. Em seguida, o capítulo analisa três projetos de sistemas de middleware orientados a mensagens.

3.1 Introdução

O presente capítulo tem o objetivo de apresentar e analisar os trabalhos relacionados à proposta da dissertação. Inicialmente, será apresentado o JMS (Java *Message Service*) [46]. Este padrão define as características e interfaces que um MOM desenvolvido em Java deve apresentar. Qualquer *middleware* que implemente o padrão pode ser facilmente utilizado por desenvolvedores Java que conheçam a especificação do JMS [47]. Além disso, estes sistemas de *middleware* podem ser facilmente utilizados no contexto do J2EE (Java 2 *Enterprise Edition*) [43], incorporados às facilidades e às implementações de infra-estrutura de um servidor de aplicações Java desenvolvido para o padrão J2EE.

Este capítulo ainda descreve as características principais de três MOMs amplamente utilizados por desenvolvedores de aplicações distribuídas [11][12][24][40][51]. O primeiro deles é o JORAM [31], um projeto de software livre. Em seguida, o MOM apresentado é o *MQ Series* [18], um software de comunicação largamente utilizado na comunicação e troca de dados de muitos sistemas de software existentes. Finalmente, o último MOM apresentado é o *Fiorano MQ* [10].

3.2 Java Message Service

O JMS (Java *Message Service*) é um conjunto de interfaces e requisitos que uma aplicação desenvolvida em Java deve implementar para representar um sistema de troca de mensagens que pode ser usado de forma padrão por grupos de desenvolvedores distintos, em aplicações distintas, mas de uma maneira uniforme.

Este conjunto de interfaces e requisitos foi especificado e é atualmente mantido e atualizado pela *Sun Microsystems*, que entende as funcionalidades associadas a um MOM como sendo de suma importância para a interoperabilidade e a comunicabilidade dos sistemas modernos.

3.2.1 Propósitos, Serviços e Características Básicas

O principal objetivo do JMS é o de prover um padrão para as funcionalidades dos sistemas de mensagens existentes, a fim de minimizar o esforço de projeto e de implementação de novos sistemas de mensagens, pois os desenvolvedores destes novos sistemas já encontrarão prontos todo o modelo associado a um sistema de mensagens e a sua respectiva especificação [6].

Outro objetivo fundamental do JMS é o de criar uma API de uso e de manipulação de um sistema de mensagens que seja independente do fabricante que irá implementar a API. Assim, usuários desta API podem trocar de implementação sem alterar o uso do sistema de mensagens nas suas aplicações [6].

Por fim, outro objetivo importante do JMS é o de estabelecer dois domínios distintos onde um sistema de mensagens se aplica: o domínio de mensagens ponto-a-ponto e o domínio de mensagens *publish-subscribe*. A especificação padrão do JMS prevê a implementação de, pelo menos, um destes dois domínios. O JMS não inclui na sua especificação alguns pontos relacionados a um sistema de mensagens típico. Estes pontos são implementados ou não de acordo com as necessidades específicas de cada fabricante ou desenvolvedor dos MOMs.

Uma das características gerais dos MOMs que não consta na especificação do JMS é a previsão de um padrão ou até de uma implementação obrigatória do balanceamento de carga e da tolerância a falhas. O JMS não especifica nenhuma API para administração do ambiente de utilização do sistema de mensagens, nem requer que um esquema de administração seja implementado de forma obrigatória. Dentro do propósito de administração, podemos incluir outro item não mencionado no padrão JMS, que é a definição de um padrão para notificação e aviso de erros.

O conjunto de padrões do JMS não inclui nem define um protocolo padrão específico para a transmissão das mensagens que o sistema deve utilizar. A única e óbvia exigência do JMS é que um padrão de comunicação seja escolhido para transmissão e recepção das mensagens. Outro tópico relacionado ao uso de tecnologias e que não é mencionado no padrão JMS é a forma de armazenamento das mensagens que o sistema deve utilizar. Para o JMS, deve ser escolhido um mecanismo que garanta a persistência das mensagens eventualmente enfileiradas.

O JMS foi projetado para ter relações diretas e estreitas com outras APIs e padrões definidos para o Java, a fim de facilitar a incorporação de serviços adicionais e complementares ao uso de um sistema de mensagens dentro de um ambiente integrado de ferramentas e soluções, tipicamente um servidor de aplicação Java padrão J2EE [26][33]. Alguns serviços essenciais a soluções modernas de sistemas podem ser diretamente acoplados às funcionalidades de um MOM padrão JMS executado dentro do contexto de um servidor de aplicação J2EE típico [41].

Um destes serviços essenciais é o acesso ao banco de dados, que é provido no Java pelo padrão JDBC (Java *Database Connectivity*) [52]. Em alguns casos, aplicações podem necessitar incluir em uma mesma transação operações realizadas pela API JDBC e pela API JMS. Isto é possível se os clientes forem implementados como componentes EJB (*Enterprise* Java *Beans*) [52]. É possível fazer isto também através do uso da JTA (Java *Transaction* API) ou do JTS (Java *Transaction Service*) [42].

Outro serviço essencial em um servidor de aplicações J2EE é a comunicação de componentes de serviços ou de dados, os *Enterprise* Java *Beans*. Tais componentes podem usar sessões JMS para enviar e receber mensagens e o padrão de *Enterprise* Java *Beans* suporta, em conjunção com outras APIs Java, participação do JMS em transações gerenciadas por *beans* ou por *containers* [42].

Um serviço adicional importante é o de controle de transações distribuídas, que no contexto J2EE pode ser realizado pelos padrões JTA e JTS [42]. Um cliente JMS pode participar de uma transação distribuída coordenada por uma implementação da JTA ou de JTS. Opcionalmente, a própria implementação da JMS pode prover controle de transações distribuídas [4].

Outro serviço bastante utilizado no contexto de servidores de aplicação é um serviço de nomes, que no padrão J2EE é representado pelo padrão JNDI (Java *Naming and Directory Interface*) [27]. No padrão JMS, um serviço de nomes deve ser implementado e segue o padrão definido na especificação do padrão JNDI.

O JMS se integra e utiliza recursos de diversas APIs ligadas à especificação do J2EE. Assim, o J2EE suporta o uso de uma API JMS dentro do contexto de um servidor de aplicações Java. Para que este suporte seja efetivo, a implementação do JMS deve contemplar alguns pontos adicionais aos que estão descritos na especificação do JMS. Um deles é o suporte simultâneo a ambos os domínios de aplicabilidade do JMS: ponto-a-ponto e *publish-subscribe* [49].

O JMS propõe um modelo de especificação de mensagens e apresenta três conjuntos de interfaces a serem implementadas por um sistema de mensagens. O primeiro conjunto define interfaces que permitem os seus usos nos dois domínios de aplicabilidade. O segundo conjunto define uma série de funcionalidades e padrões para o domínio de aplicabilidade ponto-a-ponto. O terceiro conjunto define uma série de funcionalidades e padrões para o domínio de aplicabilidade *publish-subscribe*. Existe uma correspondência unívoca entre as interfaces definidas nos três grupos, de forma a permitir que as interfaces de uso comum possam ser utilizadas de forma independente do domínio de aplicabilidade [47].

Embora tais domínios representem semânticas diferentes, é possível que as aplicações das interfaces usem o conjunto que representa as interfaces comuns, pois o tratamento efetivo de uma mensagem recebida e a entrega de uma mensagem a ser enviada são processos independentes da forma de propagação da mesma. A especificação do JMS recomenda, sempre que possível, a utilização das interfaces comuns pelas aplicações [47].

3.2.2 Interfaces e Papéis

O esquema de funcionamento de um contexto JMS se baseia no fato de que um serviço de nomes JNDI será usado para registrar endereços e tópicos. A especificação do JMS não determina como as referências dos serviços JMS serão inseridas no serviço de nomes, e pressupõe que uma ferramenta administrativa da implementação JMS faça este papel. As aplicações do JMS usam este serviço de nomes para identificar os endereços para onde mensagens serão transmitidas, estabelecendo uma conexão lógica com a implementação JMS.

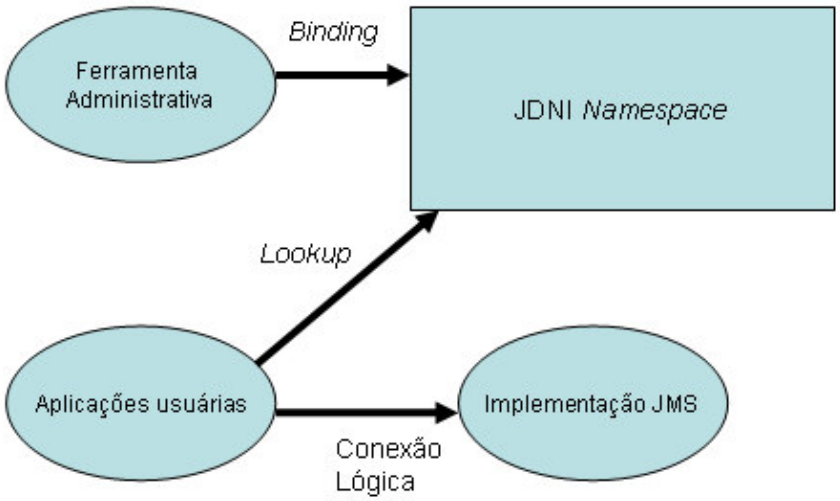

Figura 3.1 – Esquema Geral do JMS

A Figura 3.1 mostra o esquema descrito. O resultado de um *lookup* no servidor de nomes é uma conexão lógica estabelecida entre a aplicação da implementação JMS e esta última. Esta conexão é representada por uma interface Java e pode ser considerada como ponto de partida para os esquemas de transmissão e de recepção de mensagens.

A forma de popular o JNDI com informações sobre localização de serviços, funções e tópicos deve ser definida por cada implementação JMS em particular. O importante é o papel que o serviço JNDI desempenha: estabelecer uma conexão lógica entre uma aplicação e uma implementação JMS.

A lista de interfaces comuns pressupõe um modelo de relacionamentos entre tais interfaces e a definição dos papéis das mesmas. A Tabela 3.1 mostra as interfaces comuns e seus correspondentes unívocos definidos para os domínios de aplicabilidade ponto-a-ponto e *publish-subscribe*. Cada uma destas interfaces tem um papel definido no processo de transmissão e de entrega de uma mensagem em um contexto JMS.

Tabela 3.1 – Interfaces JMS

Interfaces JMS Comuns	**Interfaces JMS Ponto-a-Ponto**	**Interfaces JMS *Publish-Subscribe***
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnectionFactory`
`Connection`	`QueueConnection`	`TopicConnection`
`Destination`	`Queue`	`Topic`
`Session`	`QueueSession`	`TopicSession`
`MessageProducer`	`QueueSender`	`TopicPublisher`
`MessageConsumer`	`QueueReceiver` `QueueBrowser`	`TopicSubscriber`

Os papéis das interfaces específicas para os domínios de aplicabilidade ponto-a-ponto e *publish-subscribe* têm similaridade com os definidos para as interfaces comuns na Tabela 3.2. Ou seja, o papel que um objeto do tipo `Session` tem em um contexto de implementação JMS com as interfaces comuns é o mesmo que outro objeto do tipo `QueueSession` desempenha em um contexto de implementação JMS ponto-a-ponto. Além dos papéis desempenhados isoladamente, as interfaces descritas anteriormente se relacionam umas com as outras, e por isso possuem relações de dependência e de operacionalidade definidas pelo padrão JMS.

Tabela 3.2 – Interfaces Comuns JMS e Papéis

Interface	Papel
`ConnectionFactory`	Usada por uma aplicação para criar uma conexão.
`Connection`	Uma conexão lógica ativa com uma implementação JMS.
`Destination`	A representação de um objeto que encapsula a identidade de um destinatário.
`Session`	Um contexto mono-processo (ou *single-thread*) preparado para receber e enviar mensagens.
`MessageProducer`	A representação de um objeto criado por uma `Session` que é usado para transmitir mensagens a um destinatário.
`MessageConsumer`	A representação de um objeto criado por uma `Session` que é usado para receber mensagens enviadas a um destinatário.

A Figura 3.2 mostra os relacionamentos entre as interfaces do padrão JMS. Cada uma destas interfaces possui um conjunto de métodos que devem ser implementados de acordo com a especificação JMS e com as forma de cada aplicação JMS lidar com os conceitos apresentados.

Um elemento novo surge no contexto JMS: a interface `Message` aparece como resultado da criação de uma `Session` e representa uma mensagem a ser enviada por um `MessageProducer` a um `Destination` e recebida por um `MessageConsumer`.

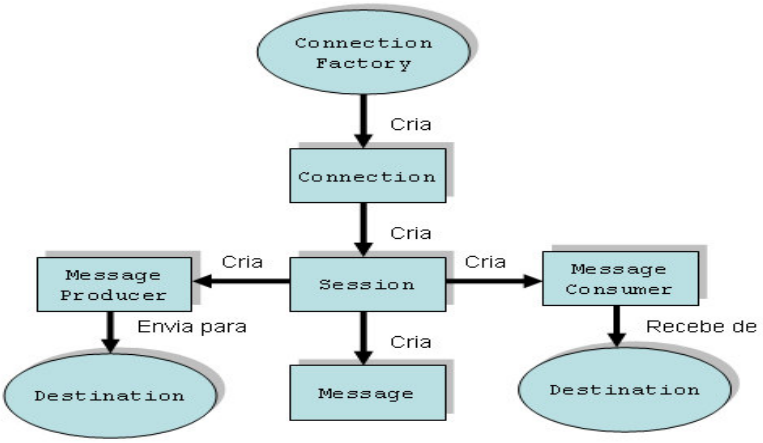

Figura 3.2 – Relacionamentos entre as Interfaces JMS

3.2.3 Modelo de Mensagens JMS

O modelo de mensagens do JMS procura definir características e premissas que possibilitem a um sistema de mensagens tratar o processo de transmissão das mesmas de forma consistente e completa. Ao mesmo tempo, o modelo proposto estabelece alguns pontos importantes a serem encontrados em uma API genérica de transmissão de mensagens.

Dois destes pontos dizem respeito à natureza de identificação e ao formato da mensagem. Unicidade na representação das mensagens e possibilidade de criar mensagens com formatos usados por APIs que não implementam o padrão JMS são duas premissas básicas a serem seguidas no modelo de representação de mensagens no padrão JMS.

O suporte a aplicações heterogêneas é outro ponto que deve ser contemplado por um modelo de mensagens no padrão JMS. Tal modelo deve permitir que um MOM JMS receba e envie mensagens para MOMs escritos em outros padrões e em outras linguagens. Para isto, o padrão estabelece que mensagens JMS devem poder conter documentos XML (*Extended Markup Language*) válidos. A compatibilidade de aplicações Java passa pelo fato delas trocarem mensagens que representem objetos Java válidos. Assim, o JMS exige que o modelo de mensagens suporte conteúdo de objetos Java.

O JMS estabelece, além destas premissas, um conjunto de características de conteúdo que o objeto que implementa a interface `Message` deve ter para representar uma mensagem JMS. Estas características permitirão à implementação JMS tratar de forma conveniente uma mensagem. Basicamente, uma mensagem JMS tem três partes distintas: um cabeçalho (*Header*), uma parte reservada ao conteúdo de propriedades da mensagem (*Properties*) e uma terceira parte onde se encontra o conteúdo da mensagem (*Body*).

No *Header* são especificadas informações gerais sobre a mensagem: o destino de envio, o modo de entrega, o identificador único, a data / hora na qual a mensagem foi entregue para envio, um eventual identificador de correlação contendo o identificador de uma mensagem correlacionada, um campo de *reply to*, um indicador de mensagem reenviada, o tipo, o tempo de expiração e a prioridade de processamento.

No *Properties*, podem ser acrescentados pares de identificadores e valores, modelo similar ao esquema de *resource bundle* do Java, onde um valor é associado a uma constante em um arquivo de configuração. Na prática, este mecanismo permite que sejam adicionadas mais informações de controle com semânticas próprias, em complemento às já definidas no *Header*.

O JMS especifica obrigatoriedade de conversão de tipos dos valores segundo uma tabela pré-definida na especificação. No *Body*, o conteúdo da mensagem deve ser armazenado. O JMS prevê cinco tipos distintos de conteúdo de uma mensagem, mostrados na Tabela 3.3:

Tabela 3.3 – Tipos de Mensagens JMS

Tipo de Mensagem	Formato do Conteúdo
`StreamMessage`	O corpo contém uma seqüência de valores Java primitivos. Deve ser lido seqüencialmente.
`MapMessage`	O corpo contém um conjunto de pares onde os nomes são *Strings* e os valores tipos primitivos Java. Cada um destes pares pode ser acessado randomicamente através do nome ou de forma seqüencial, através de uma enumeração. A ordem das entradas é indefinida.
`TextMessage`	O corpo contém uma *String* Java. É neste tipo de corpo que deve se inserir documentos XML.
`ObjectMessage`	O corpo contém um objeto Java serializado.
`BytesMessage`	O corpo contém uma seqüência de *bytes* sem significado definido.

3.3 JORAM

O JORAM (*Java Open Reliable Asynchronous Messaging*) [31] é um MOM desenvolvido para a troca de mensagens assíncronas Java.

3.3.1 Arquitetura e Características Gerais

O JORAM implementa o padrão JMS, portanto ele pode ser usado em qualquer servidor de aplicação Java que implemente o padrão J2EE e o seu uso por equipes de desenvolvimento é bastante facilitado, pois é suficiente conhecer o padrão JMS para usar o JORAM como cliente de transmissão e como um servidor de mensagens.

O uso do JORAM se baseia na definição de uma configuração de plataforma e do uso das funcionalidades clientes e servidoras do *middleware*. A configuração da plataforma consiste em definir um ou mais nós servidores JORAM, que poderão estar interconectados. Cada servidor deve ter um conjunto de serviços nele disponível e os clientes devem solicitar a tais servidores a execução dos citados serviços [31].

Assim, um serviço JORAM é um processo Java (normalmente executado em uma máquina virtual) que provê as funcionalidades de um sistema de mensagens e a entrega das mensagens aos *hosts* de destino. Por sua vez, um cliente JORAM é um processo Java (também executado em uma máquina virtual) que usa as funcionalidades do sistema de mensagens através das interfaces JMS. As implementações destas interfaces, por sua vez, se conectam a um servidor JORAM [31][40].

A Fgura 3.3 mostra um servidor JORAM, as suas características e a sua interação com clientes JORAM. Um servidor JORAM pode hospedar quatro tipos distintos de serviços, sendo todos estes usados pelas implementações das interfaces JMS usadas nos clientes

JORAM. Estes tipos de serviços dizem respeito à forma e aos meios de conexão entre duas aplicações que usam clientes JORAM e que trocam mensagens entre si [22][31]:

- Serviço de nomes, usado internamente pelas rotinas do JORAM. Deve existir pelo menos no servidor zero (principal) de uma plataforma JORAM;

- Serviço de administração de tópico, que recebe mensagens administrativas e realiza tarefas administrativas (criação de destinos, de usuários, etc..);

- Serviço de conexão, que permite que clientes JORAM se conectem através de conexões JMS. Em qualquer servidor JORAM, deve haver pelo menos um serviço de conexão disponível; e

- Serviço JNDI disponibiliza uma implementação JNDI de nomes para clientes JORAM. Este padrão de nomes não é obrigatório no uso do JORAM, portanto um serviço desta natureza só deve existir se clientes forem usar o padrão JNDI.

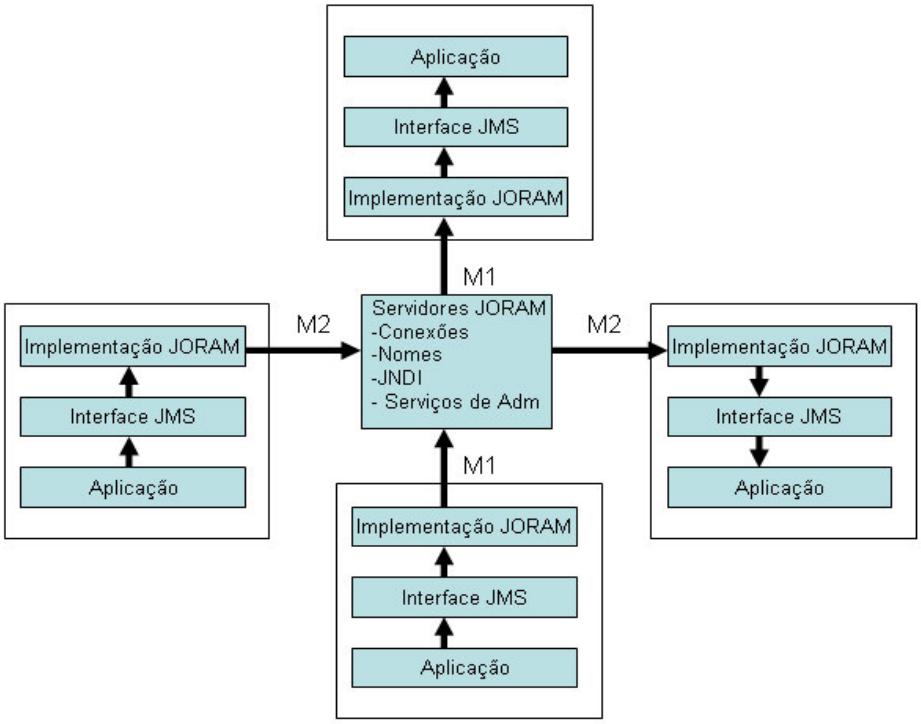

Figura 3.3 – Arquitetura Geral do JORAM

Na Figura 3.3 mostrada, pode-se notar que um servidor JORAM, além de tarefas administrativas, faz o papel de enviar a mensagem de um cliente para um destinatário específico, identificado através de um serviço de nomes JNDI ou não JNDI. O cliente interage com o servidor de nomes para identificar o endereço do serviço onde está hospedada a fila com a qual ele irá interagir. Neste contexto, o servidor é o hospedeiro efetivo das filas de mensagens, cada uma das quais relacionadas com um tópico específico. A estrutura do JORAM permite que uma plataforma funcione nos dois domínios de aplicabilidade definidos para o JMS [31].

O modo ponto-a-ponto, ilustrado na Figura 3.4, é caracterizado pelo fato de que as transmissões de mensagens se dão sempre entre um cliente e um único servidor. Tais mensagens são colocadas em filas e associadas a tópicos. Para cada tópico, existe uma fila única de mensagens, alimentada pelo cliente (produtor) e processada pelo servidor (consumidor). O produtor envia as mensagens para um servidor JORAM, que se encarrega de colocar a mensagem na fila correspondente e encaminhar a mesma para o consumidor.

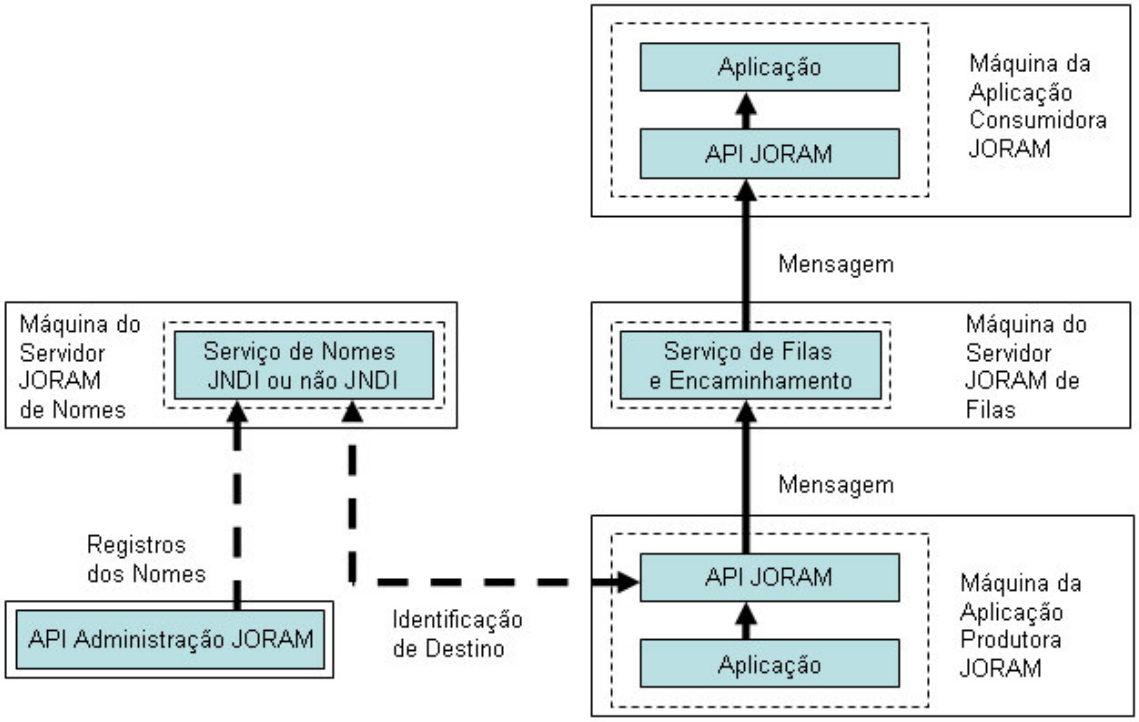

Figura 3.4 – Esquema Ponto-a-Ponto do JORAM

A Figura 3.4 mostra que a entrega de mensagens no modo ponto-a-ponto é feita através da entrada e da saída de mensagens em filas administradas por um único servidor JORAM. É possível a própria aplicação que inicializa um servidor JORAM registrar automaticamente os serviços disponíveis no contexto ou até mesmo programar cada serviço para se registrar quando for inicializado.

No modo *publish-subscribe*, ilustrado na Figura 3.5, o servidor JORAM cadastra diversos consumidores de mensagens como potenciais receptores de mensagens associadas a tópicos específicos. Quando algum cliente envia uma mensagem associada a um dado tópico, esta é encaminhada a todos os consumidores cadastrados para receber mensagens do tópico. As filas de mensagens continuam organizadas por tópicos, mas o modo de encaminhamento é diferente, pois o cliente não especifica o destinatário e sim apenas o tópico. O servidor JORAM tem a lista de consumidores cadastrados para receber mensagens de determinados tópicos e encaminha as mensagens a quem de direito.

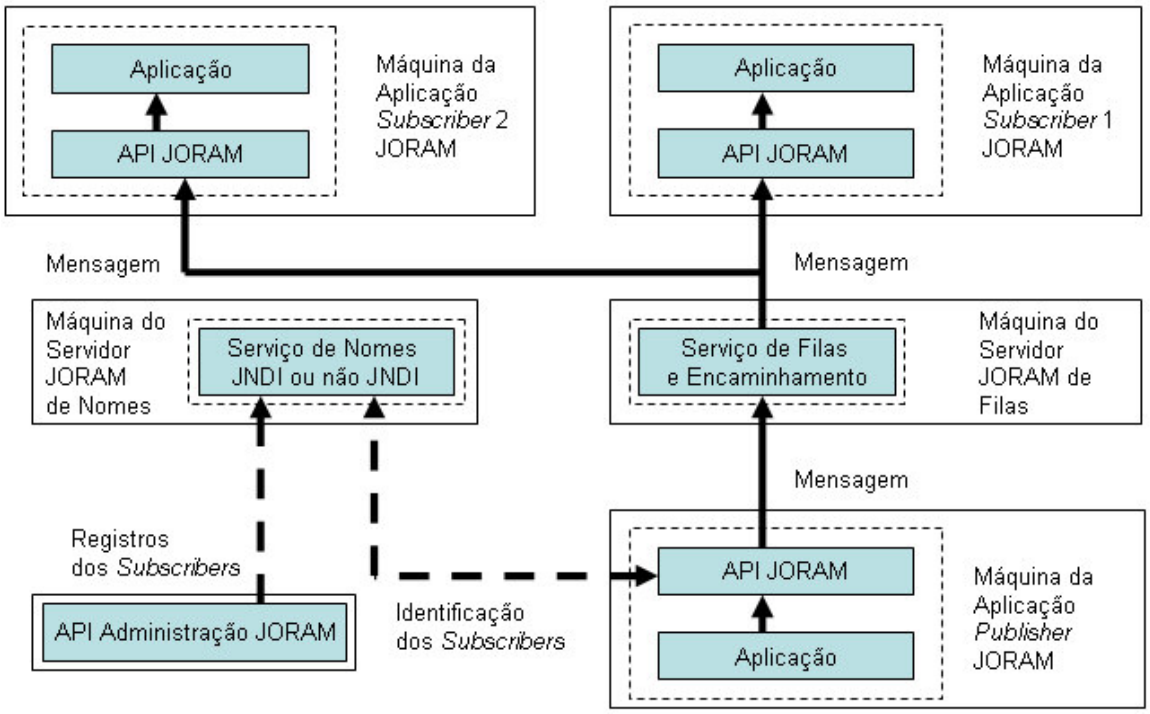

Figura 3.5 – Esquema *Publish-Subscribe* do JORAM

A Figura 3.5 mostra que a entrega de mensagens no modo *publish-subscribe* é feita de um elemento que "publica" a mensagem na fila de tópicos para "N" elementos que se registraram, a fim de receber mensagens referentes ao tópico. O registro dos *subscribers* é feito pela API de administração do JORAM, que pode ser executada para realizar tais registros toda vez que uma aplicação subscriber é inicializada.

3.3.2 Implementação das Características Gerais de um MOM

O JORAM implementa a maioria das características apresentadas na Seção 2.7, que menciona as características de um *middleware* orientado a mensagens [31].

Enfileiramento e sincronismo das mensagens
O JORAM disponibiliza sincronismo e enfileiramento de mensagens através da implementação das filas nos servidores JORAM. O sincronismo de entrega é coordenado pelo servidor JORAM, mas possui implementações de sincronismo no *middleware* e nos eventuais consumidores das mensagens.

Assincronismo de transmissão e de processamento das mensagens
Filas coordenadas pelos servidores JORAM permitem que as funcionalidades de assincronismo de processamento e de transmissão sejam apreciadas pelos usuários do JORAM. É importante notar que o assincronismo se dá no servidor JORAM, pois ele é o coordenador das filas de mensagens.

Persistência das mensagens enfileiradas
Os servidores JORAM podem ser usados em uma modalidade que as mensagens não são persistidas nas filas. Neste caso, falhas de execução do servidor JORAM fazem com que as mensagens armazenadas nas filas na hora da ocorrência do problema sejam perdidas. Em modo de persistência de mensagens, os servidores JORAM garantem a entrega de mensagens enfileiradas mesmo depois de um desligamento forçado ou voluntário. A implementação de persistência das mensagens usa recursos próprios e não permite escolha ou seleção do mecanismo de persistência.

Tolerância a falhas no cliente
No JORAM, o enfileiramento de mensagens é controlado pelo servidor JORAM. Assim, se um cliente conseguir entregar uma mensagem a uma das instâncias destes serviços, a entrega da mesma ao destinatário está garantida.

Filas e tópicos
A política de tópicos no JORAM segue o padrão JMS. Cada tópico é associado a uma fila de mensagens e na hora do envio de uma mensagem a um ou mais destinatários, esta é colocada na fila associada ao tópico. Qualquer necessidade de se associar automaticamente filas a tipos de mensagens ou a destinatários específicos deve ser tratada pela aplicação do JORAM, pois é esta que deve definir tópicos com estas semânticas, já que só é possível definir e trabalhar com tópicos de significado geral definidos e posteriormente associados a mensagens.

Formas de tratamento das mensagens
O JORAM permite que as mensagens sejam recebidas e tratadas de duas formas. A primeira forma prevê que um processo (em Java, pode-se considerar uma *thread*) da aplicação fique bloqueado por um método da API que aguarda a notificação de chegada de uma mensagem. Quando esta for entregue à aplicação destinatária, o processo bloqueado é reiniciado e a mensagem é disponibilizada para processamento. A segunda forma prevê que uma mensagem recém-chegada na aplicação, e associada a uma determinada fila / tópico, é entregue a um tratador específico desta mensagem, previamente associado à fila / tópico. Na prática, o JORAM notifica a aplicação da chegada da mensagem invocando um método do tratador previamente associado ao tópico da mensagem. A escolha da forma de tratamento das mensagens depende da arquitetura da aplicação que está usando o JORAM.

Tolerância a falhas na rede
A implementação deste mecanismo no JORAM é feita no lado do servidor JORAM, já que é ele que controla o fluxo de entrada e de saída das filas e o estado das conexões com os clientes e com os destinatários das mensagens.

Tolerância a falhas no servidor
O fato do JORAM trabalhar com um gerenciador de filas único, as instâncias dos servidores JORAM, faz com que a implementação de tolerância a falhas no servidor seja mais simples, já que o processo de entrada e de saída das mensagens nas filas e o fluxo de

dados nas conexões com os produtores e consumidos é controlado por apenas um único componente.

Balanceamento de carga
O JORAM realiza balanceamento de carga quando está trabalhando em modo *publish-subscribe* com utilização de *clusters* de processamento. Neste modo, "N" servidores JORAM são habilitados a tratar de forma cooperativa o encaminhamento das mensagens publicadas por clientes *publishers* aos *subscribers* daquele tópico em especial. Assim, um *publisher* pode ter o encaminhamento de sua mensagem roteado para outro servidor JORAM que faz parte do grupo de servidores cooperativos relacionados ao tópico. Este roteamento é feito internamente pelo JORAM e se baseia na carga de processamento de cada servidor cooperativo associado ao tópico da mensagem. Assim, é possível usar o JORAM com balanceamento de carga, desde que um tópico e o encaminhamento de suas mensagens relacionadas sejam associados a um ou mais *clusters* de processamento.

Transparência de localização
O JORAM trabalha com servidores de nomes. Desta forma, a transparência de localização entre transmissores e receptores de mensagens está garantida. O JNDI é uma forma mais abstrata de prover transparência de localização. Como a arquitetura do JORAM prevê que, para trabalhar em modo produtor-consumidor ponto-a-ponto deve-se ter um servidor JORAM para ligar as aplicações, a transparência de localização é provida por este servidor ou por outro que tenha um serviço de nomes disponível.

Transparência de migração
O JORAM não apresenta esta funcionalidade, embora possa trabalhar em ambientes que possuam softwares aptos a realizar migração automática de serviços e roteamento de mensagens para os serviços novos disponibilizados.

Transparência de replicação
O JORAM não apresenta esta funcionalidade, embora a associação entre *clusters* de processamento e tópicos possa ser considerada uma forma mais inteligente de balanceamento de carga e pode muitas vezes suprir necessidades de replicação de serviços. De qualquer forma, o JORAM não é capaz de replicar serviços de forma automática.

Segurança – autenticação e criptografia
O JORAM implementa uma política completa de segurança e de administração dos seus serviços, permitindo autenticação de usuários comuns e com atribuições de administrador e autenticação de mensagens. Mensagens cifradas e criptografadas devem ser previamente tratadas pela aplicação antes de serem enviadas no lado produtor e antes de serem processadas no lado consumidor. Não há mecanismo interno de criptografia no JORAM.

Heterogeneidade

Do ponto de vista de plataforma / sistema operacional, o JORAM suporta 100% de heterogeneidade, desde que o sistema operacional possua uma JVM. Dentre os que possuem JVM, podemos destacar: família *Windows* (95, 98, 2000, *Millenium*, XP, *Server* 2000, *Server* 2003), *Linux*, família *Unix* (HP UX, SCO, *Solaris*, etc.), OS2 e *Mac*. Do ponto de vista de plataforma / linguagem de programação, o JORAM a princípio troca mensagens ente aplicativos Java, mas está preparado para trocar mensagens com programas escritos em outras linguagens, pois ele suporta dois protocolos: TCP [28] e SOAP [1].

É possível trabalhar com o JORAM em modo SOAP, permitindo desta forma que todas as mensagens trocadas entre aplicativos tenham o formato de XML no padrão SOAP estabelecido. Trabalhar desta forma tem um custo de performance, pois o padrão SOAP exige uma carga maior de conversão de objetos em XMLs [8] na saída e de XML em objetos na entrada.

Além das características citadas, o JORAM possui algumas funcionalidades úteis e adicionais que podem ser usadas em contextos e necessidades mais específicas. Uma das funcionalidades adicionais é a capacidade de criar uma hierarquia de tópicos no JORAM.

A criação da hierarquia permite que mensagens associadas a tópicos em níveis mais altos de uma hierarquia de tópicos sejam enviadas a todos os subscritos em tópicos pertencentes aos sub-niveís da hierarquia. Esta forma de organizar a relação entre tópicos permite uma maior facilidade de seleção e de distribuição de mensagens.

Outra funcionalidade adicional do JORAM é a capacidade de armazenar mensagens "mortas" em filas. Mensagens "mortas" são as consideradas impossíveis de serem entregues aos destinatários por diversas razões, estabelecidas pelo JORAM. Estas razões estão relacionadas com a própria existência do destinatário, com a permissão que os transmissores têm de escrever mensagens e com o prazo de expiração de mensagens. As filas de mensagens mortas podem ser acessadas diretamente pela API do JORAM, que permite à aplicação uma completa gerência das mensagens não enviadas.

Além disso, os critérios de mensagens mortas podem ser úteis para aplicações que precisem gerenciar de forma mais precisa contingências de transmissão e para aplicações que necessitam ter um "*log*" histórico de mensagens não enviadas. O JORAM permite a definição de mensagens com prioridades de envio e de processamento diferentes, possibilitando aos seus usuários a priorização de tarefas, de processamento e sinalização de emergência entre aplicações. As funções administrativas estão disponíveis na própria API do JORAM. Embora as rotinas de administração estejam prontas, elas têm que ser executadas através de linhas de comando Java.

Outras rotinas de manipulação e configuração mais avançadas têm que ser programadas a partir de uma classe Java simples que usa a API do JORAM. Nos dois casos, o programador tem que conhecer a semântica de algumas classes, interfaces e métodos do

JORAM, além de ter de saber o significado de *tags* XML e de propriedades de alguns arquivos de configuração necessários ao funcionamento do JORAM.

O JORAM possui uma versão *"light"*(o kJORAM), que permite a sua utilização em contextos de aplicações embarcadas. A implementação desta versão usa as classes da plataforma J2ME (Java 2 *Micro Edition*) e pode interagir com servidores JORAM implementados na versão convencional. É possível destacar algumas diferenças e semelhanças entre o JORAM e o *Hermes*, objeto de estudo deste trabalho.

O *Hermes* apresenta características similares ao JORAM com respeito a assincronismo, persistência das mensagens de uma forma proprietária, tolerância a falhas no cliente e no servidor e transparência de localização e heterogeneidade – os dois softwares suportam SOAP (*Simple Object Access Protocol*) e são desenvolvidos na linguagem de programação Java.

As diferenças principais surgem nas partes de tratamento e de transmissão das mensagens, onde o *Hermes* é mais flexível e encapsula uma gama maior de opções para troca de mensagens entre clientes e servidores, que serão explicadas e detalhadas no capítulo seguinte. Outras diferenças entre o *Hermes* e o JORAM podem ser observadas nas características relativas à transparência de migração e à segurança. O *Hermes* possui uma implementação simples de transparência de migração e não possui mecanismos de criptografia de mensagens, apenas autenticação de um esquema próprio e simples de reconhecimento de mensagens.

3.4 MQ Series

O *MQ Series* [18] foi desenvolvido pela IBM e possui uma vasta lista de características, extensões e aplicabilidades. O *MQ Series* é um sistema de troca de mensagens assíncronas, desenvolvido em C e é utilizado em projetos críticos por diversas instituições corporativas, fazendo parte de projetos de softwares e realizando integrações de sistemas novos com sistemas legados, principalmente os desenvolvidos em *mainframes* [51].

3.4.1 Arquitetura e Características Gerais

O *MQ Series* é um *middleware* projetado para ser usado como uma API, dentro de várias linguagens de programação e para ser executado sob diversos sistemas operacionais. O uso do *MQ Series* baseia-se na instanciação de um serviço MQ que atua como cliente (transmissor) e servidor (receptor) de mensagens assíncronas, colocadas em filas, que por sua vez são associadas a tópicos. Para cada servidor MQ, deve-se definir os tópicos e conseqüentemente as filas que tal servidor irá prover [25].

Este serviço MQ é um programa executado em uma máquina e provê todas as funcionalidades de transmissão de mensagens disponíveis no MQ. Esta configuração

pressupõe a existência de um servidor MQ em cada ponto que necessite transmitir e receber mensagens [18].

No modo de operação convencional, o MQ não utiliza um servidor de nomes, sendo necessário que o transmissor da mensagem provenha o endereço da máquina destino. Desta forma, a mensagem é enviada de um cliente para um único servidor. A interação do processo de transmissão da mensagem é realizada entre os dois serviços MQ ativos nas máquinas transmissora e receptora. A aplicação que transmite a mensagem via MQ usa uma API disponibilizada pela IBM que se comunica internamente com o serviço MQ, este último faz o papel de transmitir a mensagem [18].

Da mesma forma, a aplicação que recebe a mensagem via MQ pode ser executada pelo serviço MQ que recebe a mensagem ou pode ficar bloqueada por um método da API até que uma mensagem seja colocada em uma dada fila no serviço MQ. Quando isto ocorre, este se comunica internamente com a API, que desbloqueia a aplicação e retorna a mensagem. Nos dois casos, a gerência de filas e o processo de transmissão de dados são realizados entre dois serviços MQ, executados como aplicações independentes dos programas que efetivamente tratam as mensagens. É de responsabilidade da aplicação retirar as mensagens das filas de entrada.

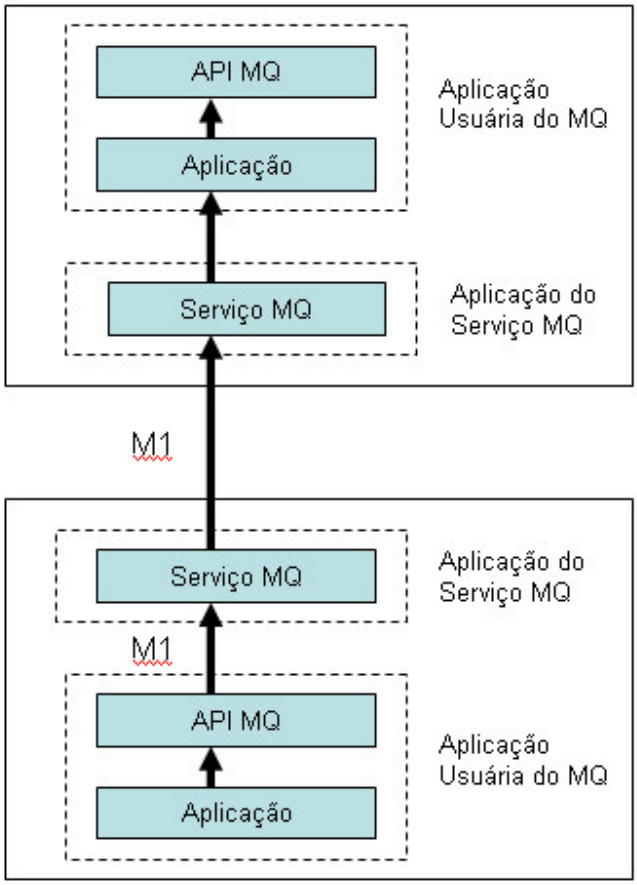

Figura 3.6 – Arquitetura Básica de Comunicação no *MQ Series*

As tarefas de autenticação de mensagens e de usuários / permissões são realizadas pelos serviços MQ, que devem estar previamente configurados para tais propósitos. Nesta modalidade de uso do *MQ Series*, as mensagens são transmitidas ponto a ponto.

Em versões mais atuais, o *MQ Series* suporta serviços de nomes, que podem estar localizados em qualquer servidor MQ ativo em uma máquina qualquer da rede onde a solução distribuída é executada e troca mensagens. Estes serviços de nomes permitem que mensagens sejam trocadas entre serviços MQ identificados por nomes, e não por endereços físicos [18].

A Figura 3.7 mostra que os serviços MQ interagem com o servidor de nomes. O serviço receptor da mensagem se registra no servidor de nomes e o serviço transmissor usa o servidor para mapear um nome em um endereço físico, a fim de transmitir a mensagem. A aplicação transmissora não precisa especificar o endereço físico do destinatário, mas apenas o nome do mesmo.

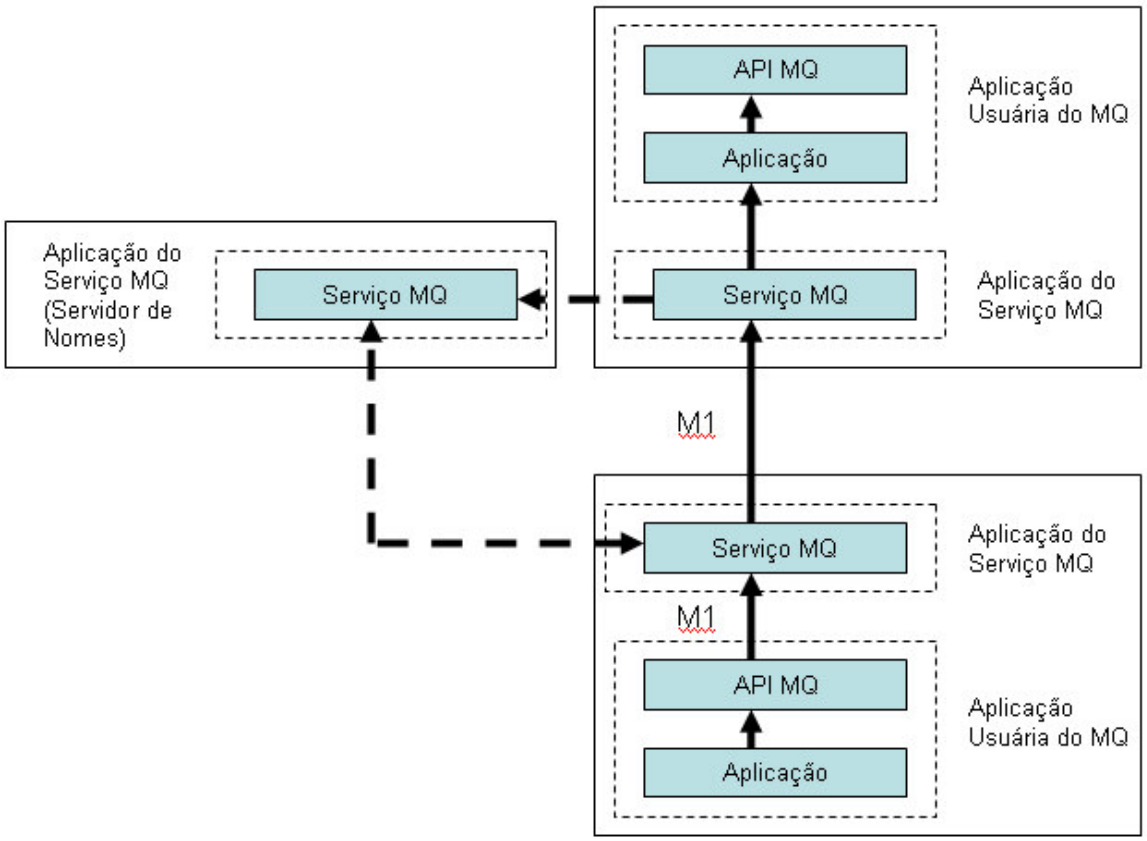

Figura 3.7 – Arquitetura do MQ *Series* com Servidor de Nomes

Fica a cargo do serviço MQ interagir com o servidor de nomes para identificar o endereço do receptor. Além das possibilidades descritas, o *MQ Series* pode funcionar no esquema *publish-subscribe*, onde um serviço MQ pode conter informações sobre a relação entre tópicos e serviços MQ subscritos aos tópicos.

Na Figura 3.8, dois serviços MQ se cadastram no servidor de lista de tópicos para receber mensagens relacionadas a um dado tópico. Um outro serviço MQ identifica, interagindo com o servidor de lista de tópicos, quais os assinantes do tópico relacionado à mensagem que será transmitida e envia a mensagem aos assinantes identificados.

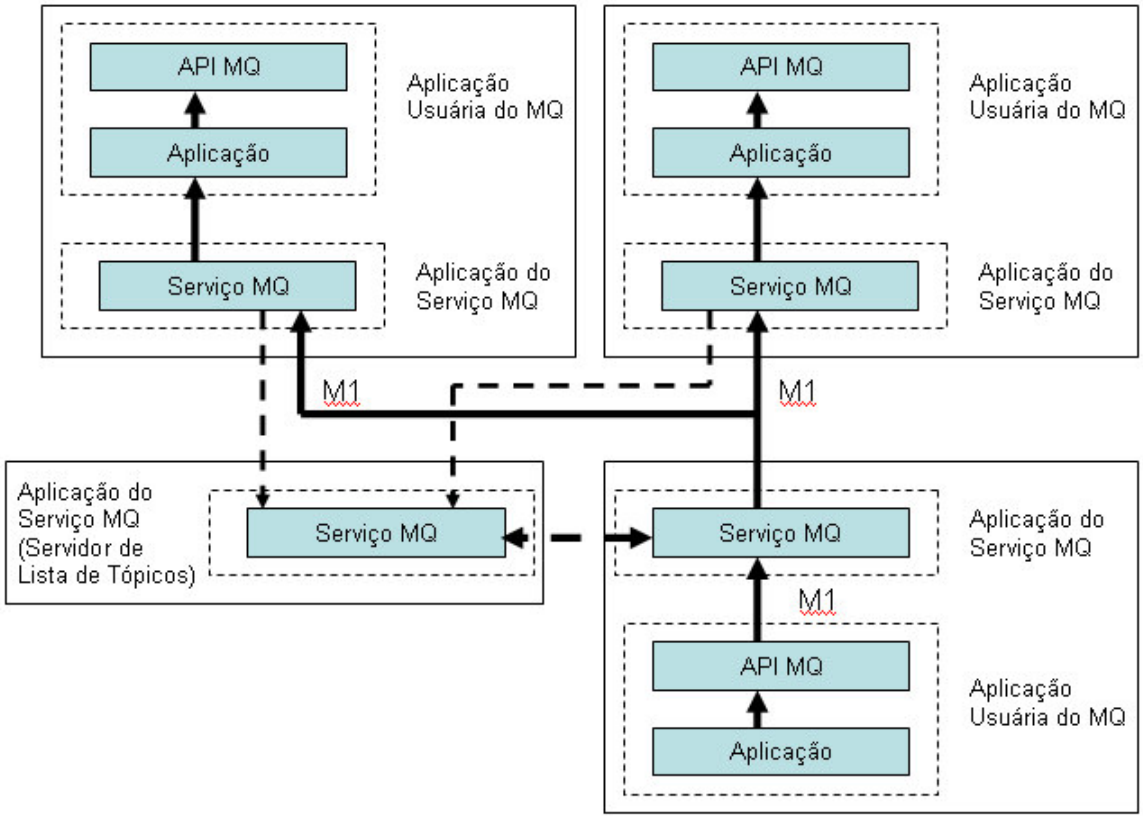

Figura 3.8 – MQ *Series* no Esquema *Publish-Subscribe*

Para programas Java, o *MQ Series* disponibiliza um conjunto de classes que implementam as interfaces do JMS. Além desta implementação, o *MQ Series* permite que seus serviços MQ sejam utilizados como servidores JNDI e disponibiliza uma API em Java para uso do MQ.

Para os usuários do *Websphere* [39], o servidor de aplicações Java da IBM, o *MQ Series* aparece como uma opção de *plug-in* ou de *add-in* totalmente integrado às funcionalidades do servidor de aplicação. Neste caso, é possível inclusive que os programas que usam o MQ para transmissão de mensagens usem as características do *Websphere* neste contexto, possibilitando controle transacional das mensagens transmitidas e outras funções adicionais.

Além disso, o *MQ Series* possui diversos *plug-ins* que incorporam características adicionais ao *middleware*. O controle transacional por exemplo pode ser conseguido através do uso integrado do *MQ Series* dentro do *Websphere* ou através da adição de um *plug-in* para controle de transações. Estas características de integrabilidade e de potencial para incorporação de serviços adicionais ao *MQ Series* foram adicionadas ao

mesmo nas suas últimas versões, pois até então o *MQ Series* era um software fechado e sem possibilidade de integração com outros serviços.

3.4.2 Implementação das Características Gerais de um MOM

O *MQ Series* implementa a maioria das características apresentadas na Seção 2.7 referente a MOMs. Nesta seção, as formas de implementação destas características no *MQ Series* serão descritas [18][25].

Enfileiramento e sincronismo das mensagens
O *MQ Series* disponibiliza sincronismo e enfileiramento de mensagens através da implementação das filas nos serviços MQ. O mecanismo do sincronismo de entrega é compartilhado entre os dois serviços MQ, existindo filas persistentes de saída no serviço MQ transmissor e filas persistentes de entrada no serviço MQ receptor.

Assincronismo de transmissão e de processamento das mensagens
Filas de saída existentes nos serviços MQ transmissores e filas de entrada nos serviços MQ receptores permitem que as funcionalidades de assincronismo de processamento e de transmissão sejam apreciadas pelos usuários do *MQ Series*. O assincronismo é de responsabilidade compartilhada entre os serviços MQ de origem e de destino.

Persistência das mensagens enfileiradas
O *MQ Series* pode ser usado em uma modalidade que as mensagens não são persistidas nas filas de saída. Neste caso, não é necessário o uso de um serviço MQ no lado do transmissor e falhas de execução da aplicação fazem com que as mensagens armazenadas nas filas de saída na hora da ocorrência do problema sejam perdidas. Se as mensagens forem persistidas, um serviço MQ deve ser usado nos lados transmissor e receptor e os serviços MQ garantem a entrega de mensagens enfileiradas mesmo depois de um desligamento forçado ou voluntário de um dos lados. A implementação de persistência das mensagens pode usar recursos próprios do MQ ou pode ser definida para as mensagens serem gravadas em um banco de dados.

Tolerância a falhas no cliente
No *MQ Series*, o enfileiramento de mensagens na transmissão é controlado pelo serviço MQ. Assim, se uma aplicação transmissora conseguir entregar uma mensagem ao seu serviço MQ correspondente, a entrega da mesma ao destinatário está garantida.

Filas e tópicos
A política de tópicos no MQ segue o padrão JMS. Cada tópico é associado a uma fila de mensagens e na hora do envio de uma mensagem a um ou mais destinatários, esta é colocada na fila associada ao tópico. Qualquer necessidade de se associar automaticamente filas a tipos de mensagens ou a destinatários específicos deve ser tratada pela aplicação do *MQ Series*, pois é esta que deve definir tópicos com estas semânticas, já que só é possível definir e trabalhar com tópicos de significado geral definidos e posteriormente associados a mensagens.

Formas de tratamento das mensagens
O *MQ Series* permite que as mensagens sejam recebidas e tratadas de três formas. A primeira forma prevê que um processo (em Java, pode-se considerar uma *thread*) da aplicação fique bloqueado por um método da API que aguarda a notificação de chegada de uma mensagem. Quando esta for entregue ao serviço MQ destinatário, o mesmo notifica a API da chegada da mensagem, o processo bloqueado é reiniciado e a mensagem é disponibilizada para processamento. A segunda forma prevê que uma mensagem recém-chegada no serviço MQ e associada a uma determinada fila / tópico seja entregue à API *MQ Series*, que associa a mensagem a um tratador específico desta mensagem, previamente associado à fila / tópico. Na prática, a API do *MQ Series* notifica a aplicação da chegada da mensagem invocando um método do tratador previamente associado ao tópico da mensagem. A terceira forma permite que o serviço MQ execute um programa quando uma dada mensagem for inserida em uma dada fila. A escolha da forma de tratamento das mensagens depende da arquitetura da aplicação que está usando o *MQ Series*.

Tolerância a falhas na rede
A implementação deste mecanismo no *MQ Series* é feita nos serviços MQ, pois estes possuem mecanismos específicos de retransmissão e de confirmação de chegada de mensagens.

Tolerância a falhas no servidor
Cada serviço MQ cuida do gerenciamento de falhas na própria execução do serviço ou de falhas na máquina onde o serviço está sendo executado.

Balanceamento de carga
O *MQ Series* pode realizar balanceamento de carga quando está funcionando integrado ao *Websphere* ou em consonância com outros softwares de mercado que realizam balanceamento de carga baseados em avaliação do nível de uso dos processadores e de roteamento de transmissão de mensagens para os processadores menos ocupados.

Transparência de localização
O *MQ Series* pode trabalhar com servidores de nomes. Desta forma, a transparência de localização entre transmissores e receptores de mensagens está garantida. O JNDI é uma forma mais abstrata de prover transparência de localização e é suportado pelo *MQ Series*, principalmente quando este trabalha integrado ao *Websphere*.

Transparência de migração
O *MQ Series* funcionando sozinho não apresenta esta funcionalidade, mas se estiver integrado ao *Websphere* pode utilizar as capacidades de migração de serviços deste software, que possibilita a transferência de uma solicitação de processamento para outro servidor quando o servidor destino original estiver indisponível.

Transparência de replicação
O *MQ Series* funcionando sozinho não apresenta esta funcionalidade, mas se estiver integrado ao *Websphere* pode utilizar as suas capacidades de replicação, que possibilita a instanciação dinâmica de um serviço novo em uma outra máquina.

Segurança – autenticação e criptografia
O *MQ Series* implementa uma política completa de segurança e de administração dos seus serviços, permitindo autenticação de usuários comuns e com atribuições de administrador e autenticação de mensagens. Mensagens cifradas e criptografadas também são opcionalmente suportadas pelo *MQ Series*, que pode ter os algoritmos de criptografia configurados de acordo com as necessidades específicas do nível de segurança desejado.

Heterogeneidade
O *MQ Series* é um dos mais versáteis sistemas de *middleware* neste sentido. Como usa a filosofia de trabalhar com serviços MQ, estes conversam entre si independente de plataforma ou de sistema operacional, e estão disponíveis em diversos sistemas operacionais, dos quais podemos destacar: família *Windows* (95, 98, 2000, *Millenium*, XP, *Server* 2000, *Server* 2003), *Linux*, família *Unix* (HP UX, SCO, *Solaris*, etc.), OS2 e alguns outros. Por outro lado, existem APIs do *MQ Series* para um sem número de linguagens de programação, das quais podemos destacar: C, C++, Java, COBOL, *Natural*, *Visual Basic*, *Delphi*, .NET (C#, *Visual Basic* .NET, C++ .NET, ASP .NET). Muitos sistemas de *middleware* implementam integração direta com o padrão MQ. Isto faz com que o padrão de mensagens MQ seja suportado por uma grande quantidade de softwares e de sistemas. Outro ponto forte de heterogeneidade do *MQ Series* é a quantidade de protocolos de comunicação que ele suporta. Dentre estes, podemos destacar: TCP, UDP [28], X-25 [17], IPX/SPX [17], http [28] e SOAP.

Além destas funcionalidades, o *MQ Series* possui um mecanismo de *log* que pode ser visualizado através de uma aplicação de monitoração de serviços MQ, mostrando o histórico de mensagens recebidas e transmitidas e o estado atual de cada fila de entrada e de saída, permitindo aos seus usuários a obtenção de informações sobre ocorrências em processos de transmissão de mensagens. As configurações do *MQ Series* podem ser feitas através de uma API de configuração e de montagem do esquema de filas e de protocolos, onde os comandos são executados em um contexto *MQ Series* similar a um programa de FTP. Alternativamente, existem programas gráficos que permitem ao usuário montar um esquema de filas e protocolos, associando tal esquema a serviços MQ.

O *MQ Series* permite a definição de mensagens com prioridades de envio e de processamento diferentes, possibilitando aos seus usuários a priorização de tarefas, de processos e sinalização de emergência entre aplicações. É possível destacar algumas diferenças e semelhanças entre o *MQ Series* e o *Hermes*, objeto de estudo deste trabalho.

O *Hermes* apresenta características similares ao *MQ Series* com respeito ao assincronismo, às tolerâncias às falhas no cliente e no servidor, à transparência de localização. As diferenças principais surgem nas partes de tratamento e de transmissão das mensagens.

O *Hermes* é mais flexível e encapsula uma gama maior de opções para troca de mensagens entre clientes e servidores, que serão explicadas e detalhadas no capítulo seguinte. O *MQ Series* trata a persistência de mensagens de uma forma aberta, onde o usuário pode optar pelo mecanismo de persistência a ser utilizado. No *Hermes*, esta característica é própria do software, embora o projeto permita que opções adicionais relativas às formas de persistência possam ser facilmente incorporadas.

Outras diferenças entre o *Hermes* e o *MQ Series* podem ser observadas nas características relativas à transparência de migração, à transparência de replicação a ao balanceamento de carga. Todos estes mecanismos no *MQ Series* são associados à utilização integrada com o servidor de aplicações *Websphere*. A segurança no *MQ Series* é extremamente elaborada e flexível, sendo possível até determinar qual algoritmo de criptografia será utilizado na encriptação de mensagens. O *Hermes* não possui mecanismos de criptografia.

Com relação à heterogeneidade, o *Hermes* e o *MQ Series* têm flexibilidades similares, embora implementadas de formas diferentes. O *MQ Series* disponibiliza uma versão de software para cada sistema operacional e uma API Java que encapsula a implementação em C do *MQ Series*. O *Hermes* é naturalmente multi-plataforma, pois é implementado nativamente em Java. Quanto à natureza das mensagens, o suporte ao SOAP por ambos garante uma possível interação com outros softwares de comunicação ou de troca de mensagens.

O *MQ Series* ainda apresenta uma suíte completa de ferramentas administrativas, permitindo a gerência centralizada de diversos servidores MQ, cada qual implementando diversas filas de mensagens. O *Hermes* possui uma estrutura básica de administração, até porque a sua forma de utilização é bem mais simples do que a do *MQ Series*.

3.5 Fiorano MQ

O *Fiorano MQ* [10] é um MOM projetado para prover uma solução completa para comunicação entre aplicativos baseada em troca de mensagens.

3.5.1 Arquitetura e Características Gerais

O *Fiorano MQ* implementa o padrão JMS, portanto ele pode ser usado em qualquer servidor de aplicação Java que implemente o padrão J2EE e o seu uso por equipes de desenvolvimento é bastante facilitado, pois é suficiente conhecer o padrão JMS para usar o *Fiorano MQ* como cliente de transmissão e como um servidor de mensagens.

O contexto de utilização do *Fiorano MQ* pressupõe a existência de servidores *Fiorano MQ*, que nada mais são do que máquinas virtuais Java executando os serviços de nomes JNDI, os processos de encaminhamento de mensagens e o gerenciamento das filas. As aplicações do *Fiorano MQ* se comunicam com os servidores *Fiorano MQ* através da API

JMS previamente definida. As implementações das interfaces JMS realizam a interação entre aplicações do *Fiorano MQ* e os seus serviços [10][13].

Um servidor *Fiorano MQ* pode hospedar serviços de entrega de mensagens e ou serviços de nomes, ficando a critério do projetista definir as configurações de cada servidor a ser instanciado em um contexto de utilização do *Fiorano MQ*. Os servidores *Fiorano MQ* podem trabalhar cooperativamente a fim de disponibilizar recursos avançados de um sistema de troca de mensagens, tais como balanceamento de carga [10].

Na Figura 3.9, pode-se notar que servidores *Fiorano MQ* realizam entrega de mensagens e fazem o papel do servidor de nomes JNDI. Neste contexto, o servidor é o hospedeiro efetivo das filas de mensagens, cada uma das quais relacionadas com um tópico específico [10].

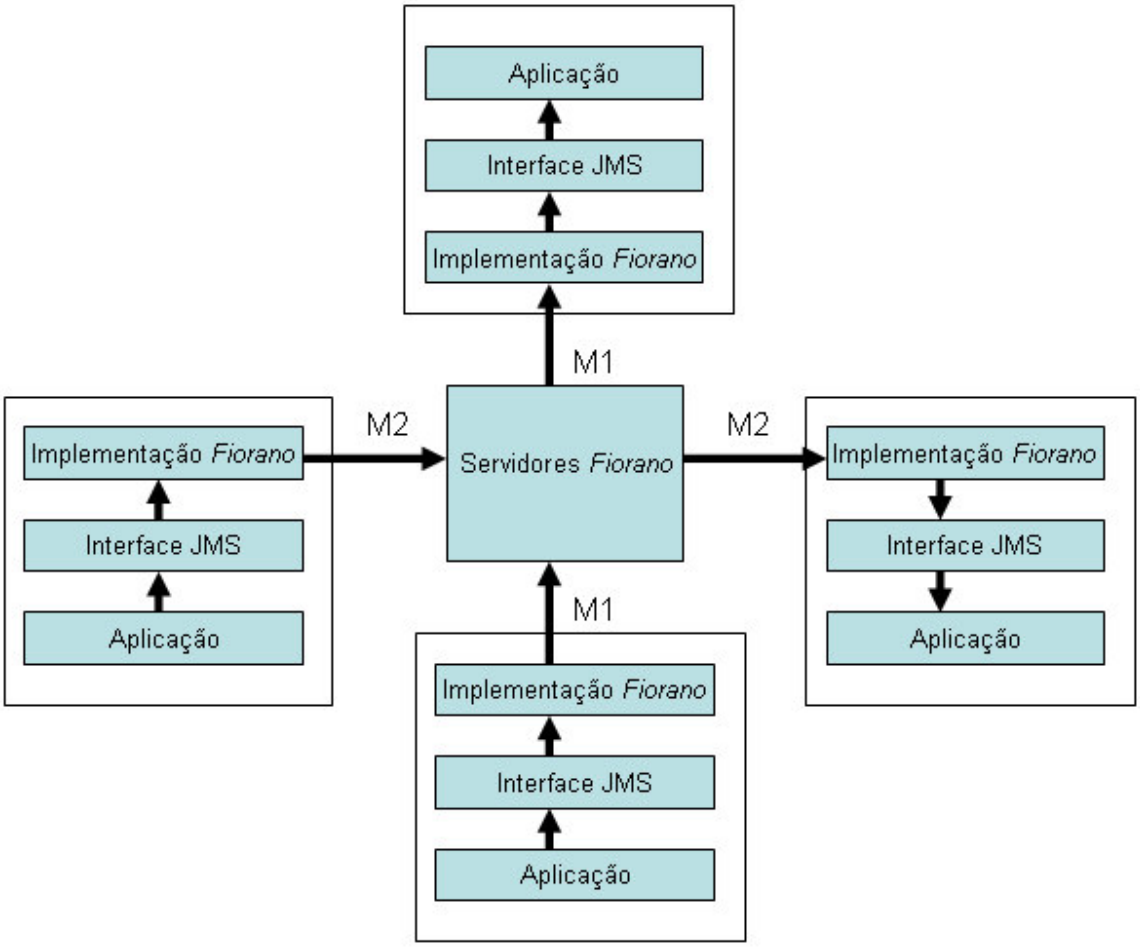

Figura 3.9 – Arquitetura Geral do *Fiorano* MQ

O cliente interage com o servidor de nomes para identificar o servidor *Fiorano MQ* hospedeiro da fila com a qual ele irá interagir. A estrutura do *Fiorano MQ* permite que um contexto de utilização funcione nos dois domínios de aplicabilidade do JMS. No modo ponto-a-ponto, funciona de maneira similar à forma ponto-a-ponto descrita para o JORAM, pois a arquitetura de funcionamento e a forma de interação das aplicações com os servidores são as mesmas.

O papel que o servidor JORAM faz em uma plataforma JORAM é o mesmo que um servidor *Fiorano MQ* faz em um contexto *Fiorano MQ*. A única diferença é que o servidor de nomes neste último *middleware* obedece ao padrão JNDI, enquanto que no JORAM é permitida a utilização de servidores de nomes que não implementam o padrão JDNI. A figura da arquitetura ponto-a-ponto para o *Fiorano MQ* é idêntica à mostrada na descrição do JORAM, trocando-se apenas os papéis dos servidores JORAM por servidores *Fiorano MQ*. No modo *publish-sibscribe*, também são guardadas as mesmas características descritas para o JORAM, trocando-se apenas o nome de servidor JORAM para servidor *Fiorano MQ* [10].

3.5.2 Implementação das Características Gerais de um MOM

O *Fiorano MQ* implementa a maioria das características apresentadas na Seção 2.7. Nesta seção, as formas de implementação destas características no *Fiorano MQ* serão descritas [10].

Enfileiramento e sincronismo das mensagens
O *Fiorano MQ* disponibiliza sincronismo e enfileiramento de mensagens através da implementação das filas nos servidores *Fiorano MQ*. O sincronismo de entrega é coordenado pelo servidor, mas possui implementações de sincronismo no produtor e nos eventuais consumidores das mensagens.

Assincronismo de transmissão e de processamento das mensagens
Filas coordenadas pelos servidores *Fiorano MQ* permitem que as funcionalidades de assincronismo de processamento e de transmissão sejam apreciadas pelos usuários do *Fiorano MQ*. É importante notar que o assincronismo se dá no servidor *Fiorano MQ*, pois ele é o coordenador das filas de mensagens. Esta coordenação é independente da aplicação produtora e das aplicações consumidoras de mensagens.

Persistência das mensagens enfileiradas
A implementação de persistência das mensagens usa recursos próprios ou apresenta a opção de persistência em bancos de dados. Caso o usuário opte por esta última modalidade de persistência, o *Fiorano MQ* permite controle transacional de persistência de dados das mensagens dentro de um escopo de processamento no consumidor e suporte a transações distribuídas desde que os bancos envolvidos suportem o protocolo 2-*Phase Commit*.

Tolerância a falhas no cliente
O enfileiramento de mensagens é controlado pelo servidor *Fiorano MQ*. Assim, se um cliente conseguir entregar uma mensagem a uma das instâncias destes serviços, a entrega da mesma ao destinatário está garantida.

Filas e tópicos
A política de tópicos no *Fiorano MQ* segue o padrão JMS. Cada tópico é associado a uma fila de mensagens e na hora do envio de uma mensagem a um ou mais destinatários, esta é colocada na fila associada ao tópico. Qualquer necessidade de associar automaticamente filas a tipos de mensagens ou a destinatários específicos deve ser tratada pela aplicação do *Fiorano MQ*, pois é esta que deve definir tópicos com estas semânticas, já que só é possível definir e trabalhar com tópicos de significado geral definidos e posteriormente associados a mensagens.

Formas de tratamento das mensagens
O *Fiorano MQ* permite que as mensagens sejam recebidas e tratadas de duas formas. A primeira forma prevê que um processo (em Java, pode-se considerar uma *thread*) da aplicação fique bloqueado por um método da API que aguarda a notificação de chegada de uma mensagem. Quando esta for entregue à aplicação destinatária, o processo bloqueado é reiniciado e a mensagem é disponibilizada para processamento. A segunda forma prevê que uma mensagem recém-chegada na aplicação, e associada a uma determinada fila ou a um tópico, é entregue a um tratador específico desta mensagem, previamente associado à fila ou tópico. Na prática, o *Fiorano MQ* notifica a aplicação da chegada da mensagem invocando um método do tratador previamente associado ao tópico da mensagem. A escolha da forma de tratamento das mensagens depende da arquitetura da aplicação que está usando o *Fiorano* MQ.

Tolerância a falhas na rede
A implementação deste mecanismo no *Fiorano MQ* é feita no lado do servidor *Fiorano MQ*, já que é ele que controla o fluxo de entrada e de saída das filas e o estado das conexões com os clientes e com os destinatários das mensagens.

Tolerância a falhas no servidor
O fato do *Fiorano MQ* trabalhar com um gerenciador de filas único, as instâncias dos servidores *Fiorano MQ*, faz com que a implementação de tolerância a falhas no servidor seja mais simples, já que o processo de entrada e de saída das mensagens nas filas e o fluxo de dados nas conexões com os produtores e consumidos é controlado por apenas um único componente.

Balanceamento de carga
Servidores *Fiorano MQ* podem trabalhar de forma cooperativa, em uma configuração que permite balanceamento automático de carga de processamento para mensagens associadas a um dado tópico ou tipo de fila. Esta é uma característica que já vem implementada no *middleware* e pode ser utilizada de acordo com as necessidades específicas de cada aplicação.

Transparência de localização
O *Fiorano MQ* trabalha com servidores de nomes JNDI. Desta forma, a transparência de localização entre transmissores e receptores de mensagens está garantida.

Transparência de migração
O *Fiorano MQ* apresenta esta funcionalidade, podendo ter seus servidores configurados para receberem requisições roteadas por conta de falha em servidores anteriormente ativos.

Transparência de replicação
O *Fiorano MQ* apresenta um mecanismo de repetição, que implementa características similares à replicação automática de serviços.

Segurança – autenticação e criptografia
O *Fiorano MQ* implementa uma política completa de segurança e de administração dos seus serviços, permitindo autenticação de usuários comuns e com atribuições de administrador e autenticação de mensagens. Mensagens podem ser cifradas e criptografadas internamente pelo *Fiorano MQ*, possuindo suporte para SSL (*Security Socket Layer*).

Heterogeneidade
Do ponto de vista de plataforma / sistema operacional, o *Fiorano MQ* suporta 100% de heterogeneidade, desde que o sistema operacional possua uma JVM. Dentre os que possuem JVM, podemos destacar: família *Windows* (95, 98, 2000, *Millenium*, XP, *Server* 2000, *Server* 2003), *Linux*, família *Unix* (HP UX, SCO, *Solaris*, etc.), OS2 e *Mac*. Do ponto de vista de plataforma / linguagem de programação, o *Fiorano* MQ pode interagir com aplicações desenvolvidas em C++ que acessem uma API de interação com o Java. Um suporte adicional aos padrões LDAP e JNI de serviços de diretórios fazem com que o *Fiorano MQ* possa ser utilizado por aplicativos não Java, que não suportam JNDI, embora esta seja a forma padrão de localização de nomes em contextos Java que utilizam o *Fiorano MQ*. Além das características citadas, o *Fiorano MQ* possui algumas funcionalidades úteis e adicionais que podem ser usadas em contextos e necessidades mais específicas.

O *Fiorano MQ* permite a definição de mensagens com prioridades de envio e de processamento diferentes, possibilitando aos seus usuários a priorização de tarefas, de processamento e sinalização de emergência entre aplicações.O *Fiorano MQ* possui um mecanismo de *log* configurável, permitindo aos seus usuários a obtenção de informações sobre ocorrências em processos de transmissão de mensagens, incluindo dados estatísticos e informações sobre medida de performance dos servidores.

Toda a monitoração de um contexto *Fiorano MQ* pode ser feita remotamente via interface web, desde que um servidor web comum esteja disponível na máquina onde o serviço do *Fiorano MQ* está instalado. Além de ser possível administrar o *Fiorano MQ* graficamente, as funções administrativas estão disponíveis na própria API do *Fiorano MQ*. As rotinas de administração têm que ser executadas através de linhas de comando Java ou através da execução de programas com chamadas aos métodos da API disponibilizada. É possível destacar algumas diferenças e semelhanças entre o *Fiorano MQ* e o *Hermes*, objeto de estudo deste trabalho.

O *Hermes* apresenta características similares ao *Fiorano MQ* com respeito ao assincronismo, às tolerâncias às falhas no cliente e no servidor, à transparência de localização. Quanto à heterogeneidade, os dois softwares suportam SOAP (*Simple Object Access Protocol*) e são desenvolvidos na linguagem de programação Java.

As diferenças principais surgem nas partes de tratamento e de transmissão das mensagens, pois o *Hermes* é mais flexível e encapsula uma gama maior de opções para troca de mensagens entre clientes e servidores, que serão explicadas e detalhadas no capítulo seguinte.

O *Fiorano MQ* apresenta duas opções para a forma de persistência de mensagens: banco de dados ou implementação proprietária. No *Hermes*, esta característica é própria do software, embora o projeto permita que opções adicionais relativas às formas de persistência possam ser facilmente incorporadas.

Outras diferenças entre o *Hermes* e o *Fiorano MQ* podem ser observadas nas características relativas à transparência de migração e à transparência de replicação, sistemáticas mais elaboradas no *Fiorano MQ* do que no *Hermes*. O balanceamento de carga no *Fiorano MQ* é mais elaborado e completo, pois o software suporta *clusters* de processamento para os serviços. No *Hermes*, a implementação do balanceamento de carga é mais simples e não suporta *clusters* de processamento.

A segurança no *Fiorano MQ* é elaborada e possui boas opções para utilização de mecanismos de encriptação e de autenticação de mensagens. O *Hermes* não possui um sistema de criptografia para as suas mensagens. O *Fiorano MQ* ainda apresenta uma suíte completa de ferramentas administrativas. O *Hermes* possui uma estrutura básica de administração, até porque a sua forma de utilização é mais simples e direta do que a do *Fiorano MQ*.

3.6 Considerações Finais

Este capítulo apresentou o padrão JMS de implementação em Java estabelecido pela *Sun Microsystems* para projetos de MOMs. Deste estudo, identificam-se dois principais propósitos do JMS: o provimento de um padrão de uso dos MOMs, facilitando a incorporação dos mesmos a sistemas e contextos e permitindo até mesmo uma troca de implementação sem maiores traumas, já que o JMS é um conjunto de interfaces a serem implementadas por diferentes fornecedores de MOMs; o estabelecimento de um padrão de troca de mensagens que seja compatível com servidores de aplicação J2EE. Esta compatibilidade permite que serviços adicionais importantes providos por tais servidores de aplicação possam ser naturalmente incorporados aos MOMs que implementem as interfaces e sigam as especificações de implementação JMS determinadas.

Também foram apresentadas as principais características de três MOMs e, ao final de cada tópico referente a um dado MOM, um sumário comparativo entre este e o *Hermes*

foi discutido. Em todos os casos, notam-se semelhanças em algumas características e diferenças em outras.

O ponto mais destacado do *Hermes* em relação aos sistemas de *middleware* analisados é a maior flexibilidade do primeiro nas modalidades de transmissão e tratamento de mensagens, tópicos que serão detalhados no capítulo seguinte. O que se pode notar também é a ênfase dos projetos de MOMs em itens relativos a heterogeneidade, que permitem uma maior interação destes com outros softwares e sistemas; e relativos a integrabilidade com outros sistemas e *frameworks* de serviços, como servidores de aplicação Java. No capítulo seguinte, aspectos do projeto do *Hermes* serão apresentados e relacionados às características comuns dos MOMs utilizadas para descrever os três sistemas de *middleware* analisados.

Capítulo 4

MOM *Hermes* – Características Gerais e Arquitetura

Este capítulo apresenta as características gerais do MOM Hermes, depois descreve a sua arquitetura e dois aspectos importantes relacionados às suas funcionalidades: o modelo de mensagens e o serviço de nomes.

4.1 Introdução

Este capítulo apresenta as características gerais do MOM *Hermes*, relacionando funções normalmente encontradas em MOMs. Depois, descreve a arquitetura do *Hermes*, mostrando uma visão geral dos seus módulos. Finalmente, apresenta dois aspectos importantes relacionados às funcionalidades do *Hermes*: o modelo de mensagens e o serviço de nomes.

4.2 Características do Hermes

As características do *Hermes* foram classificadas em seis grupos distintos, de acordo com as propriedades dos *MOMs* detalhadas no Capítulo 2: tipos de transmissão, tolerância a falhas, processamento de mensagens, tópicos, monitoramento e interoperabilidade. Em algumas destas características, a implementação do *Hermes* propõe melhorias e extensões em relação aos *MOMs* apresentados no Capítulo 3. Os detalhes associados a cada um destes grupos e implementados no *Hermes* serão descritos nas seções seguintes.

4.2.1 Tipos de Transmissão e de Processamento

Os tipos de transmissão e de processamento em um MOM são características importantes, pois vão determinar como as mensagens são enviadas de um ponto a outro, e como elas serão processadas. O assincronismo na transmissão e no processamento de mensagens, existente nos *MOMs*, garante um desacoplamento total do processo de transmissão em relação ao processamento da mensagem. Neste caso, as *threads* que transmitem e que recebem a mensagem nos lados transmissor e receptor são diferentes da *thread* de solicitação de envio da mensagem e da *thread* de processamento da mesma. Neste caso, diz-se que a transmissão e o processamento da mensagem são assíncronos, existindo quatro pontos independetes de tratamento da mensagem: solicitação de envio, transmissão, recepção e processamento.

Por outro lado, inúmeras aplicações necessitam mesclar os mecanismos síncronos e assíncronos de transmissão, descritos no Capítulo 2. Muitas vezes, tais aplicações têm que implementar estas características, porque os sistemas de *middleware* por elas utilizados não disponibilizam tais combinações de mecanismos de envio e de recepção de mensagens. No *Hermes*, foi prevista a implementação de todas as combinações possíveis das referidas modalidades de transmissão. O assincronismo típico dos MOMs garante que apenas os processos de transmissão e de recepção estejam ligados entre si. Como eles pertencem ao MOM, a aplicação, responsável por solicitar transmissão e por processar mensagens, fica independente dos mecanismos de comunicação, não esperando que estes ocorram e se mantendo imunes a problemas de infra-estrutura associados aos processos de transmissão de dados que possam ocorrer (indisponibilidade de rede e de serviços, problemas físicos, etc.).

Transmissão Síncrona no Transmissor – Processamento Síncrono no Receptor
O *Hermes* suporta transmissão síncrona no transmissor e processamento síncrono no receptor. Isto significa que uma mensagem é enviada de um transmissor para o receptor e o primeiro aguarda o fim do processamento da mensagem pelo segundo, que retorna uma mensagem de resposta. Consulta de estoque em tempo real, realizada por aplicações de venda de produtos, é um exemplo de funcionalidade que utiliza este tipo de transmissão.

Transmissão Assíncrona no Transmissor – Processamento Assíncrono no Receptor
O *Hermes* suporta transmissão assíncrona no transmissor e processamento assíncrono no receptor, com enfileiramento da mensagem em uma fila associada a um tópico no transmissor e enfileiramento da mensagem em uma fila associada ao mesmo tópico no receptor. A transmissão efetiva da mensagem no transmissor é feita em um processo diferente do que requisita a transmissão da dada mensagem, que apenas coloca a mesma em uma fila associada a um tópico. No receptor, o processamento da mensagem é feito por um processo diferente do que recebeu a mensagem do transmissor. Neste caso, o processo de transmissão da mensagem no transmissor não aguarda o processamento da mesma no receptor, e sim recebe apenas uma confirmação do mesmo de que a mensagem foi incluída na fila e será posteriormente processada. É a modalidade de transmissão e processamento de mensagens usada pelos MOMs, e é usada em diversos domínios de aplicações. Transmissão de boletins metereológicos para uma lista de assinantes, realizada por aplicações de controle de clima, é um exemplo de funcionalidade que utiliza este tipo de transmissão.

Transmissão Assíncrona no Transmissor – Preocessamento Síncrono no Receptor
O *Hermes* suporta transmissão assíncrona no transmissor e processamento síncrono no receptor, com enfileiramento da mensagem em uma fila associada a um tópico no transmissor e processamento imediato da mensagem no receptor. A transmissão efetiva da mensagem no transmissor é feita em um processo diferente do que requisita a transmissão da dada mensagem, que apenas coloca a mesma em uma fila associada a um tópico. No receptor, o processamento da mensagem é feito pelo mesmo processo que recebeu a mensagem do transmissor. Neste caso, o processo de transmissão da mensagem no transmissor aguarda o processamento da mesma no receptor, e recebe deste a resposta de processamento da mensagem. Transmissão e processamento de vendas de lojas, realizada por aplicações de venda de produtos, é um exemplo de funcionalidade que utiliza este tipo de transmissão.

Transmissão Síncrona no Transmissor – Processamento Assíncrono no Receptor
O *Hermes* suporta transmissão síncrona no transmissor e processamento assíncrono no receptor, com transmissão imediata da mensagem no transmissor e enfileiramento da mensagem em uma fila associada a um tópico no receptor. A transmissão efetiva da mensagem no transmissor é feita pelo mesmo processo que solicitou o envio da dada mensagem. No receptor, o processamento da mensagem é feito por um processo diferente do que recebeu a mensagem do transmissor. Neste caso, o processo de transmissão da mensagem no transmissor não aguarda o processamento da mesma no receptor, e sim recebe apenas uma confirmação do mesmo de que a mensagem foi incluída na fila e será processada. Integração de cupons cancelados, realizadas por

aplicações de venda de produtos, é um exemplo de funcionalidade que utiliza este tipo de transmissão.

Os dois últimos tipos de transmissão não são suportados pela maioria dos *MOMs* similares ao *Hermes*. O *Hermes* suporta assincronismo parcial, seja de transmissão de mensagens no transmissor, seja de processamento de mensagens no receptor, permitindo uma maior flexibilidade às funções de troca de mensagens.

4.2.2 Tolerância a Falhas

Mecanismos de tolerância a falhas em *MOMs* normalmente são associados à garantia de integridade e de transmissão de mensagens quando há desligamentos abruptos, falhas de execução nas aplicações dos *MOMs* e situações de contingência similares. Além destes aspectos, o contingenciamento de receptores indisponíveis deve ser também previsto em implementações de *MOMs* típicos.

A preocupação principal é garantir que se uma mensagem é entregue a um transmissor de um MOM para envio assíncrono, ela será enviada, mesmo após um religamento da aplicação, ocasionado por falhas de software ou de hardware, ou por interrupção do fornecimento de energia.

Além deste aspecto, tolerância a falhas em *MOMs* requer alternativas de processamento, no caso de indisponibilidade de algum receptor. Neste caso, é comum a configuração, nos transmissores ou até mesmo nos serviços de nomes, de rotas alternativas de envio para determinados tipos de mensagens. A implementação de tolerância a falhas no *Hermes* tem como objetivo principal garantir os pontos mencionados. Os itens que se seguem contém uma descrição das características de tolerância a falhas presentes no *Hermes*:

- O *Hermes* garante entrega e processamento das mensagens, mesmo em caso de queda da aplicação que esteja usando um transmissor *Hermes* e em caso de queda da aplicação na qual esteja rodando um receptor *Hermes*. Este requisito aponta para um sistema de persistência das mensagens nas filas do transmissor e do receptor, quando as mensagens são enviadas e ou processadas de forma assíncrona; e

- O *Hermes* garante alternativas de processamento de mensagens em caso de indisponibilidade de um ou mais receptores. Rotas alternativas de processamento de mensagens devem ser configuradas no Serviço de Nomes do *Hermes*, e são associadas aos tópicos das mensagens. Quando uma mensagem é transmitida a um receptor (de qualquer forma), se este não responder, deve ser possível transmitir a mesma mensagem a receptores alternativos em uma dada ordem de prioridade, até que um deles responda e processe a mensagem ou encaminhe a mensagem para processamento.

4.2.3 Processamento de Mensagens

O processamento de mensagens em um MOM diz respeito à forma de tratamento das mesmas no lado receptor, e é caracterizada pelo mecanismo de entrega, pelo MOM, da mensagem à aplicação responsável por tratar mensagens. O *Hermes* implementa duas das três formas de processamento descritas na Seção 2.7.2, sendo estas implementações diferenciadas por algumas características adicionais. Estas características foram introduzidas para que os usuários do *Hermes* tenham uma menor preocupação com alguns controles que os programas usuários de MOM devem implementar na parte receptora.
O *Hermes* implementa o processamento por notificação de eventos, tratados por *handlers* associados ao tópico da mensagem. A funcionalidade adicional implementada no *Hermes* é a associação entre tópico e tipo de mensagem. Esta modalidade de tratamento por notificação de eventos está disponível para mensagens síncronas.

O *Hermes* implementa o processamento com processo em espera. A funcionalidade adicional implementada no *Hermes* é a abstração provida de determinados controles inerentes à própria implementação da função de leitura das mensagens. As duas preocupações relativas à criação de processos próprios e à lógica de controle implementada dentro da função são retiradas do programador da aplicação e inseridas dentro do próprio *Hermes*. Desta forma, a complexidade do código escrito para este contexto é reduzida e o risco de sobrecarga de processamento ou de mensagens represadas nas filas de entrada do lado receptor são minimizados, já que estes controles ficam a cargo da implementação interna do próprio *Hermes*. Esta modalidade de tratamento por notificação de eventos está disponível para mensagens assíncronas. As duas formas de processamento estão previstas no padrão JMS, embora este último não incorpore as características adicionais encontradas no *Hermes*.

4.2.4 Tópicos

Uma característica dos MOMs é a de associar uma mensagem a um determinado tópico, e este a um par fila de saída – fila de entrada nas aplicações transmissora e receptora. O conceito de tópico é muito relativo e variado, e depende muito do contexto onde os MOMs estão sendo utilizados. Uma lista de tópicos depende fundamentalmente da aplicabilidade do MOM dentro do contexto do sistema onde ele está sendo utilizado.

A implementação do conceito de tópicos do *Hermes* faz com que seja possível adotar uma lista livre de tópicos e uma associação natural de tópico e tipo de mensagem, muito útil para atender a determinados requisitos funcionais de diversas aplicações comerciais.

O *Hermes* considera que os tipos de mensagens são tópicos. Portanto, não é obrigatório definir previamente uma configuração de tópicos suportados pelo *Hermes*, pois os próprios tipos das mensagens trocadas entre transmissores e receptores já são considerados como tópicos e associados às filas. A criação de tópicos associados a tipos e mensagens é dinâmica.

Um transmissor *Hermes* pode funcionar também com tópicos, onde o usuário deve informar a mensagem a ser transmitida e o tópico a ser associado à dada mensagem. No receptor, a fila associada ao tópico será criada dinamicamente, para que a mensagem seja inserida na fila correspondente ao tópico a ela associado.

A associação dinâmica entre tipos de mensagens e tópicos e a criação dinâmica de filas têm muita utilidade quando o MOM é usado predominantemente como um solicitador e executor de processamento, pois não é necessário criar configurações prévias de listas de tópicos nem associar de forma estática tópicos e tipos de mensagens.

4.2.5 Monitoramento

Em todo MOM, o processo de monitoramento das filas de saída e de entrada nos transmissores e receptores é importante, pois o processo de transmissão é muitas vezes assíncrono e os estados do processamento e da transmissão das mensagens não são controlados de forma direta pela maioria das aplicações.

O esquema de monitoramento do *Hermes* prevê a possibilidade de avaliar remotamente os estados das diversas filas de saída e de entrada, localizadas fisicamente em transmissores e receptores ligados a uma rede onde o serviço de monitoramento é executado.

O *Hermes* provê uma ferramenta simples de monitoramento remoto de filas de entrada no receptor e das filas de saída no transmissor, onde estão mostrados: o tamanho de uma dada fila, o estado da última transmissão ou recepção ocorrida, e a data e hora da última transmissão ou recepção ocorrida. Esta ferramenta é constituída por uma interface gráfica da qual é possível consultar diversas aplicações servidoras e clientes que usem o *Hermes,* através de um protocolo de comunicação que permita interoperabilidade.

4.2.6 Interoperabilidade

A interoperabilidade é atualmente uma característica muito importante e presente na maioria dos sistemas de *middleware* modernos. A necessidade de troca de informações entre sistemas diferentes e a crescente cultura de "escreva uma vez e execute em qualquer lugar" fazem com que os MOMs tenham que funcionar sob estas condições. A interoperabilidade está relacionada com dois fatores críticos de projeto em um MOM ou em qualquer *middleware*: a linguagem de programação e o protocolo de troca de mensagens. O *Hermes* suporta interoperabilidade na troca de mensagens entre aplicações desenvolvidas em duas linguagens de programação diferentes – *Java* e qualquer linguagem de programação do *framework .NET*. O *Hermes* ainda garante que as mensagens transmitidas por aplicações *Java* sejam compreendidas por aplicações *.NET* e vice-versa, pois existe o suporte ao protocolo SOAP. O *Hermes* pode viabilizar a troca de mensagens entre aplicações executadas em sistemas operacionais diferentes.

4.3 Visão Geral da Arquitetura

A arquitetura do *Hermes* procura separar claramente funcionalidades distintas, com vistas a facilitar a troca de tecnologias e as futuras evoluções e extensões do software [3]. A arquitetura do *Hermes* descrita nesta seção será o ponto de partida e a referência para as seções subseqüentes que descrevem em detalhes o projeto do *Hermes*. Cada um dos itens previamente descritos nesta seção será detalhado nas seções seguintes. De uma maneira geral, pode-se considerar cada item da arquitetura como sendo um módulo do projeto *Hermes*. Para descrição dos módulos, considera-se que, do lado transmissor, uma *API* de transmissão será usada por qualquer programa que deseje enviar mensagens a outros programas. Estas mensagens podem conter, a princípio, qualquer conjunto de atributos. As mensagens *Hermes* são classes que herdam características comuns definidas em uma classe genérica, e possuem atributos específicos inerentes à natureza do conceito que elas representam. A arquitetura do *Hermes* divide o mesmo em módulos distintos, apresentados na Figura 4.1 e numerados. São eles: módulo *API Hermes* (1), módulo Gerenciador de Filas Saída (2), módulo Transmissor (3), módulo Receptor (4), módulo Gerenciador de Filas Entrada (5) e Módulo Controle da Aplicação (6).

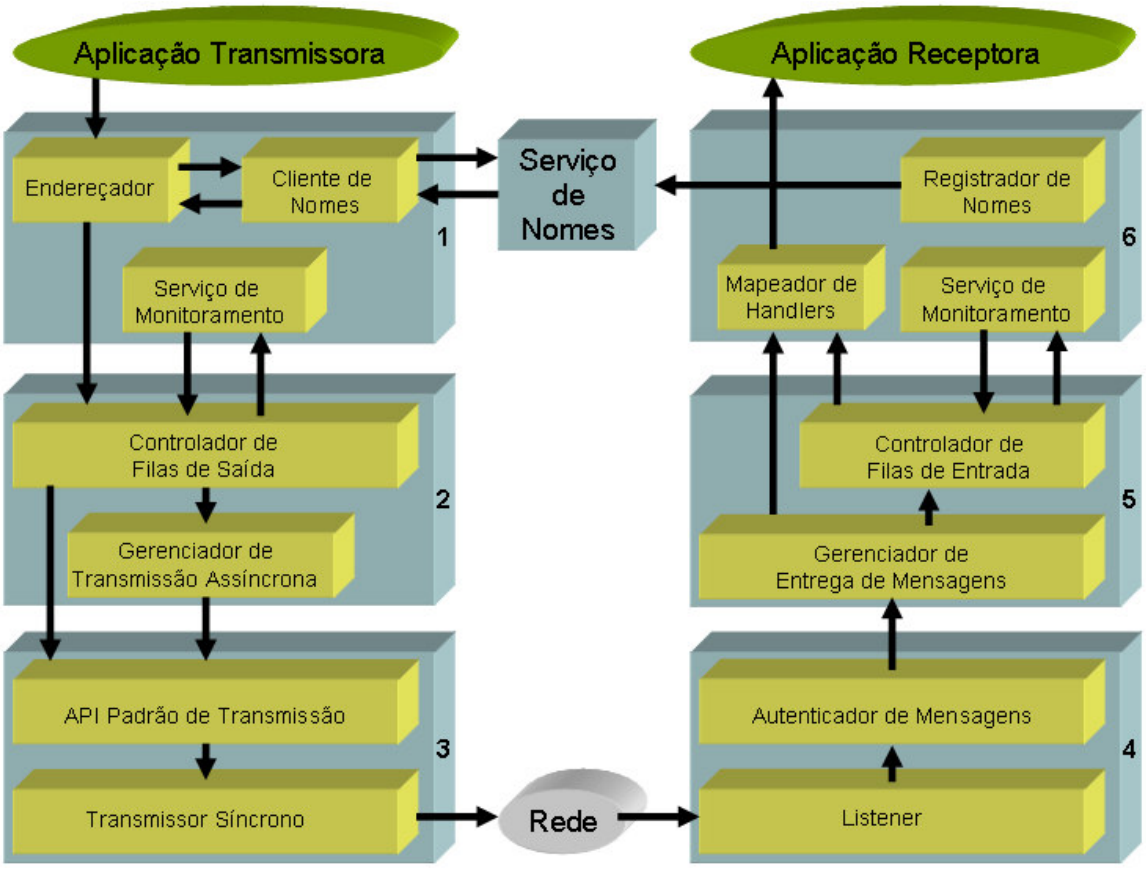

Figura 4.1 – Arquitetura do *Hermes*

O módulo *API Hermes* possui três elementos dentro de sua estrutura: Endereçador, Cliente de Nomes e Serviço de Monitoramento. O módulo Gerenciador de Filas Saída possui dois elementos dentro de sua estrutura: Controlador de Filas de Saída e Gerenciador de Transmissão Assíncrona. Finalmente, o módulo Transmissor possui dois elementos dentro da sua estrutura: API Padrão de Transmissão e o Transmissor Síncrono.

A *API Hermes* é responsável por interagir diretamente com a aplicação, provendo métodos para envio de mensagens das quatro formas mencionadas nas características descritas no Capítulo 4. Neste contexto, cada elemento da *API Hermes* tem um papel específico no processo de endereçamento e de identificação da mensagem.

O papel do Endereçador é o de avaliar resultados de transmissões síncronas diretas e de interações com o Gerenciador de Filas Saída quando transmissões assíncronas são solicitadas. Este elemento é responsável por associar uma mensagem a um tópico pré-definido quando o tópico não é o tipo da mensagem. Além disso, o Endereçador solicita ao Cliente de Nomes as opções de endereço físico – IPs e portas possíveis – para onde uma dada mensagem será transmitida. Cabe ao Cliente de Nomes interagir com o Serviço de Nomes [33][52], através de um protocolo de comunicação, para que este provenha as informações necessárias ao envio de uma dada mensagem a um determinado destino.

O Serviço de Monitoramento é um servidor destinado a atender solicitações de estado de uma dada fila de entrada. Interage com o Gerenciador de Filas Saída, a fim de obter o estado de uma fila de saída associada a um tipo de mensagem e a um endereço específico, disponibilizando-o para consultas posteriores de um sistema de monitoramento.

O Gerenciador de Filas Saída é responsável por encaminhar as mensagens às suas respectivas filas, associando estas com tópicos ou com tipos de mensagens, ou de enviar diretamente mensagens aos seus destinatários, quando a modalidade de envio for síncrona. Este papel é desempenhado pelo Controlador de Filas de Saída.

Quando a transmissão é síncrona, o Controlador de Filas de Saída não insere a mensagem em uma fila e a encaminha diretamente para o Transmissor. No caso de transmissões assíncronas, O Controlador de Filas de Saída é responsável por definir a fila de saída na qual a mensagem será inserida. O critério de escolha da fila a ser utilizada é baseado na premissa de que existirão tantas filas lógicas quantos forem os pares de endereços de destino e tópicos diferentes usados na aplicação.

Quando um tópico é associado a um tipo de mensagem, o conceito prevalece, e o tópico coincide com tipo da mensagem. Neste ponto, um mecanismo de controle de assincronismo – o Gerenciador de Transmissão Assíncrona – é acionado e assume, então, a responsabilidade da leitura da mensagem a partir de sua fila e a entrega da mesma ao Transmissor. Cada fila de saída é lida de forma assíncrona por um processo e a mensagem atual existente na fila é enviada ao seu endereço de destino através do Transmissor.

Caso este envio não seja realizado com sucesso ou a mensagem não tenha sido processada com sucesso no receptor, ela não é retirada da fila de saída e o processo de transmissão dorme por um tempo para tentar retransmitir a mensagem mais tarde. No caso da transmissão e do processamento no receptor serem bem sucedidos, a mensagem é retirada da fila de saída.

O Transmissor disponibiliza a *API* Padrão de Transmissão, que é um conjunto de funções destinadas a realizar o processo de transmissão de uma mensagem a um conjunto de IPs e de portas que servem como opções prioritárias e alternativas de transmissão. A *API* Padrão de Transmissão é implementada pelo Transmissor Síncrono, que encapsula as funcionalidades específicas inerentes ao protocolo de comunicação a ser utilizado no processo de troca de mensagens. Este isolamento da implementação do protocolo de transmissão permite que o *Hermes* suporte facilmente a incorporação de diversos mecanismos de comunicação sem necessidade de alteração da estrutura do software como um todo.

O módulo Receptor possui dois elementos dentro de sua estrutura: *Listener* e Autenticador de Mensagens. O módulo Gerenciador de Filas Entrada possui dois elementos dentro de sua estrutura: Gerenciador de entrega de Mensagens e Controlador de Filas de Entrada.

Finalmente, o módulo Controle da Aplicação possui três elementos dentro da sua estrutura: Mapeador de *Handlers*, Serviço de Monitoramento e Registrador de Nomes. O Receptor é responsável por receber as mensagens, transformá-las em objetos e enviá-las ao Gerenciador de Entrega de Mensagens. O papel do *Listener* é o de implementador de um servidor de comunicação típico, com implementação nativa e otimizada para *socket*. Para *SOAP*, o *Listener* usa uma *API* gratuita de mercado já implementada.

Uma vez com a mensagem em forma de objeto e no padrão definido, o *Listener* entrega a mesma ao Autenticador de Mensagens, que verifica se ela é uma mensagem passível de encaminhamento e de tratamento dentro do contexto de uma aplicação do *Hermes*.

O Gerenciador de Entrega de Mensagens é responsável por verificar se a mensagem deve ser processada de forma síncrona ou assíncrona. No primeiro caso, a mensagem é diretamente encaminhada para o Mapeador de *Handlers*, a fim de ser encaminhada para o seu respectivo *handler*. No segundo caso, a mensagem é entregue ao Controlador de Filas de Entrada, que é responsável por definir a fila de entrada na qual a mensagem será inserida. O critério de escolha da fila a ser utilizada é baseado na premissa de que existirão tantas filas lógicas quantos forem os tópicos diferentes usados na aplicação. Quando um tópico é associado a um tipo de mensagem, o conceito prevalece, e o tópico coincide com tipo da mensagem.

Para cada fila de entrada, existe um processo que lê as mensagens da referida fila e as encaminha para processamento através do Mapeador de *Handlers*. Este processo estabelece o critério de tentar encaminhar a mensagem para processamento, retirá-la da

fila caso ela seja processada com sucesso ou aguardar um tempo para reprocessar a mensagem caso haja sinalização de erro na primeira tentativa de processamento.

O Mapeador de *Handlers* define qual *handler* vai processar uma dada mensagem e a encaminha ao *handler* para processamento. A relação entre instâncias de *handlers* e tipos de mensagens é definida na inicialização da aplicação onde o receptor está sendo executado, através de leitura de parâmetros de configuração específicos da aplicação. A aplicação receptora trata efetivamente a mensagem recebida, através de um *handler* previamente associado à mensagem ou a um tópico. Se for um *handler* síncrono, retorna uma mensagem de resposta. Se for um *handler* assíncrono, não retorna nada caso a mensagem seja processada com sucesso, e lança uma exceção, caso haja erro de processamento da mensagem.

O Registrador de Nomes registra o IP e as portas disponíveis do receptor *Hermes* em um Serviço de Nomes disponível, utilizando um protocolo específico para tal propósito. O Serviço de Monitoramento é um servidor destinado a atender solicitações de estado de uma dada fila de entrada. Interage com o Controlador de Filas de Entrada a fim de obter o estado de uma fila de entrada associada a um tipo de mensagem.

A arquitetura do *Hermes* possui uma estrutura simples, onde as mensagens são entregues pela aplicação a uma API de transmissão disponibilizada pelo software. Um cenário dinâmico da arquitetura previamente descrita é descrito a seguir, mostrando um fluxo resumido do processo de transmissão da mensagem.

A *API Hermes* é responsável por identificar o destinatário da mensagem, interagindo com um servidor de nomes, e por entregar a mesma ao Gerenciador de Filas Saída, que irá encaminhar a mensagem a uma fila específica ou diretamente ao Transmissor. A *API Hermes* ainda inicializa o serviço de monitoramento. O Transmissor é responsável por enviar a mensagem a um receptor específico.

O Receptor é responsável por receber as mensagens e encaminhá-las ao Gerenciador de Filas Entrada, que as encaminha para as suas respectivas filas ou diretamente para os seus *handlers*. As mensagens constantes nas filas são lidas pelo Controle da Aplicação, que as entrega aos seus *handlers* específicos. O Controle da Aplicação ainda registra o mesmo no serviço de nomes e inicializa o serviço de monitoramento. Este cenário é mostrado na Figura 4.2.

A partir da arquitetura, os elementos do projeto que fazem parte destes componentes principais serão apresentados em um nível de detalhe maior. As próximas subseções dividem a arquitetura do *Hermes* em elementos menores, que compõem os módulos do sistema. O Serviço de Nomes é também considerado um módulo do *Hermes,* e interage tanto com a aplicação transmissora quanto com a aplicação receptora.

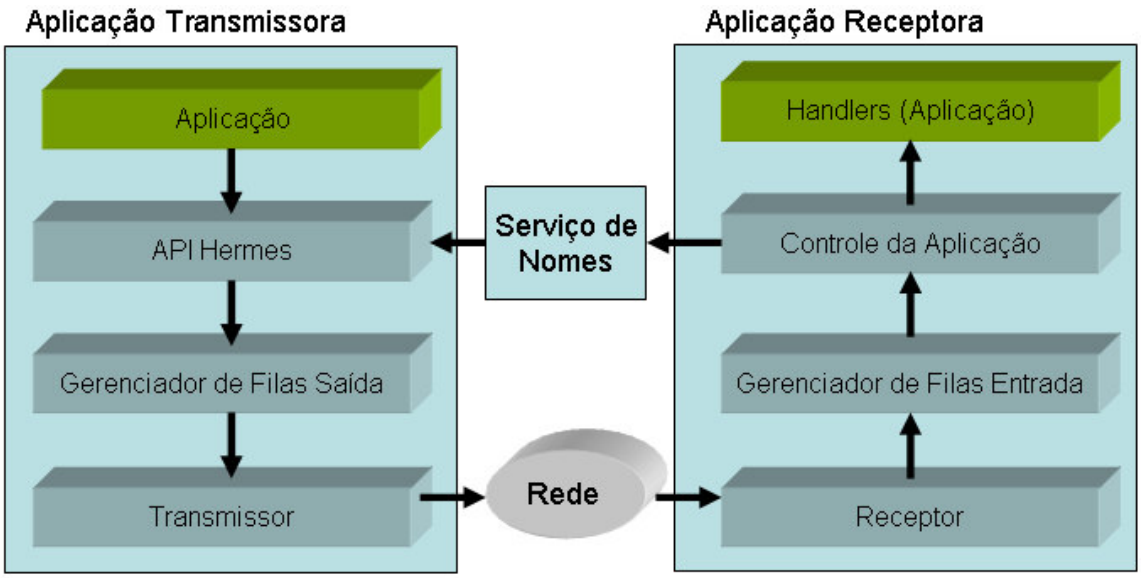

Figura 4.2 – Cenário da Arquitetura do *Hermes*

4.4 Modelo de Mensagens

O *Hermes* implementa um modelo de mensagens no padrão JMS. Cada mensagem *Hermes* tem todas as características previstas no padrão JMS. Além disso, possui dados inerentes à sua natureza específica. As mensagens do *Hermes* possuem um tipo comum, que é uma classe com os dados comuns a todas as mensagens no padrão JMS. As mensagens são sempre sub-classes da referida classe.

O modelo representado em UML [15] na Figura 4.3 mostra as relações entre a super-classe das mensagens *Hermes*, o padrão de interfaces JMS de mensagens e as sub-classes que representam os tipos específicos de mensagens que podem ser criados. O *Hermes* prevê a associação entre "N" tipos de mensagens e um *handler*, que é uma classe capaz de tratar e de processar "N" tipos de mensagens diferentes. Esta associação é feita na inicialização do receptor *Hermes*, que cadastra todos os *handlers* associados aos tipos de mensagens. Quando uma mensagem é encaminhada para processamento, um método do *handler* associado a seu tipo é chamado e a mensagem é passada para o método como parâmetro.

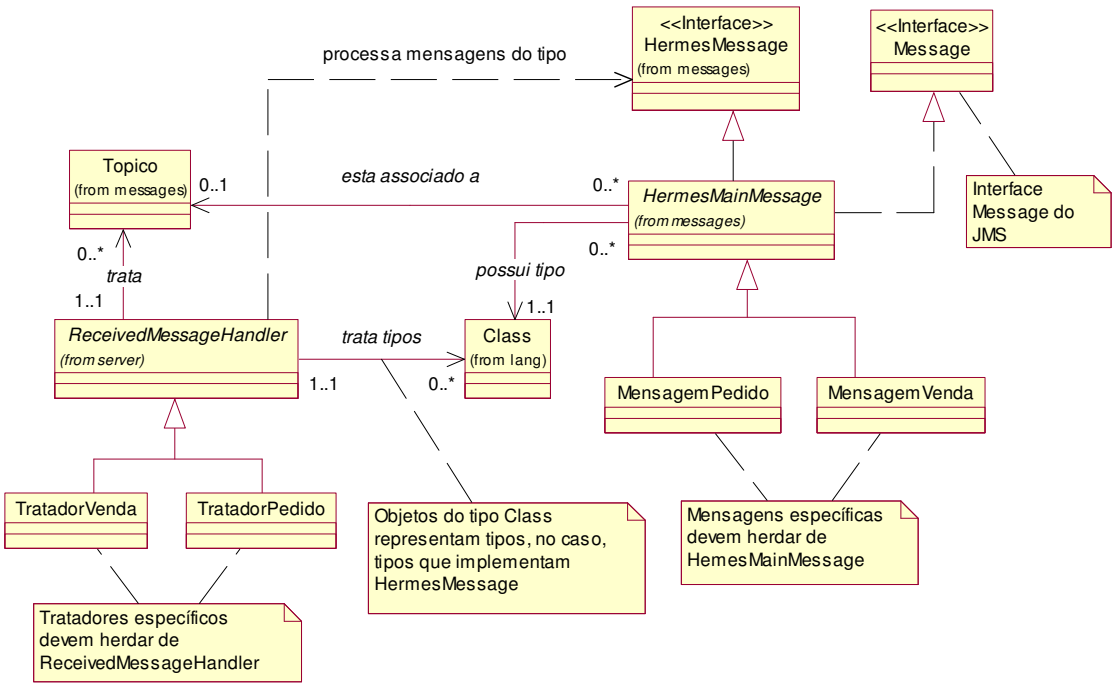

Figura 4.3 - Modelo de Mensagens e de *Handlers* do *Hermes*

A Figura 4.3 mostra a representação em UML de *handlers* e tipos de mensagens.Uma outra forma de processamento de mensagens do *Hermes* prevê que um *handler* pode ser associado a um ou mais tópicos. Assim, mensagens pertencentes a um mesmo tópico são encaminhadas para o *handler* associado ao tópico. As aplicações podem definir a associação entre *handlers* e tipos de mensagens ou entre *handlers* e tópicos escrevendo código dentro do próprio programa, em uma rotina de inicialização do *Hermes*, ou determinar estas associações através de um arquivo XML com formato específico (`ConfHandlers.xml`), onde é possível especificar os nomes das classes que representam as mensagens, os *handlers* e os nomes dos tópicos.

O Quadro 4.1 mostra o formato do arquivo XML [8]. A *tag* `<handlers>` representa a lista de *handlers* a serem configurados e é também a raiz do documento XML. A *tag* `<conf_handler>` representa um *handler* e as mensagens e tópicos a ele eventualmente associados. Dentro desta *tag*, existe a *tag* `<classe_handler>`, que representa o nome da classe onde o *handler* está implementado. Depois, as *tags* `<tipo_mensagem>` e `<topico>` aparecem na quantidade e ordem estabelecidas pelo usuário do *Hermes*. As *tags* `<tipo_mensagem>` devem conter os nomes dos tipos das mensagens que o *handler* pode tratar. As *tags* `<topico>` devem conter os nomes dos tópicos que o *handler* pode tratar.

No Quadro 4.1, a configuração do *Hermes* representada pelo documento XML determina a associação de *handlers*, tipos de mensagens e tópicos da seguinte forma. A classe `HandlerMensagemVenda` é implementada como um *handler* das mensagens cujos tipos são `MensagemVenda` e `MensagemCancelamento`, e vai ainda tratar todas as

mensagens associadas ao tópico `processamento_pedido`. A classe `HandlerMensagemPreco` é implementada como um *handler* da mensagem cujo tipo é `MensagemAtualizacaoPreco`, e vai ainda tratar todas as mensagens associadas aos tópicos `promocoes` e `impostos`.

```
<?xml version="1.0"?>
<handlers>
  <conf_handler>
    <classe_handler>
       com.acme.handlers.HandlerMensagemVenda
    </classe_handler>
    <tipo_mensagem>
       com.acme.mensagens.MensagemVenda
    </tipo_mensagem>
    <tipo_mensagem>
       com.acme.mensagens.MensagemCancelamento
    </tipo_mensagem>
    <topico>
       processamento_pedido
    </tópico>
  </conf_handler>
  <conf_handler>
    <classe_handler>
       com.acme.handlers.HandlerMensagemPreco
    </classe_handler>
    <tipo_mensagem>
       com.acme.mensagens.MensagemAtualizacaoPreco
    </tipo_mensagem>
    <topico>
       promocoes
    </tópico>
    <topico>
       impostos
    </tópico
  </conf_handler>
</handlers>
```

Quadro 4.1 – Estruturas do XML de Configuração dos *Handlers*

4.5 Serviço de Nomes

A função básica de um serviço de nomes é o de relacionar identificadores de serviços com endereços onde estes serviços estão disponíveis e acessíveis. Assim, um serviço de nomes deve ser capaz de guardar uma tabela que relaciona identificador de serviço com endereço físico do mesmo – representado no contexto moderno de comunicação por um par IP x portas disponíveis [52]. Além disso, o serviço de nomes deve ser capaz de receber solicitações de registro, vindas de aplicações servidoras que precisem ter seus endereços físicos registrados no serviço de nomes. Com isto, os usuários destes serviços

se comunicam com o serviço de nomes para mapear o nome de um serviço em um endereço físico real – IP e portas disponíveis. O Serviço de Nomes do *Hermes* contempla a funcionalidade descrita através da implementação de um serviço CORBA, tanto para registro de nomes quanto para identificação de endereços.

Além do processo básico de identificação de endereços a partir de nomes – endereçamento ponto a ponto, o Serviço de Nomes do *Hermes* provê algumas funcionalidades adicionais, relacionadas com balanceamento de carga, contingenciamento de rotas e comunicação *publish-subscribe*.

4.5.1 Endereçamento Ponto a Ponto

O endereçamento ponto a ponto é implementado no Serviço de Nomes do *Hermes* como um processo de identificação onde uma aplicação servidora registra seu endereço associado a um dado identificador de serviço e a aplicação solicita ao Serviço de Nomes um único endereço associado a tal identificador.

A Figura 4.4 mostra a interação entre o Serviço de Nomes *Hermes* e as aplicações servidora e usuária. Primeiro, a aplicação servidora se registra no serviço de nomes (1), passando para o mesmo os seus identificador, IP e portas disponíveis. Quando uma aplicação de um serviço necessita acessar o mesmo, solicita ao servidor de nomes (2) o endereço do serviço, passando o seu identificador. O Serviço de Nomes por sua vez retorna um endereço – IP e portas disponíveis – para a aplicação (3) que usa tal endereço para enviar uma mensagem à aplicação servidora associada ao identificador de serviço (4). Como o *Hermes* trabalha com a possibilidade de ligar um servidor de comunicação a várias portas, característica detalhada no Capítulo 5, que discorre sobre o Receotor, o conceito de endereço engloda um IP e várias portas.

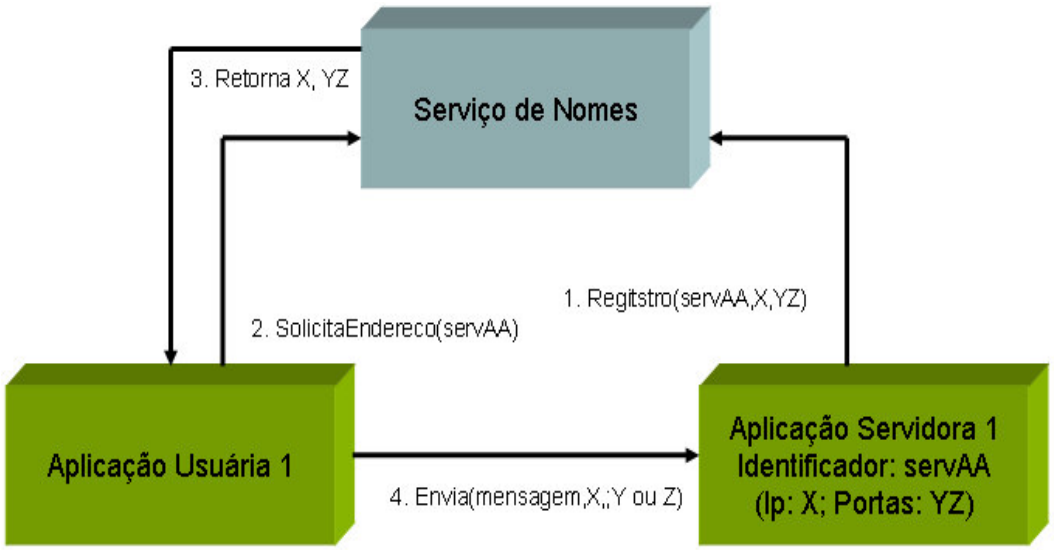

Figura 4.4 – Interação entre o Serviço de Nomes e as Aplicações Servidora e Usuária

4.5.2 Endereçamento com Balanceamento de Carga

No balanceamento de carga, característica dos MOMs já descrita no Capítulo 2, o elemento transmissor e a estrutura do MOM devem ser capazes de avaliar, dentre os possíveis destinatários de uma mensagem, qual o que está mais disponível para processar tal mensagem.

No caso do *Hermes*, quem provê esta funcionalidade é o seu Serviço de Nomes. A fim de prover funcionalidades de balanceamento de carga, o Serviço de Nomes precisa ser configurado para ser capaz de mapear um identificador em uma lista de possíveis endereços de serviços associados a tal identificador.

Além disso, todos os endereços desta lista devem se registrar no Serviço de Nomes, se associando ao identificador. O Serviço de Nomes reconhece que o identificador está associado a uma funcionalidade de balanceamento de carga através da leitura de um arquivo de configuração com a lista de todos os identificadores de serviço que necessitam de balanceamento de carga.

A aplicação que solicitar o endereço de um identificador associado à funcionalidade de balanceamento de carga, vai receber do Serviço de Nomes uma lista ordenada de endereços disponíveis daquele identificador de serviço, em ordem de preferência para envio. A aplicação envia a mensagem para o primeiro da lista. Se este não estiver disponível, tenta enviar a mensagem para o segundo, e assim sucessivamente. A ordem de preferência de envio às aplicações servidoras registradas é determinada pela ordem da lista.

O mecanismo utilizado para o balanceamento de carga guarda sempre a última lista de endereços enviada para um dado cliente. A ordem atual de envio é obtida avançando-se uma posição de cada elemento da referida lista. O primeiro elemento desta passa a ser o último da lista atual. A lista inicial é composta dos endereços já registrados e, à medida que novos endereços forem associados ao identificador do serviço com carga balanceada, são colocados como os primeiros elementos da lista atual.

A Figura 4.5 demonstra a funcionalidade de balanceamento de carga provida pelo Serviço de Nomes *Hermes*. As aplicações servidoras se registram no Serviço de Nomes (1), que as reconhece como opções de balanceamento de carga para um dado identificador, por conta deste estar presente na lista dos serviços com balanceamento de carga. Quando uma aplicação de um serviço necessita acessar o mesmo, solicita ao servidor de nomes (2) o endereço do serviço, passando o seu identificador.

O Serviço de Nomes por sua vez retorna uma lista de endereços – IP e portas disponíveis – para a aplicação (3), que envia a mensagem para o primeiro endereço da lista. Se este não estiver disponível, tenta enviar para o segundo, e assim sucessivamente, até que a mensagem seja enviada com sucesso (4).

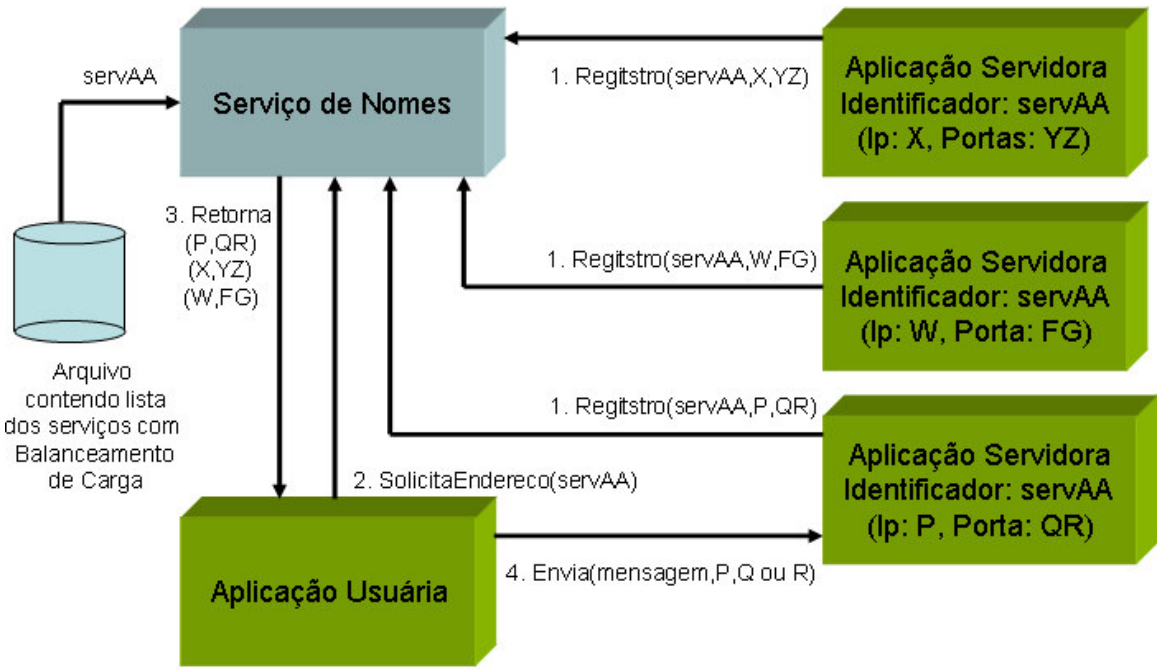

Figura 4.5 – Serviço de Nomes *Hermes* no Contexto de Balanceamento de Carga

4.5.3 Endereçamento com Contingenciamento de Rotas

O contingenciamento de rotas é uma característica implementada para possibilitar tolerância a falhas de servidores ou de provedores de serviço. A idéia é que um determinado serviço, representado por um identificador, tenha mais de uma opção de endereço de acesso, para o caso dos endereços preferenciais estarem indisponíveis.

O próprio mecanismo de balanceamento de carga descrito na seção anterior provê um mecanismo de contingenciamento de rotas, mas a decisão sobre que endereços alternativos estarão disponíveis e qual a ordem preferencial dos acessos alternativos fica estritamente a critério dos próprios servidores alternativos e do Serviço de Nomes, respectivamente. Em muitos casos, um determinado serviço deve ser executado por apenas uma máquina, e apenas em casos excepcionais, outra máquina, normalmente de menor capacidade ou com outras atribuições, deve assumir, de forma temporária, a tarefa de executar tal serviço.

O Serviço de Nomes *Hermes* é capaz de associar um identificador a uma lista de endereços, onde a ordem de tal lista estabelece a preferência de processamento. O primeiro endereço da lista é o preferencial, e os demais endereços representam as rotas alternativas de processamento. Se o identificador estiver associado a tal lista, a aplicação recebe a mesma em resposta à solicitação de endereços e interage com o primeiro da lista. Se este não estiver disponível, tenta interagir com o segundo, e assim sucessivamente. A seqüência de envio das aplicações servidoras registradas é determinada pela ordem da lista.

Do ponto de vista da aplicação, o comportamento é o mesmo observado quando um identificador associado ao mecanismo de balanceamento de carga é tratado. Do ponto de vista do Serviço de Nomes, o mecanismo é diferente, pois a forma de elaboração da lista de endereços a ser retornada para a aplicação leva em conta outras condições para montar tal lista.

Primeiro, a lista é pré-cadastrada em um arquivo de configuração próprio, lido pelo Serviço de Nomes durante a sua inicialização. Segundo, os servidores associados à lista não precisam se registrar no Serviço de Nomes. E por fim, não há a preocupação do Serviço de Nomes com o estado de cada um destes servidores. Ele provê uma lista de rotas alternativas para o envio de uma mensagem.

A Figura 4.6 demonstra a funcionalidade de contingenciamento de rotas provida pelo Serviço de Nomes *Hermes*. O Serviço de Nomes lê um arquivo de configuração que relaciona identificadores suas listas de rotas alternativas (1). Quando uma aplicação de um serviço necessita acessar o mesmo, solicita ao servidor de nomes (2) o endereço do serviço, passando o seu identificador.

O Serviço de Nomes por sua vez retorna uma lista de endereços – IP e portas disponíveis – para a aplicação (3), que envia a mensagem para o primeiro endereço da lista. Se este não estiver disponível, tenta enviar para o segundo, e assim sucessivamente, até que a mensagem seja enviada com sucesso (4).

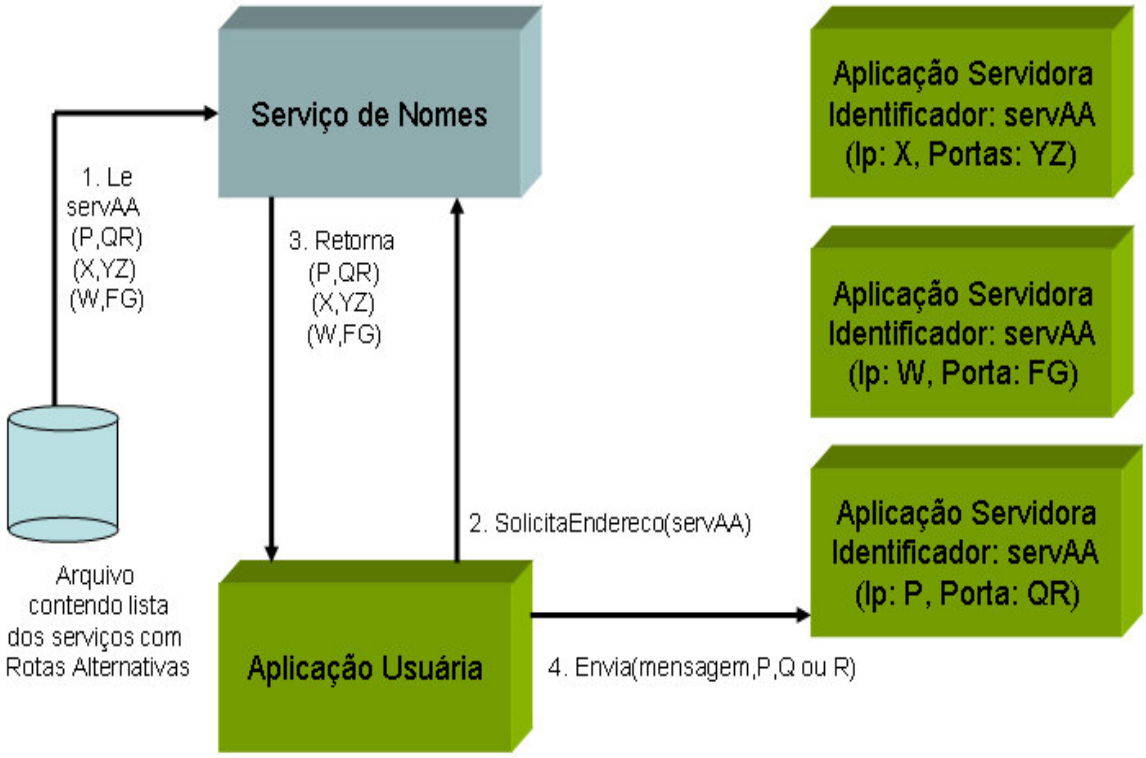

Figura 4.6 – Serviço de Nomes *Hermes* no Contexto de Contingenciamento de Rotas

4.5.4 Endereçamento *Publish-Subscribe*

A comunicação *publish-subscribe*, descrita no Capítulo 2, permite que diferentes *subscribers* se associem a um dado tópico e recebam mensagens de *plublishers* relacionadas a tal tópico. A estrutura do Servidor de Nomes do *Hermes* permite que se estabeleça uma relação entre aplicações servidoras e *subscribers*, entre identificadores de serviços e tópicos; e finalmente, entre *publishers* e aplicações.

Para que isto ocorra, basta que uma aplicação servidora que for se registrar no Serviço de Nomes identifique que a modalidade de registro é do tipo *publish-subscribe*. Quando isto ocorre, em vez de haver uma associação entre um identificador (nome) e um endereço (IP e portas disponíveis), este último é colocado em uma lista de *subscribers*, associada ao identificador. A aplicação vai receber como retorno do Servidor de Nomes uma lista de endereços, mais um indicativo de que a mensagem a ser transmitida deverá ser enviada para todos os endereços constantes na referida lista retornada.

A Figura 4.7 demonstra a funcionalidade *publish-subscribe* provida pelo Serviço de Nomes *Hermes*. As aplicações servidoras se registram no Serviço de Nomes com a opção *publish-subscribe* associada ao identificador de serviço (1). Quando uma aplicação de um serviço necessita acessar o mesmo, solicita ao servidor de nomes (2) o endereço do serviço, passando o seu identificador. O Serviço de Nomes por sua vez retorna uma lista de endereços – IP e portas disponíveis – e a indicação da modalidade de envio *publish-subscribe* para a aplicação (3), que envia a mensagem para todos os endereços da lista (4).

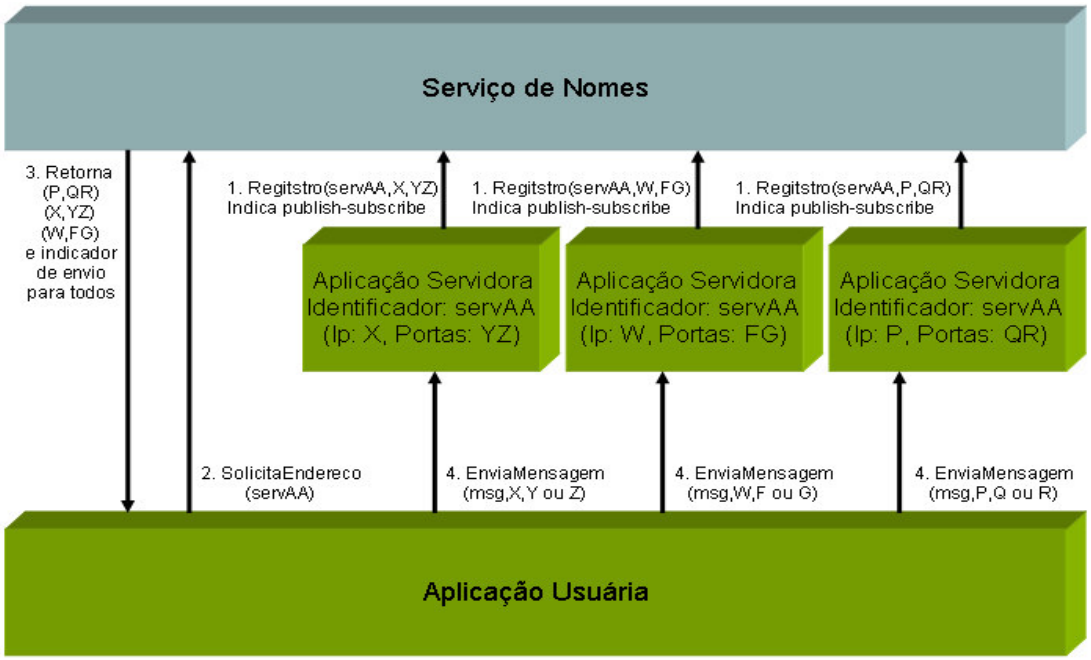

Figura 4.7 - Serviço de Nomes *Hermes* no Contexto *Publish-Subscribe*

4.6 Considerações Finais

Este capítulo, seguindo a estrutura *top-down* de apresentação do MOM *Hermes*, apresentou as características gerais do mesmo, relacionando funções normalmente encontradas em MOMs, a arquitetura do *Hermes*, mostrando uma visão geral dos seus módulos, e dois aspectos importantes relacionados às funcionalidades do *Hermes*: o modelo de mensagens e o serviço de nomes.

As características gerais do *Hermes* mostram que o mesmo apresenta aspectos inerentes a MOMs típicos e incorpora alguns elementos adicionais, dos quais pode-se destacar a flexibilidade nos tipos de transmissão, com a possibilidade de usar qualquer combinação disponível no *Hermes*; e a possibilidade de associar tipos de mensagens a tópicos sem intervenção da aplicação.

No aspecto de arquitetura, o *Hermes* possui módulos bem definidos e funcionalmente independentes. Tal característica aumenta o potencial de reuso dos componentes do *Hermes* dentro de outros módulos no próprio *Hermes* e em outros sistemas. A arquietura é logicamente dividida entre as funcionalidades de transmissão e de recepção, embora estes dois elementos possam ser usados por uma mesma aplicação, que pode fazer uso do *Hermes* tanto como transmissor quanto como receptor de mensagens.

O modelo de mensagens do *Hermes* é compatível com o padrão JMS e permite a associação entre tópicos e ou tipos de mensagens e *handlers* por meio de um arquivo de configuração. Um *handler* pode tratar ou processar mais de uma mensagem ou tópico.

O Serviço de Nomes do *Hermes* permite que o mesmo seja utilizado com formas variadas de endereçamento. Dentre as disponíveis, estão o endereçamento ponto-a-ponto, o endereçamento com balanceamento de carga, o endereçamento com contingenciamento de rotas (uso de rotas de transmissão alternativas) e o endereçamento *publish-subscribe*.

No Capítulo 5, os módulos apresentados na seção deste capítulo que discorre sobre arquitetura serão detalhados e os aspectos de implementação serão discutidos, seguindo a estrutura *top down* de apresentação do MOM *Hermes*.

Capítulo 5

MOM *Hermes* – Implementação

Este capítulo apresenta os módulos do MOM Hermes.

5.1 Introdução

Este capítulo apresenta os detalhes e aspectos de implementação dos módulos do MOM *Hermes* apresentados e brevemente descritos no Capítulo 4.

5.2 Módulo API Hermes

A *API Hermes* é o módulo do transmissor *Hermes* responsável por receber solicitações de envio de mensagens a destinatários específicos. Qualquer aplicação que utiliza o *Hermes* como transmissor deve usar a *API Hermes* para enviar mensagens de forma síncrona ou assíncrona, para um destino específico ou para "N" destinos alternativos. Além desta função, a *API Hermes* inicializa o Serviço de Monitoramento, que responde a solicitações externas de informações sobre o estado das filas de saída existentes no transmissor *Hermes* onde tal serviço é executado.Do ponto de vista funcional, é possível dividir a *API Hermes* em três elementos distintos: o Endereçador, o Cliente de Nomes e o Serviço de Monitoramento. Cada um destes três elementos desempenha um papel diferente no funcionamento da *API Hermes*.

5.2.1 Endereçador

O Endereçador tem dois papéis no cenário de arquitetura do *Hermes*. A sua primeira função é a de prover uma API de envio de mensagens que atenda às diversas necessidades de uso do *Hermes* pelas aplicações. O seu segundo papel é interagir com o Cliente de Nomes para realizar mapeamento de nomes de serviços em listas de endereços físicos. A API de envio de mensagens segue dois padrões distintos. Um deles é definido pelo padrão JMS, que contempla apenas transmissão assíncrona e destinatários identificados por um serviço de nomes JNDI. O outro é definido pelas necessidades adicionais de uso incorporadas ao *Hermes*, notadamente o processo de transmissão síncrona previsto nas formas de transmissão e de tratamento de mensagens, e as opções variadas de endereçamento das mensagens.

O primeiro padrão de API segue as especificações do padrão JMS para um cliente transmissor de mensagens de um MOM, e consiste num conjunto de classes que encapsulam o acesso ao segundo padrão e implementam as interfaces definidas no JMS. O segundo padrão é uma API de mensagens definida no próprio *Hermes*. O Quadro 5.1 mostra a lista de possíveis métodos de transmissão de mensagens a serem utilizados por uma aplicação transmissora *Hermes*. As formas de envio disponibilizadas pela API *Hermes* são diversas, e devem ser utilizadas de acordo com as necessidades específicas de transmissão, normalmente associadas às características funcionais das aplicações do *Hermes*.

Os primeiros métodos constantes no Quadro 5.1 enviam mensagens a um serviço identificado por um nome, que, para o *Hermes*, pode significar uma das diversas

modalidades de endereçamento descritas no Capítulo 4. As funcionalidades de balanceamento de carga e contingencimento de rotas são providas pelo Serviço de Nomes. Portanto, para utilizar estas duas características do *Hermes*, os dois primeiros métodos da API devem ser utilizados no envio das mensagens e o Serviço de Nomes deve estar devidamente configurado para prover tais funcionalidades. Além destas formas de endereçamento, os dois primeiros métodos podem, de acordo com a configuração do Serviço de Nomes, enviar mensagens no modo *publish-subscribe* ou ponto-a-ponto.

A forma mais comum e simples exige que a aplicação passe explicitamente o IP e a porta que caracterizam o destino da mensagem. O terceiro e o quarto métodos listados no Quadro 5.1 mostram esta opção, sendo aquele usado para envio de mensagens assíncronas e este último usado para envio de mensagens síncronas.

Os métodos mostrados no Quadro 5.2 enviam mensagens para uma lista de IPs e de portas, sendo muito utilizados para envio de mensagens *broadcasting*. Mais uma vez, as opções de envios síncrono e assíncrono estão disponíveis. É possível utilizar os referidos métodos para envio no modo *publish-subscribe*, associando a lista de endereços recebida como parâmetro a uma lista de *subscribers* de uma dada mensagem ou tópico.

```
// Envia uma mensagem de forma assíncrona para um serviço
// identificado por um nome
public void sendAsynchronousMessage(HermesMessage message,
   String servName) throws QueueOperationException,
   InvalidDataException;

// Envia uma mensagem de forma síncrona para um serviço
// identificado por um nome
public HermesMessage sendSynchronousMessage(HermesMessage message,
   String servName) throws HermesCommunicationException,
   InvalidHandlerDataException, InvalidDataException;

//Envia uma mensagem de forma assíncrona para um único IP e porta
public void sendAsynchronousMessage(HermesMessage message,
   ServerIdentifier idfServer) throws QueueOperationException,
   InvalidDataException;

// Envia uma mensagem de forma síncrona para um único IP e porta
public HermesMessage sendSynchronousMessage(HermesMessage message,
   ServerIdentifier idfServer) throws HermesCommunicationException,
   InvalidHandlerDataException, InvalidDataException;
```

Quadro 5.1 – *API Hermes* para Envio de Mensagens a Serviços e Endereços

```
// Envia uma mensagem de forma assíncrona para uma lista de IPs e
// portas
public void sendAsynchronousMessage(HermesMessage message,
   ServerIdentifier[] servers) throws QueueOperationException,
   InvalidDataException;

// Envia uma mensagem de forma síncrona para uma lista de IPs e
// portas
public HermesMessage sendSynchronousMessage(HermesMessage
   message,ServerIdentifier[] servers) throws
   HermesCommunicationException,InvalidHandlerDataException,
   InvalidDataException;
```

Quadro 5.2 – *API Hermes* para Envio de Mensagens *Multicasting*

As opções de transmissões síncrona e assíncrona estão também disponíveis no envio de mensagens para um serviço. As relações entre um serviço e as modalidades de endereçamento foram detalhadas no Capítulo 4. Além de definir semânticas para as modalidades descritas, as assinaturas dos métodos definem exceções [7], o identificador de serviço – abstração de um par IP x portas disponíveis que representa unicamente um endereço de destino – e o tipo abstrato que representa uma mensagem no *Hermes*.

Os métodos do *Hermes* trabalham com quatro tipos de exceções e cada uma das quais representa uma situação diferente de contingência no contexto de envio de uma mensagem. A Tabela 5.1 mostra a relação entre os tipos de exceções, seus significados e semânticas específicas.

Tabela 5.1 – Exceções da API *Hermes*

Exceção	Situação em que é lançada
`QueueOperationException`	Se houver erro na inserção da transação em uma fila de saída
`InvalidDataException`	Se for passado algum argumento inválido a qualquer um dos métodos
`InvalidHandlerDataException`	Se uma mensagem não estiver associada a nenhum *handler* no lado receptor
`Hermes``CommunicationException`	Se ocorrer um erro na transmissão da mensagem

O tipo `ServerIdentifier` representa um par IP x portas e o tipo *`Hermes`*`Message` é a interface [7] que toda mensagem do *Hermes* deve implementar. Esta interface está especificada no diagrama de classes que mostra o modelo de mensagens do *Hermes*, apresentado no Capítulo 4. Os métodos de transmissão síncronos sempre retornam uma mensagem de resposta, em detrimento dos métodos assíncronos, que não têm tipo de retorno definido.

A interação do Endereçador com o Cliente de Nomes se dá a fim de que o primeiro converta um nome de um serviço em uma lista de IPs e de portas. Além desta conversão,

o Cliente de Nomes deve prover ao Endereçador que mecanismo de endereçamento de mensagens será aplicado.

Os mecanismos de endereçamento estão associados às características do identificador registrado no Serviço de Nomes e são detalhados no Capítulo 4, e são implementados por um componente do módulo Gerenciador de Filas Entrada.

5.2.2 Cliente de Nomes

O Cliente de Nomes é responsável por interagir com o Serviço de Nomes para obter a lista de endereços físicos (pares IP x porta) para onde uma dada mensagem será enviada e para verificar qual a forma de endereçamento de mensagens que será utilizada no processo de transmissão. O Serviço de Nomes é uma parte do *Hermes* independente da API transmissora e dos módulos do receptor.

Sendo assim, ele é executado como um programa separado das aplicações que utilizam os módulos do transmissor e do receptor *Hermes*, e é usado para manter o registro de endereços físicos e da forma de endereçamento de mensagens para um dado serviço lógico, unicamente identificado por um nome. Portanto, o Serviço de Nomes provém um servidor que permite a realização de consultas a endereços físicos e formas de endereçamento de mensagens associadas a um dado serviço lógico.

O Cliente de Nomes utiliza este servidor do Serviço de Nomes para obter tais informações. A forma de interação entre o Cliente de Nomes e tal servidor é um processo de comunicação entre aplicativos baseado no CORBA [21], com um modelo de objetos representativos da semântica associada a uma lista de endereços físicos e à forma de endereçamento das mensagens. Assim, o Cliente de Nomes é uma implementação de um cliente CORBA [21][52]. As formas de endereçamento das mensagens são apresentadas com mais detalhes no Capítulo 4, que discorre sobre o Serviço de Nomes.

5.2.3 Serviço de Monitoramento

O Serviço de Monitoramento é um componente do *Hermes* responsável por prover um servidor de informações sobre os estados das filas de saída em um transmissor *Hermes*. Basicamente, este servidor tem o papel de responder às consultas que os clientes de monitoramento realizam.

Nestas consultas, são solicitadas informações sobre as filas de saída do transmissor *Hermes* onde o Serviço de Monitoramento está sendo executado. Ao conjunto do Serviço de Monitoramento e dos clientes de monitoramento que realizam consultas a tal serviço, dá-se o nome de sistema de monitoramento de filas. O sistema de monitoramento de filas pode ser visto como um componente do *Hermes* independente dos demais elementos do sistema. Ele é composto do Serviço de Monitoramento e do cliente de monitoramento de filas. Estes dois elementos são executados normalmente em máquinas distintas como

programas distintos, logo deve haver um processo de comunicação para que eles possam interagir entre si.

O Serviço de Monitoramento é incluído como parte do módulo *API Hermes* porque este inicializa tal serviço e ele é executado no mesmo programa de um transmissor *Hermes*, mas, do ponto de vista lógico, o restante do *Hermes* e o Serviço de Monitoramento são componentes distintos e independentes. Do ponto de vista de projeto, o Serviço de Monitoramento é um servidor CORBA [21][52], que disponibiliza funções remotas de consulta aos estados das filas de saída de um transmissor *Hermes* utilizando objetos com semântica própria e representativa destas consultas.

O cliente de monitoramento é uma tela gráfica que interage remotamente com o Serviço de Monitoramento através de um protocolo de comunicação entre aplicativos baseados no protocolo CORBA, e permite a visualização dos estados das filas de um dado transmissor *Hermes*. É possível realizar consultas aos estados de todas as filas ativas ou apenas das associadas a um determinado tópico ou tipo de mensagem. A Figura 5.1 mostra um modelo simplificado do sistema de monitoramento de filas e a sua relação com os demais componentes do *Hermes*.

A classe `FifoMonitorServer`, que representa o servidor, implementa uma interface de serviço nos padrões do CORBA, `FifoMonitorOperations`, e o cliente, representado por `FifoMonitorClient`, usa tal interface para realizar as consultas aos status das filas.

O diagrama mostra ainda a independência entre o sistema de monitoramento e os demais componentes do *Hermes*. A classe `FifoMonitorServer` não interage de forma direta com o componente `OutputQueueManager`, o controlador de filas de saída. A interface `FifoMonitorable` garante que qualquer implementador dela pode ser monitorado por `FifoMonitorServer`, que tem como seu atributo uma referência para qualquer objeto que implementa `FifoMonitorable`, e que inicializa tal referência no seu construtor. Para que o sistema de monitoramento realize consulta aos estados das filas de saída do *Hermes*, na hora da criação do objeto servidor pelo seu construtor é passado para este uma instância de `OutputQueueManager`.

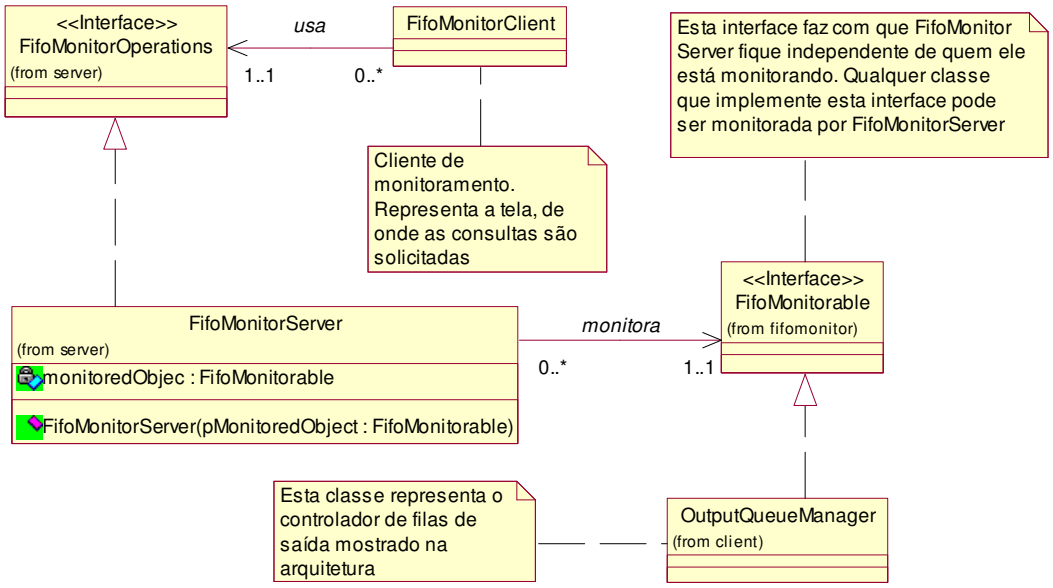

Figura 5.1 – Modelo do Sistema de Monitoramento de Filas – Lado Transmissor

A interface `FifoMonitorable` possui métodos de consulta aos estados das filas do *Hermes*. Tais métodos, listados no Quadro 5.3, provêem algumas possibilidades de consulta, a serem realizadas pelo cliente de monitoramento. A interface de consulta remota `FifoMonitorOperations`, usada pelo cliente de monitoramento e implementada pelo servidor, possui exatamente os mesmos métodos listados.

```
// Realiza consulta do status de uma fila associada a um
// determinado tópico
public QueueStatus getQueueStatusByTopic(String topic);

// Realiza consulta do status de todas as filas ativas
// em um programa que utiliza o Hermes
public QueueStatus[] getAllStatus();

// Realiza consulta do status de uma fila associada a um
// tipo de mensagem
public QueueStatus getStatusByType(Class type);

// Realiza consulta do status das filas associadas aos
// tópicos passados no parâmetro lista de tópicos
public QueueStatus[] getQueueStatusByTopic(
   String[] topicList);

// Realiza consulta do status das fila associada aos
// tipos de mensagens passados no parâmetro lista de tipos
public QueueStatus[] getStatusByType(Class[] typeList);
```

Quadro 5.3 – Assinaturas dos Métodos da Interface `FifoMonitorable`

O trecho de código mostrado no Quadro 5.4 mostra a implementação de um destes métodos na classe `FifoMonitorServer`. Pode-se notar que o servidor repassa a solicitação de consulta para o seu objeto monitorado e que tal objeto tem sua referência interna inicializada no construtor da classe `FifoMonitorServer`. Com esta forma de implementação, o referido construtor [7] pode receber referência a qualquer objeto que implemente a interface `FifoMonitorable`.

```
public class FifoMonitorServer implements FifoMonitorOperations {
   private FifoMonitorable monitoredObject;
     ....

   public FifoMonitorServer(FifoMonitorable pMonitoredObject) {
         monitoredObject = pMonotoredObject;
   }
   public QueueStatus getStatusByType(Class type) {
        return monitoredObject.getStatusByType(type);
   }
}
```

Quadro 5.4 – Repasse da Função de Monitoramento por `FifoMonitorServer`

5.3 Módulo Gerenciador de Filas Saída

Neste módulo, estão implementados os mecanismos de controle para envio síncrono e assíncrono de mensagens. Estes mecanismos contemplam diversos pontos importantes que serão abordados nesta seção: modalidades de endereçamento, representação das filas lógicas, criação e gerência dinâmica das filas lógicas associadas a tipos de mensagens ou a tópicos, persistência das mensagens e controle de transmissão assíncrona. Estes pontos estão implementados nos dois componentes lógicos do módulo Gerenciador de Filas Saída, o Controlador de Filas de Saída e o Gerenciador de Transmissão Assíncrona.

5.3.1 Controlador de Filas de Saída

Este componente é responsável pela implementação das modalidades de endereçamento e criação e gerência dinâmica das filas lógicas associadas a tipos de mensagens ou a tópicos. A Figura 5.2 mostra um modelo simplificado de classes do Controlador de Filas de Saída. A maioria das classes e interfaces apresentadas desempenha uma funcionalidade especifica associada às responsabilidades definidas para o Controlador de Filas de Saída. Outras são inerentes ao Gerenciador de Transmissão Assíncrona, cujas interações são também mostradas no diagrama.

As interações com o módulo de transmissão, representado pela interface *Hermes*`TransmissorInterface` e pelas classes `ObjectClientSocket` e `SOAPClient`, são também explicitadas. Para compreender o funcionamento das classes mostradas no diagrama da Figura 5.2, serão abordados em separado os dois mecanismos

de transmissão de mensagens possíveis em um transmissor *Hermes*: os mecanismos síncrono e assíncrono.

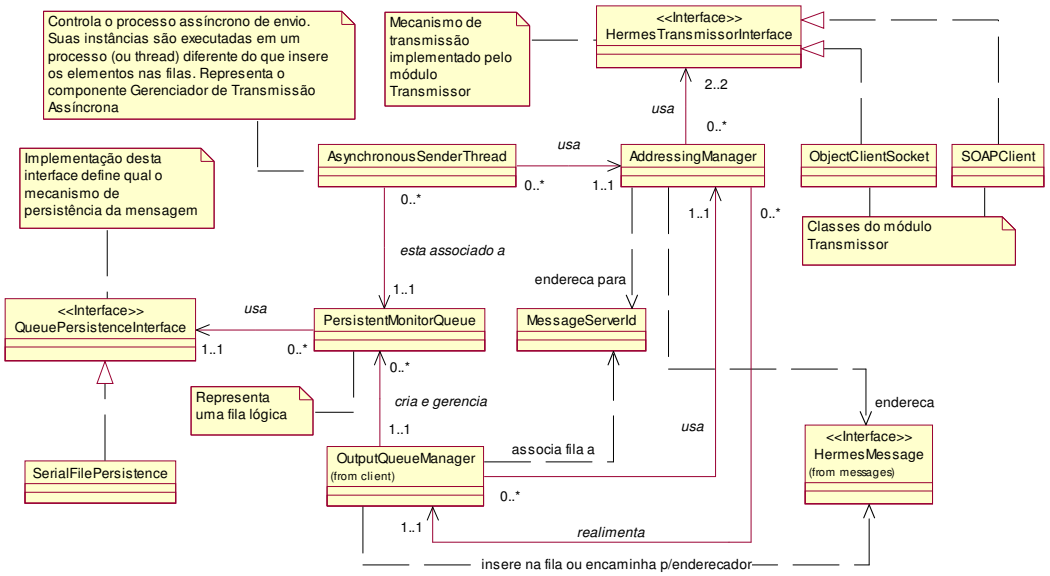

Figura 5.2 – Modelo Simplificado de Classes do Módulo Gerenciador de Filas Saída

A classe `OutputQueueManager` utiliza um objeto do tipo `AddressingManager` para enviar mensagens síncronas, seja para um endereço (ponto a ponto), para um dos endereços de uma dada lista (balanceamento de carga ou contingenciamento de rotas) ou para uma lista de endereços (*publish-subscribe*).

A classe `AddressingManager` implementa as funções de endereçamento de mensagens. Quando uma mensagem síncrona deve ser enviada, o `OutputQueueManager` entrega a mesma ao objeto do tipo `AddressingManager`, informando uma das quatro modalidades de endereçamento suportadas pelo *Hermes*, e o endereço ou uma lista de endereços de destino.

Desta forma, o objeto do tipo `AddressingManager` envia a mensagem a um ou mais destinatários, dependendo da modalidade de endereçamento associada, usando uma implementação específica da interface *`Hermes`*`TransmissorInterface`.

A lógica utilizada pela classe `AddressingManager` para enviar mensagens síncronas depende da forma de endereçamento associada a tal mensagem. Ela usa uma implementação de *`Hermes`*`TransmissorInterface` para realizar tal envio.

A Tabela 5.2 resume as situações de endereçamento para mensagens síncronas. A classe `MessageServerId` representa um endereço e a interface *`Hermes`*`Message` representa uma mensagem.

Tabela 5.2 – Modo de Funcionamento da classe `AddressingManager`

Elementos que `AddressingManager` recebe	**Indicador de envio**	**Modalidade de endereçamento contemplada**
`MessageServerId` *`Hermes`*`Message`	Apenas um endereço	Ponto a ponto
`MessageServerId[]` *`Hermes`*`Message`	O primeiro disponível	Balanceamento de Carga Contingenciamento de Rotas
`MessageServerId[]` *`Hermes`*`Message`	Todos os endereços	*Publish-subscribe*

Os processos de endereçamento da classe `AddressingManager` para cada um dos indicadores de envio e para transmissão síncrona são descritos abaixo.

Apenas um endereço

A classe `AddressingManager` envia a mensagem e retorna para OutputQueueManager um dos dois status possíveis: transmissão realizada com sucesso ou erro de transmissão.

```
HermesTransmissorInterface.send(HermesMessage, MessageServerId)
if (send == OK)
   //Sinaliza Tx OK
else
   //Sinaliza Tx Nao OK
```

Quadro 5.5 – Endereçamento para Apenas um Endereço e Transmissão Síncrona

O primeiro disponível

A classe `AddressingManager` percorre a lista de endereços e envia a mensagem para os endereços da lista enquanto não houver sucesso no processo de envio. Quando isto ocorre, a interação sobre a lista de endereços é interrompida e o retorno para `OutputQueueManager` é de transmissão realizada com sucesso. Se a mensagem não for enviada para nenhum endereço com sucesso, o retorno dado para `OutputQueueManager` é um status de erro no envio.

```
txOk = false
loop:for i = 1 to lista.length {
  HermesTransmissorInterface.send(HermesMessage,lista[i]);
  if (send == OK) {
      txOk = true;
      break loop;
  }
}
if (txOk)
   //Sinaliza Tx OK
else
   //Sinaliza Tx Nao OK
```

Quadro 5.6 – Endereçamento para o Primeiro Disponível e Transmissão Síncrona

Todos os endereços

A classe `AddressingManager` percorre a lista de endereços e envia a mensagem para todos os endereços da lista. Se todas as mensagens forem transmitidas com sucesso, o retorno para `OutputQueueManager` é um status indicando que a transmissão foi realizada com sucesso. Caso contrário, o retorno para a mesma classe indica a quantidade de transmissões realizadas com sucesso. O Quadro 5.7 mostra tal processo.

```
for i = 1 to lista.length {
  HermesTransmissorInterface.send(HermesMessage,lista[i]);
  if (send == OK)
     qtdTransmitida = qtdTransmitida + 1;
}
if (qtdTransmitida == lista.length)
   //Sinaliza Tx OK
else
   //Sinaliza Tx OK para "qtdTransmitida" endereços
```

Quadro 5.7 – Endereçamento para todos os Endereços e Transmissão Síncrona

O processo de envio assíncrono é mais elaborado, pois envolve questões relativas ao gerenciamento das filas, além de algumas funcionalidades e controles adicionais referentes aos mecanismos de endereçamento.

A classe `OutputQueueManager` possui uma tabela interna populada com diversos objetos do tipo `PersistentMonitorQueue`, a classe que representa uma fila lógica no *Hermes*. Para cada objeto deste tipo constante na tabela, está associada uma chave, que é composta por um nome e um endereço. Este nome pode representar um tópico ou um tipo de mensagem, e o endereço representa um IP.

Assim, para cada tópico ou tipo de mensagem, podem existir "N" filas de saída, cada uma das quais associadas a um endereço de destino diferente. A criação e eventual remoção destas filas é dinâmica, ou seja, à medida que novas solicitações de envio de mensagens, associadas a novos tópicos, de novos tipos e ou de novos endereços, ainda não conhecidos pelo `OutputQueueManager`, for solicitado, novas filas vão sendo criadas e inseridas na tabela.

Após a criação da nova fila, a mensagem é inserida na mesma. Se a chave, composta pelos elementos nome e endereço, existir na tabela, a mensagem é inserida na fila que está associada a tal chave. Este mecanismo elimina o trabalho dos usuários do *Hermes* de criarem configurações de tópicos e de filas previamente, como acontece com alguns MOMs existentes.

Não há a preocupação de definição prévia das filas a serem gerenciadas, pois os próprios mecanismos do *Hermes* criam filas dinamicamente. Por outro lado, é necessário existir um controle de filas lógicas ociosas. Para isto, o *Hermes* destrói, periodicamente, as filas que estão vazias e inativas há mais de um determinado tempo. Este mecanismo está implementado na classe `OutputQueueManager`, que inicializa uma *thread* independente de "*garbage collector*" para filas lógicas ociosas e vazias.

Esta *thread* verifica periodicamente, para cada fila, se ela está vazia e se a diferença entre a data e hora da última operação nela realizada e a data e hora atual é maior que um tempo pré-determinado. Em caso positivo, a fila é retirada da tabela que a relaciona com a chave nome + endereço.

A Figura 5.3 mostra um diagrama de objetos que indica as relações entre a instância [7] de `OutputQueueManager`, a instância da classe `Hashtable`, que representa uma tabela em memória [35], as instâncias de `PersistentMonitorQueue` e as `Strings [35]` que representam as chaves das filas. A classe `Hashtable` é a implementação do relacionamento entre `OutputQueueManager` e `PersistentMonitorQueue`, mostrado do diagrama da Figura 5.2.

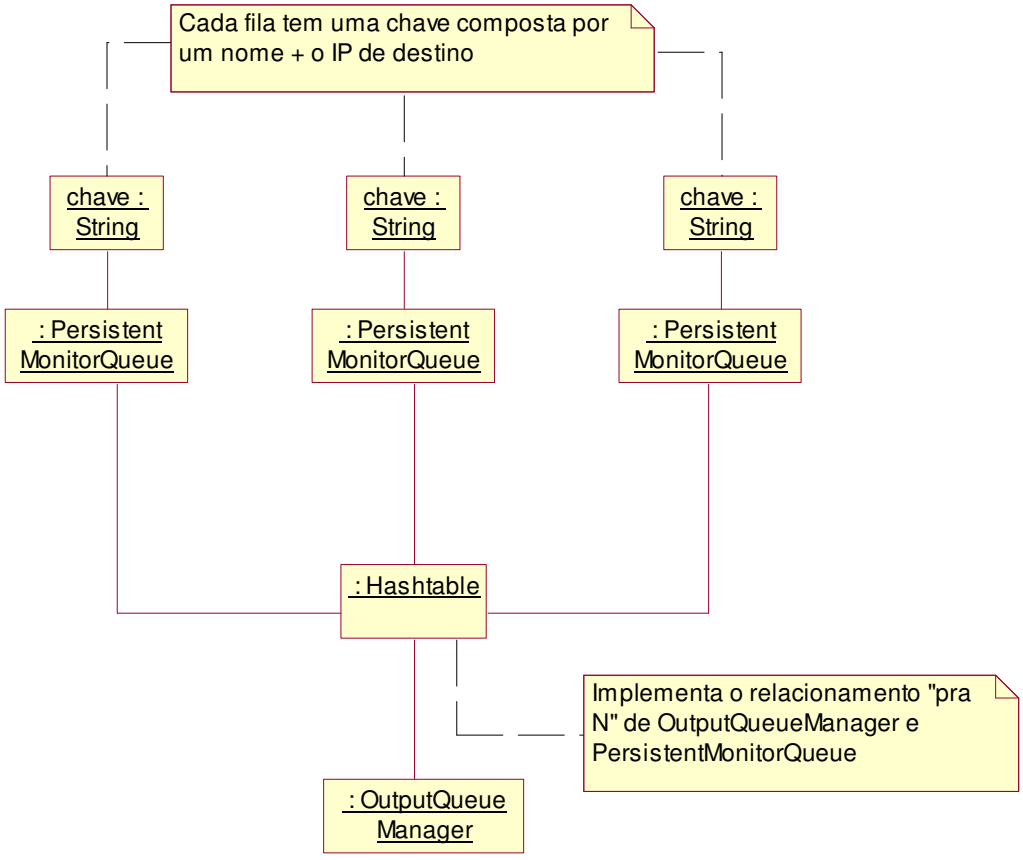

Figura 5.3 – Diagrama de Objetos para `OutputQueueManager` e Filas

O processo de inserção de uma mensagem na fila, realizado pelo `OutputQueueManager`, avalia o endereço de envio, pois este último depende da modalidade de endereçamento. O endereço de envio é que vai determinar em qual fila a mensagem será inserida. O `OutputQueueManager` avalia o endereço de envio de três maneiras diferentes:

- Ponto a ponto: O endereço de envio é o recebido e a mensagem é inserida em apenas uma fila, cuja chave é o IP deste endereço mais um tópico a ela associado ou seu tipo;

- Balanceamento de carga ou contingenciamento de rota: O endereço de envio é determinado por um índice, que indica qual dos endereços constantes na lista de endereços será utilizado para envio. A mensagem neste caso também é inserida em apenas uma fila, cuja chave é o IP deste endereço obtido através do índice da lista, mais um tópico a ela associado ou seu tipo. Nestas modalidades, a mensagem guarda a lista de endereços a ela associada e o índice do endereço atual de envio; e

- *Publish-subscribe*: A mensagem deve ser enviada a todos os endereços da lista, e é inserida em "N" filas (onde "N" representa o tamanho da lista), cujas chaves são compostas pelos respectivos IPs dos endereços da lista, mais um tópico associado à mensagem ou seu tipo.

Desta forma, as mensagens são encaminhadas às suas respectivas filas de envio, em qualquer modalidade de endereçamento. Uma vez que uma mensagem é inserida em uma dada fila, uma *thread* representada pela classe `AsynchronousSenderThread`, diferente da que inseriu a mensagem na fila e a esta última associada, é acordada e lê da fila a mensagem para, utilizando o `AddressingManager`, enviá-la ao seu destino. No caso deste último confirmar o envio, a mensagem é retirada da fila. Em caso negativo, a mensagem permanece na fila e, depois de um certo tempo, uma nova tentativa de envio é realizada pelo `AsynchronousSenderThread`. Esta permanece neste ciclo até que a mensagem seja enviada com sucesso ou até que a mesma expire por *timeout*, sendo retirada da fila nesta ocasião e inserida em um repositório específico de mensagens mortas e não transmitidas. As próprias mensagens podem implementam um critério específico de expiração. Na super classe das mensagens do *Hermes*, o critério padrão implementado para expiração é um *timeout*.

Ao `AddressingManager` cabe enviar as mensagens assíncronas entregues por `AsynchronousSenderThread`. A lógica utilizada por aquela para enviar mensagens síncronas utilizando uma interface transmissora, sempre implementada por uma das classes do módulo transmissor referentes ao protocolo de comunicação utilizado, depende da forma de endereçamento associada a tal mensagem.

Em modo de transmissão assíncrono, a classe `AddressingManager` não trata mensagens *publish-subscribe*, porque uma mensagem que possui esta modalidade de envio já foi replicada pelo `OutputQueueManager` nas suas respectivas filas de saída, associadas aos "N" endereços constantes na lista de envio. Como existe um `AsynchronousSenderThread` associado a cada fila, e aquele interage com o `AddressingManager`, este último só recebe mensagens assíncronas a serem enviadas a um único endereço, que é justamente o especificado na chave da fila.

Para o `AddressingManager`, uma mensagem publish-subscribe assíncrona equivale a "N" chamadas de uma das suas funções pelos "N" objetos do tipo

`AsynchronousSenderThread` associados às filas onde a mensagem foi inserida. No caso, cada fila por sua vez está associada a um dado endereço.

Desta forma, o `AddressingManager` trata mensagens *publish-subscribe* e mensagens ponto a ponto indistintamente, pois o próprio mecanismo de avaliação do endereço de destino das mensagens implementado na classe `OutputQueueManager` garante esta premissa. A lógica de tratamento nestes casos é idêntica à observada no Quadro 5.5, referente ao "apenas um endereço para transmissão síncrona", apenas se trocando o papel de `OutputQueueManager` por `AsynchronousSenderThread`.

Em relação às modalidades de endereçamento balanceamento de carga e contingenciamento de rotas, que se resumem à forma de "o primeiro disponível", o tratamento dado às mensagens assíncronas é diferenciado em relação ao procedimento descrito para transmissões síncronas. Tanto no processo de transmissão síncrono quanto no processo de transmissão assíncrono, uma mensagem é sucessivamente enviada a endereços constantes na lista de endereços até que um destes envio seja realizado com sucesso.

A diferença entre um e outro caso é que o envio no processo síncrono é a chamada a uma função de transmissão disponibilizada pela interface de transmissão *Hermes*`TransmissorInterface;` e o envio no processo assíncrono é a inserção da mensagem em uma fila associada a seu tópico ou tipo e o seu endereço, feita pelo `OutputQueueManager`.

No modo assíncrono, o `AddressingManager` recebe a mensagem, um endereço de envio e o indicador de primeiro disponível (modalidades de endereçamento balanceamento de carga e contingenciamento de rotas). Na mensagem, constam também a lista de endereços alternativos e o índice atual, que indica a posição na lista dos endereços a ser utilizado para transmissão. Estas informações são usadas pelo `OutputQueueManager` no processo de avaliação dos endereços de transmissão e pelo `AddressingManager`, na hora de reencaminhar a mensagem.

O `AddressingManager` tenta enviar a mensagem para o endereço de envio. Caso a mensagem seja transmitida com sucesso, o `AddressingManager` retorna este status para o `AsynchronousSenderThread`, que retira a mensagem da fila. Em caso negativo, o `AddressingManager` incrementa o índice atual, que indica a posição, na lista, do endereço a ser utilizado para transmissão.

Neste ponto, ele devolve a mensagem para o `OutputQueueManager` e retorna um status para o `AsynchronousSenderThread` indicando que este último deve retirar a mensagem da fila. A mensagem sai então da fila de um endereço onde não foi possível o processo de transmissão e entra na fila do próximo endereço disponível na lista. O Quadro 5.8 mostra a lógica de interação descrita.

```
indAtual = HermesMessage.indiceAtual
HermesTransmissorInterface.send(HermesMessage,lista[indAtual]);
if (send == OK) {
    //Sinaliza Tx OK
} else {
    HermesMessage.indiceAtual = HermesMessage.indiceAtual + 1
    OutputQueueManager.insertQueue(HermesMessage)
    //Sinaliza Tx Ok
}
```

Quadro 5.8 – Endereçamento para o Primeiro Disponível e Transmissão Assíncrona

5.3.2 Gerenciador de Transmissão Assíncrona

Este componente é responsável pela representação das filas lógicas, pela persistência das mensagens e pelo controle de transmissão assíncrona. Fazem parte deste componente alguns elementos mostrados no diagrama de classes da Figura 5.2. A fila e seu mecanismo de persistência, implementadas pelas classes `PersistentMonitorQueue` e `SerialFilePersistence` e a interface `QueuePersistenceInterface`; e a classe `AsyncrhonousSenderThread`.

A representação das filas lógicas é a implementação da classe `PersistentMonitorQueue`, que é, ao mesmo tempo, um mecanismo de FIFO (*First In, First Out*) [48], tipicamente implementado por estruturas de dados representativas de filas, e um monitor de dois processos (*threads*) [35].

Para a implementação de um FIFO, são providas algumas funções de acesso a uma estrutura de dados do tipo fila, tais como: ler primeiro elemento, remover primeiro elemento, inserir elemento, retornar tamanho da fila e remover todos os elementos . As assinaturas de algumas destas funções estão mostradas no Quadro 5.9.

```
public void insert(Object obj) throws QueueException;
public Object read() throws QueueException;
public Object remove()throws QueueException;
public int size() throws QueueException;
public readIndex(int index) throws QueueException;
```

Quadro 5.9 – Métodos da Classe `PersistentMonitorQueue`

As funcionalidades de fila foram implementadas de forma que a classe enfileira objetos genéricos, do tipo `Object [7]`, sem depender de tipos específicos ou mesmo da interface *Hermes* que representam as mensagens. Desta forma, *PersistentMonitorQueue* é uma classe independente do *Hermes* e de uso geral, estando inserida no contexto lógico deste último.

A funcionalidade do monitor de *threads [19][32][35]*, implementada em `PersistentMonitorQueue`, garante a integridade de operações realizadas em

objetos deste tipo por *threads* diferentes. Pelo modelo da Figura 5.2, duas classes distintas acessam um objeto do tipo fila, e estas classes são executadas em *threads* diferentes: uma delas é `OuputQueueManager`, que insere mensagens em diversos objetos do tipo `PersistentMonitorQueue`. Outra classe é o processo que lê e transmite as mensagens, representada pela classe `AsyncrhonousSenderThread`.

Estas operações alteram simultaneamente o estado dos objetos do tipo `PersistentMonitorQueue`, por isso precisam ser sincronizadas para evitar leituras ou atualizações indesejáveis, realizadas eventualmente de forma simultânea por *threads* distintas. Para isto, as operações de `PersistentMonitorQueue` utilizam uma regra de sincronização de acesso a objetos, definida da seguinte forma:

a) Quando uma *thread* acessa uma função de um objeto, ela ganha o *lock* deste objeto, e só libera tal *lock* quando a função terminar a sua execução ou quando houver um bloqueio da execução da função dentro da mesma (chamada a funções de dormir ou de suspender execução, disponíveis no sistema operacional) [35].

b) Se uma *thread* tentar acessar uma função de um objeto e outra *thread* já estiver de posse do *lock* deste objeto, aquela aguarda a liberação do *lock* para executar a função chamada [35].

Este mecanismo funciona como um semáforo [17], que controla as chamadas a funções de um determinado objeto. Enquanto uma função está sendo executada por uma *thread*, nenhuma outra pode executar a mesma função em execução ou até outras funções até que o *lock* do objeto seja liberado.

Um objeto que possui estas características é chamado de monitor das *threads*, e as funções que prendem o *lock* a uma determinada *thread* são ditas funções sincronizadas [32][35]. É desta forma que o controle de concorrência das operações da fila `PersisentMonitorQueue`, é implementado. A Figura 5.4 mostra quais as *threads* e métodos usados no acesso concorrente aos objetos deste tipo.

A forma de funcionamento da interação mostrada na Figura 5.4 é a de um esquema produtor – consumidor [17][35]. Cada fila é provida de mensagens por um processo produtor, a *thread* que executa a chamada do `insert()` pela classe `OutputQueueManager`, e tem suas mensagens consumidas por um processo consumidor, a *thread* que executa as chamadas do `read()` e do `remove()` pela classe `AsyncrhonousSenderThread`.

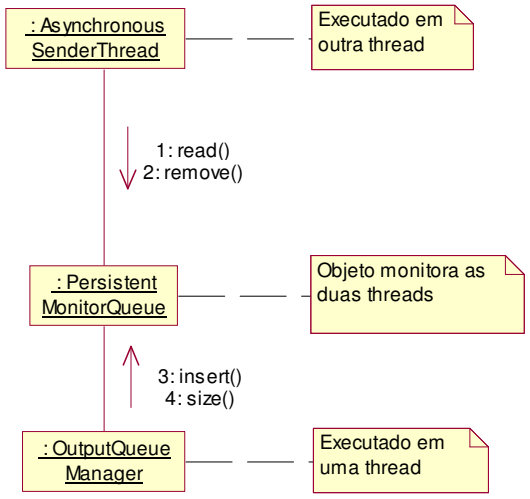

Figura 5.4 – Fila como Monitor de Objetos

Em um processo desta natureza, o consumidor deve estar sempre apto a processar mensagens, lendo as mesmas da fila e as removendo quando necessário. Desta forma, o controle das chamadas aos métodos de `read()` e de `remove()`, feito pela classe `AsyncrhonousSenderThread`, é implementado dentro de um *loop* infinito, que sempre lê, processa e remove mensagens da fila. No caso de `AsyncrhonousSenderThread`, o processamento da mensagem é justamente o envio da mesma através da classe `AddressingManager`. A classe `AsyncrhonousSenderThread` é uma *thread* que implementa uma função executada dentro do processo consumidor.

```
while(true) {
    HermesMessage message = PersistentMonitorQueue.read();
    AddressingManager.send(message);
    if (send == OK) {
        PersistentMonitorQueue.remove();
    } else if (message.isExpired()) {
        insertMessageInDeadMessageRepository(message);
        PersistentMonitorQueue.remove();
    } else
        sleep(TEMPO_SLEEP_ERRO);
    }
}
```

Quadro 5.10 – Lógica da Função do Processo Consumidor

A lógica mostrada no Quadro 5.10 mantém a *thread* representada por uma instância de `AsyncrhonousSenderThread` sempre em funcionamento ou dormindo. Enquanto o programa no qual ela está operando estiver em execução, ela estará viva. Uma mensagem é lida da fila, representada por uma instância de `PersistentMonitorQueue`. Depois de lida, a mensagem é enviada ao seu destino pelo `AdderssingManager`, que retorna um status de envio realizado com sucesso ou

não. No primeiro caso, a mensagem é removida da fila e o ciclo se repete para a próxima mensagem. No segundo caso, é checado se a mensagem já expirou por *timeout* ou não.

Ocorrendo a expiração da mensagem, ela é inserida no repositório de mensagens mortas, removida da fila e o ciclo se repete para a próxima mensagem. Se a mensagem ainda não estiver expirada, a *thread* que executa o processo descrito dorme por um tempo determinado e, após decorrido este tempo, o ciclo se repete para a mesma mensagem, pois ela não foi removida da fila e será novamente lida.

Enquanto a fila estiver vazia, o processo consumidor permanece inativo. Em outros termos, dormindo. Assim que uma mensagem for inserida na fila, o processo consumidor acorda e processa a mesma. Isto ocorre na implementação do *Hermes*, e os controles necessários estão nas funções `insert()` e `read()` da fila.

A função `read()` da fila bloqueia o fluxo de execução da *thread* que o chamou enquanto não existirem elementos na fila. Este bloqueio de execução faz com que a *thread* fique dormindo, liberando o *lock* do objeto referente à fila. O Quadro 5.11 mostra, de forma simplificada, esta lógica.

```
public Object read() {
    // Enquanto o tamanho da fila for zero, a mesma esta vazia
    while(size() == 0) {
        // Este comando bloqueia o fluxo de execução da thread
        // corrente, fazendo tal fluxo parar neste ponto do
        // código (função bloqueante)
        wait();
    }
    return getFirstElement();// Retorna o primeiro elemento
}
```

Quadro 5.11 – Lógica da Função `read()` da Fila

Se algum elemento for inserido na fila através da função `insert()`, ela deve, além de realizar uma operação interna de inserção, executar um comando para acordar o processo consumidor que eventualmente tenha invocado a função `read()` e esteja na condição de bloqueio. O Quadro 5.12 mostra, de forma simplificada, esta lógica

```
public void insert(Object obj) {
        // Chama função interna para inserir fisicamente o
        // objeto recebido em uma estrutura que representa fila
        insereElemento(obj);
        // Executa um comando que acorda todos os processos
        // bloqueados pelo objeto corrente
        notifyAll();
}
```

Quadro 5.12 – Lógica da Função `insert()` da Fila

Todos os mecanismos de controle relativos a uma fila estão implementados na classe `PersistentMonitorQueue`, que ainda possui controles para acesso concorrente de

threads e para bloqueio e desbloqueio de processos. A outra característica importante na implementação da fila é a forma de persistência. O mecanismo de persistência foi implementado em um modelo a parte, independente da classe que implementa os controles de fila e de concorrência.

Todas as operações de leitura, inserção e remoção de elementos da fila são delegadas pela classe `PersistentMonitorQueue` para a implementação de uma interface cuja semântica é um mecanismo genérico de persistência, a interface `QueuePersistenceInterface` da Figura 5.2. Desta forma, é possível trocar uma determinada implementação deste mecanismo genérico por outra sem que seja necessário realizar alterações na classe `PersistentMonitorQueue`. O mecanismo implementado pelo *Hermes* usa o padrão de serialização e persistência de objetos [7][35], disponível nas linguagens de programação utilizadas. Cada fila é um diretório com um determinado nome e cada elemento é um arquivo físico diferente. O nome do diretório é o identificador da fila, no caso nome e IP de destino.

5.4 Módulo Transmissor

Este módulo é responsável pela implementação do protocolo de comunicação estabelecido para troca de mensagens entre um cliente e um servidor de comunicação, fazendo o papel do primeiro. Os dois componentes deste módulo são a API Padrão de Transmissão e o Transmissor Síncrono.

5.4.1 API Padrão de Transmissão

A API Padrão de Transmissão define um conjunto de funções necessárias ao processo de transmissão de mensagens. Estas funções devem ser implementadas por elementos diferentes que se propõem a disponibilizar protocolos de comunicação diferentes. Desta forma, é possível isolar o mecanismo de transmissão de dados das demais funcionalidades do *Hermes*. As premissas estabelecidas para a definição da API Padrão de Transmissão são:

- Sinalização de erros de transmissão, causados por indisponibilidade da rede ou do receptor com o qual o processo de comunicação é estabelecido; e

- Comunicação baseada em um processo de envio da mensagem e de recepção de uma resposta do receptor. Esta resposta pode assumir três indicadores: resposta do processamento (no caso de processamento síncrono), mensagem inserida na fila de entrada com sucesso (no caso de processamento assíncrono), não há *handlers* habilitados para processar a mensagem e erro de processamento da mensagem.

De acordo com as premissas estabelecidas, a API Padrão de Transmissão deve sinalizar todas as situações de exceção na transmissão e no processamento da mensagem no

receptor. Uma mesma situação de exceção pode ter causas e conseqüências diferentes, em função do tipo de transmissão e de processamento. A relação entre as combinações dos tipos de transmissão e de processamento suportados pelo *Hermes* e os significados das situações de exceção estão nas Tabelas 5.3, 5.4, 5.5 e 5.6.

Tabela 5.3 – Exceções para Transmissão e Processamento Síncronos

Transmissão Síncrona – Processamento Síncrono		
Situação de Exceção	**Causas**	**Conseqüência**
Erro na transmissão	Indisponibilidade de rede Indisponibilidade do receptor Erros de protocolo	Sinalização de exceção para a aplicação
Handler não associado à mensagem	No lado receptor, o tipo da mensagem ou o tópico não está associado a nenhum *handler*	Sinalização de exceção para a aplicação
Erro de processamento da mensagem	No lado servidor, o *handler* que processa a mensagem sinalizou exceção	Sinalização de exceção para a aplicação

Tabela 5.4 – Exceções para Transmissão Assíncrona e Processamento Síncrono

Transmissão Assíncrona – Processamento Síncrono		
Situação de Exceção	**Causas**	**Conseqüência**
Erro na transmissão	Indisponibilidade de rede Indisponibilidade do receptor Erros de protocolo	Mensagem é mantida na fila de saída até expirar, e enquanto ela se mantiver na fila de saída, haverá solicitações periódicas de reenvio
Handler não associado à mensagem	No lado receptor, o tipo da mensagem ou o tópico não está associado a nenhum *handler*	Mensagem é mantida na fila de saída até expirar, e enquanto ela se mantiver na fila de saída, haverá solicitações periódicas de reenvio
Erro de processamento da mensagem	No lado receptor, o *handler* que processa a mensagem sinalizou exceção	Mensagem é mantida na fila de saída até expirar, e enquanto ela se mantiver na fila de saída, haverá solicitações periódicas de reenvio

No processamento síncrono, sempre ocorre avaliação de status do processamento da mensagem, seja pela aplicação que envia a mesma (caso de transmissão síncrona), seja pelo processo que lê as mensagens da fila de saída e as envia (caso de transmissão assíncrona).

Tabela 5.5 – Exceções para Transmissão Síncrona e Processamento Assíncrono

Transmissão Síncrona – Processamento Assíncrono		
Situação de Exceção	**Causas**	**Conseqüência**
Erro na transmissão	Indisponibilidade de rede Indisponibilidade do receptor Erros de protocolo	Sinalização de exceção para a aplicação
Handler não associado à mensagem	Não ocorre para o transmissor, pois a mensagem sempre é colocada na fila de entrada no lado do receptor	Mensagem é mantida na fila de entrada até que ela expire ou até que um *handler* seja associado a ela
Erro de processamento da mensagem	Não ocorre para o transmissor, pois a mensagem é colocada na fila de entrada no lado do receptor e um processo independente vai tentar processar a mensagem	Mensagem é mantida na fila de entrada até expirar, e enquanto ela se mantiver na fila de entrada, haverá solicitações periódicas de reprocessamento

Tabela 5.6 – Exceções para Transmissão e Processamento Assíncronos

Transmissão Assíncrona – Processamento Assíncrono		
Situação de Exceção	**Causas**	**Conseqüência**
Erro na transmissão	Indisponibilidade de rede Indisponibilidade do receptor Erros de protocolo	Mensagem é mantida na fila de saída até expirar, e enquanto ela se mantiver na fila de saída, haverá solicitações periódicas de reenvio
Handler não associado à mensagem	Não ocorre para o transmissor, pois a mensagem sempre é colocada na fila de entrada no lado do receptor	Mensagem é mantida na fila de entrada até expirar ou até que um *handler* seja associado a ela
Erro de processamento da mensagem	Não ocorre para o transmissor, pois a mensagem é colocada na fila de entrada no lado do receptor e um processo independente vai tentar processar a mensagem	Mensagem é mantida na fila de entrada até expirar, e enquanto ela se mantiver na fila de entrada, haverá solicitações periódicas de reprocessamento

No processamento assíncrono, não ocorre avaliação de status do processamento da mensagem pela aplicação que envia a mesma, que é sempre inserida na fila de entrada do lado do receptor. Outro processo, diferente do que recebe a mensagem, tem a responsabilidade de avaliar se houve erro de processamento ou se a mensagem não está associada a nenhum *handler*.

5.4.2 Transmissor Síncrono

O Transmissor Síncrono tem a responsabilidade de realizar o processo de transmissão da mensagem através da rede. É representado por uma classe que implementa a interface `HermesTransmissorInterface`, e usa um protocolo específico de comunicação para realizar o processo de transmissão.

No *Hermes*, há duas implementações para a referida interface. Uma delas utiliza o protocolo *socket* TCP [23], mas envia pela rede bytes que representam um objeto da linguagem de programação Java. A outra implementação usa uma API de transmissão baseada no protocolo SOAP [1], que utiliza XML para representar objetos transmitidos pela rede através do protocolo http.

A implementação do *socket* TCP que envia objetos é feita pela classe `ObjectClientSocket`, representada no diagrama da Figura 5.5. Para cada processo de transmissão de uma mensagem, esta classe utiliza um objeto do tipo `Socket [23][28]`, provida pelo Java e responsável pela implementação do TCP, e duas classes do Java, `ObjectOutputStream` e `ObjectInputStream` [28][35], que realizam respectivamente as funções de transformar objetos em bytes e bytes em objetos.

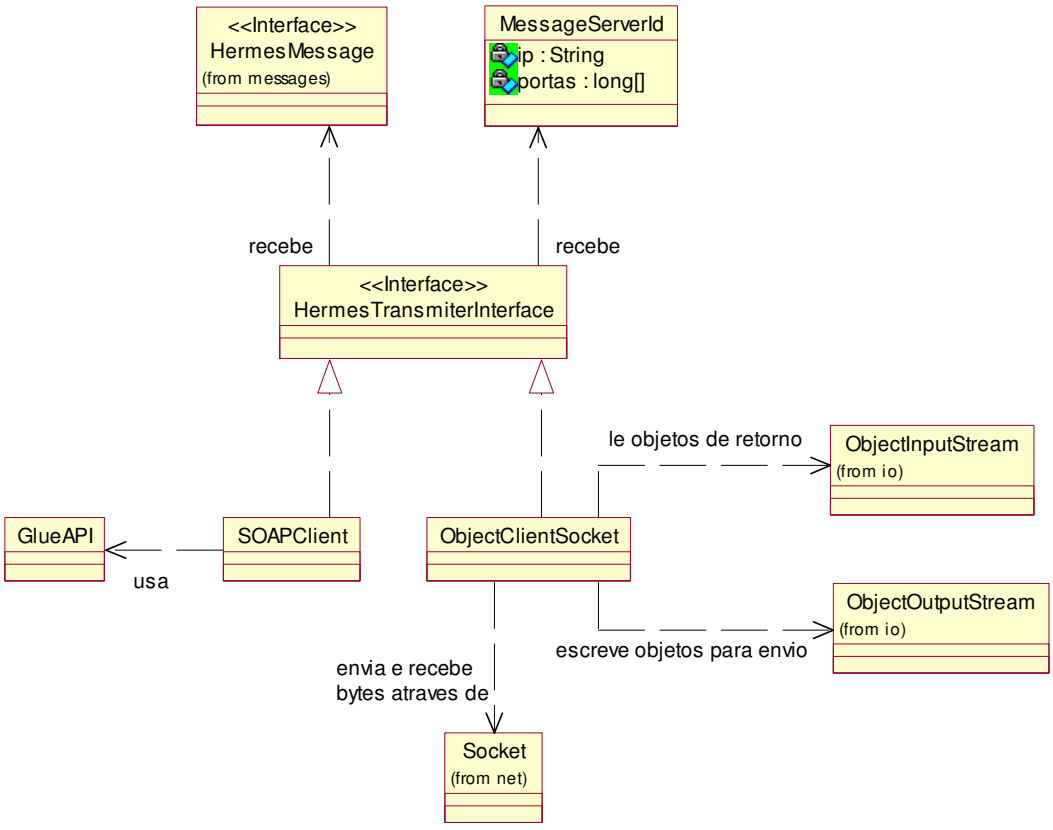

Figura 5.5 – Modelo das Classes Transmissoras

Um ponto importante e adicional na implementação da classe `ObjectClientSocket` é a opção de transmissão para mais de uma porta. É possível que um servidor *socket* [23] do *Hermes* esteja ligado a mais de uma porta, proporcionando opções de envio para um IP e "N" portas possíveis. Desta forma, a classe `MessageServerId`, que representa um endereço de destino, contém um IP e um *array* de portas como atributos. A classe `ObjectClientSocket` tenta enviar a mensagem para as sucessivas portas da lista enquanto o envio não for realizado com sucesso.

A implementação do protocolo SOAP, que envia objetos em formato XML usando http, é feita pela classe `SOAPClient`, representada no diagrama da Figura 5.5. Esta classe usa uma API COTS (*Components Off The Shelf*) para Java que implementa todos os pormenores do protocolo de comunicação SOAP, representada pela classe `GlueAPI`. O resultado do uso desta API COTS é a abstração, pelo *Hermes*, de todo o processo de conversão de objetos em XMLs no padrão SOAP e vice-versa, e de toda implementação do código de envio da mensagem e de recebimento da resposta usando o protocolo http. O uso do SOAP permite que o *Hermes* possa ser implementado em mais de uma linguagem de programação, pois é um protocolo de comunicação padrão para troca de objetos através de redes locais e Internet.

O objeto de retorno do processo de transmissão contém duas informações essenciais para avaliação do resultado do processamento no lado receptor. A mensagem de resposta, no caso da transmissão ser síncrona, e o status do processamento, composto de indicadores representativos das situações de exceção.

5.5 Módulo Receptor

Este módulo é responsável pela implementação do protocolo de comunicação estabelecido para troca de mensagens entre um cliente e um servidor de comunicação, fazendo o papel deste último. Os dois componentes deste módulo são o *Listener* e o Autenticador de Mensagens.

5.5.1 Listener

O *Listener* tem a responsabilidade de realizar o processo de recepção da mensagem oriunda da rede, exercendo o papel de um servidor de comunicação. É representado por uma classe que implementa um protocolo específico de comunicação para realizar o processo de recepção. Tal classe é inicializada e se liga a uma ou mais portas lógicas de comunicação, aguardando as solicitações de conexão, estabelecidas através das referidas portas lógicas.

No *Hermes*, há duas classes que implementam um servidor de comunicação. Uma delas utiliza o protocolo *socket* TCP, mas recebe pela rede bytes que representam um objeto da linguagem de programação Java. A outra classe usa um servidor de comunicação baseado

no protocolo SOAP, que utiliza XML para representar objetos recebidos pela rede através do protocolo http.
A implementação de um servidor de comunicação pressupõe o estabelecimento de políticas relativas ao controle das portas lógicas utilizadas e ao uso racional dos recursos de processamento disponíveis na máquina onde a aplicação receptora é executada . A classe que implementa o protocolo *socket* TCP é provida de algumas otimizações relativas a estes dois pontos, posteriormente detalhadas: a utilização de múltiplas instâncias de objetos servidores e a presença de um pool de *threads* [9][17][28] receptoras.

A Figura 5.6 mostra o modelo de classes do *Listener*, implementado para *socket* TCP. A classe `ListenerSocket` é inicializada e lê um arquivo de configuração do *Hermes* que indica quais portas lógicas estarão disponíveis para receber conexões de transmissores. Para cada porta lógica presente no arquivo de configuração, é criada uma instância da classe `ServerSocket [28]`, implementação de um servidor de comunicação *socket* TCP disponibilizada na linguagem de programação Java.

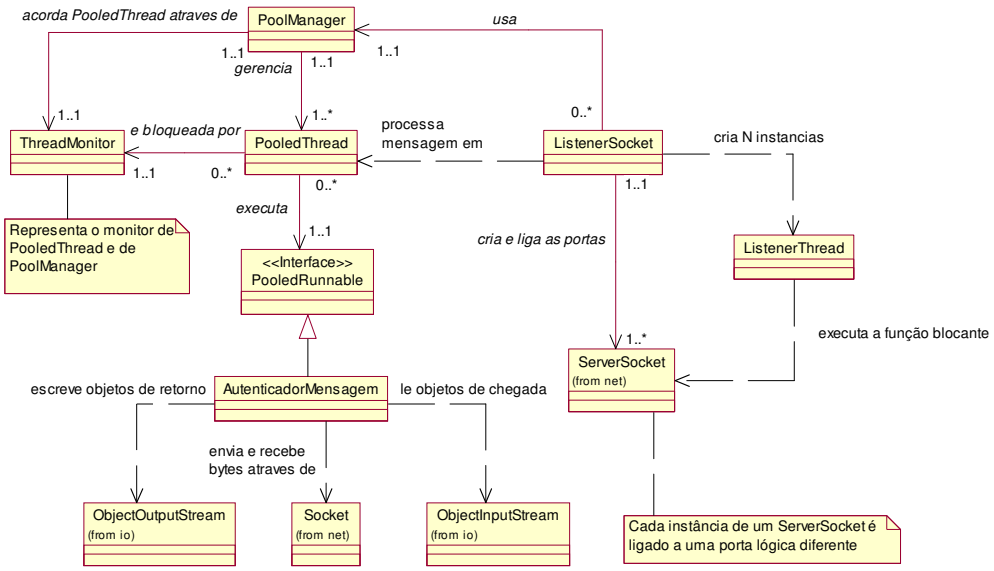

Figura 5.6 – Modelo das Classes Receptoras para Protocolo *Socket* TCP

A instanciação de "N" objetos servidores *socket* TCP associados a "N" portas lógicas, em vez de apenas um objeto ligado a uma única porta, propicia um considerável aumento de disponibilidade de serviço. Para entender porque isto ocorre, é necessário conhecer a forma de utilização de um servidor *socket* TCP.

A forma de utilização de um servidor *socket* TCP [28] é um *loop* infinito executado em uma *thread* independente, que realiza chamada a uma função bloqueante do servidor *socket* TCP. Designa-se esta *thread* de *listener thread*, representada no modelo da Figura 5.6 pela classe `ListenerThread`. Enquanto não houver solicitação de conexão, a referida *thread* fica dormindo.

Quando há a solicitação de conexão, o servidor *socket* TCP desbloqueia a *listener thread* e retorna, na chamada da referida função, os bytes representativos da mensagem recebida. A *listener thread* cria uma outra *thread* que processa os bytes recebidos usando o autenticador de mensagens, retornando à chamada da função bloqueante do servidor *socket* TCP. O Quadro 5.13 mostra a lógica descrita.

```
while(true) {
    // Função bloqueante do servidor socket TCP
    // Quando uma conexão é estabelecida por um cliente,
    // a função accept() é desbloqueada e retorna um array
    // de bytes representando a mensagem
    byte[] mensagem = ServerSocket.accept();
    // Cria uma thread que irá processar a mensagem
    PooledThread threadProcessadora = PoolManager.getThread();
    // Inicializa a thread processadora da mensagem
    threadProcessadora.start(mensagem,AutenticadorMensagem);
}
```

Quadro 5.13 – Lógica da *Listener Thread*

A indisponibilidade do serviço pode ocorrer se uma nova tentativa de conexão for estabelecida enquanto a *listener thread* realiza os passos de criar a *thread* processadora e inicializar a mesma. Se apenas uma *listener thread* estiver atendendo às solicitações de conexão, a probabilidade de ocorrer indisponibilidade é maior do que se existissem "N" *listener threads*, desde que a distribuição de conexões entre as portas seja relativamente eqüitativa.

Com base neste raciocínio, o modelo de *listener threads* do *Hermes* é múltiplo, permitindo que se associem "N" *listeners threads* a "N" portas lógicas. Desde que as configurações de portas nos transmissores sejam estabelecidas de forma a distribuir as conexões de diferentes transmissores eqüitativamente entre as "N" portas lógicas disponíveis, pode-se obter um significativo aumento de disponibilidade de um receptor *Hermes*.

A *thread* de processamento é criada através de uma chamada a uma função da classe `PoolManager` da Figura 5.6, responsável pela gerência das *threads* a serem utilizadas no contexto da aplicação receptora do *Hermes*. A criação e o uso descontrolado de *threads* em aplicações podem causar degradação de performance e ocupação excessiva de memória [17][35]. Desta forma, é importante que aplicações com características servidoras tenham algum mecanismo de controle e de reaproveitamento de *threads* já criadas [17][28]. Tais mecanismos são implementados por componentes designados *pool* de *threads*. A classe `PoolManager` desempenha tal papel dentro do *Hermes*.

A implementação de `PoolManager` mantém em memória uma lista de *threads* inativas, adormecidas por funções bloqueantes. Estas *threads* são instâncias da classe `PooledThread` da Figura 5.6. A cada instância de `PooledThread`, é associado um objeto monitor [35], instância de `ThreadMonitor` da Figura 5.6. Esta classe possui

duas funções, uma delas com implementação bloqueante - `waitExecutionRequest()`- e outra com um comando para acordar processos bloqueados – `wakeUpExecutor()`.

Quando o `PoolManager` é inicializado, instancia "N" `PooledThreads` e inicializa cada instância em uma *thread* diferente. A execução destas *threads* ocorre em um *loop* infinito, onde o primeiro comando deste *loop* é a chamada à função bloqueante de `ThreadMonitor`. Quando for necessário usar uma destas *threads*, a classe `PoolManager` invoca a função de `ThreadMonitor` que acorda processos bloqueados, liberando o fluxo de execução do *loop* infinito. O próximo passo então é a *thread* executar uma tarefa qualquer, que é a implementação de uma função definida na interface `PooledRunnable` da Figura 5.6. Após a execução da função de execução definida na referida interface, a *thread* volta a dormir, invocando novamente o método bloqueante de `ThreadMonitor`. Desta forma, é possível manter *threads* vivas mas inativas, torná-las ativas quando necessário e fazê-las voltar a dormir enquanto nenhuma execução for solicitada.

A classe `PoolManager` trabalha com uma quantidade inicial de *threads* e pode, dependendo da demanda de solicitação simultânea de uso das *threads* inicialmente criadas, instanciar *threads* adicionais, até um certo limite máximo. O Quadro 5.14 mostra a lógica simplificada de execução da *thread* `PooledThread` e o Quadro 5.15 mostra as funções da classe `ThreadMonitor`.

```
while(true) {
    // Função bloqueante do monitor
    // O fluxo de execução é liberado quando a classe
    // PoolManager invoca a função de MonitorThread responsável
    // por acordar processos bloqueados
    MonitorThread.waitExecutionRequest();
    // Chama função da interface de processamento
    PooledRunabble.executeProcess();
    // FIM DO LOOP – volta para chamada à função bloqueante
}
```

Quadro 5.14 – Lógica de Execução de `PooledThread`

```
public void waitExecutorRequest() {
    wait();
}
public void wakeUpExecutor() {
    notifyAll();
}
```

Quadro 5.15 – Funções de `ThreadMonitor`

O modelo de funcionamento de `PooledThread` descrito prevê a independência dos controles inerentes à ativação e inativação de *threads*, em relação ao processo a ser executado por uma das *threads* controladas. Desta forma, ele se constitui em um componente independente do *Hermes*, sendo inclusive utilizado em outros contextos.

Uma vez que a mensagem é entregue a *thread* de processamento, disponibilizada pelo *pool* de *threads* cujo mecanismo foi descrito, aquela executa uma função do Autenticador de Mensagens, representado pela classe `AutenticadorMensagem` da Figura 5.6, que implementa a interface `PooledRunnable`, cuja função de execução é invocada pela *thread*.

A implementação do servidor de comunicação que utiliza o protocolo SOAP é feita pela classe `SOAPListener`, mostrada na Figura 5.7. Esta classe usa uma API COTS para Java, que implementa todos os pormenores do protocolo de comunicação SOAP, representada pela classe `GlueSOAPServer` da Figura 5.7. O resultado do uso desta API COTS é a abstração, pelo *Hermes*, de todo o processo de conversão de XMLs no padrão SOAP em objetos e vice-versa, e de toda implementação do código de recebimento da mensagem usando o protocolo http.

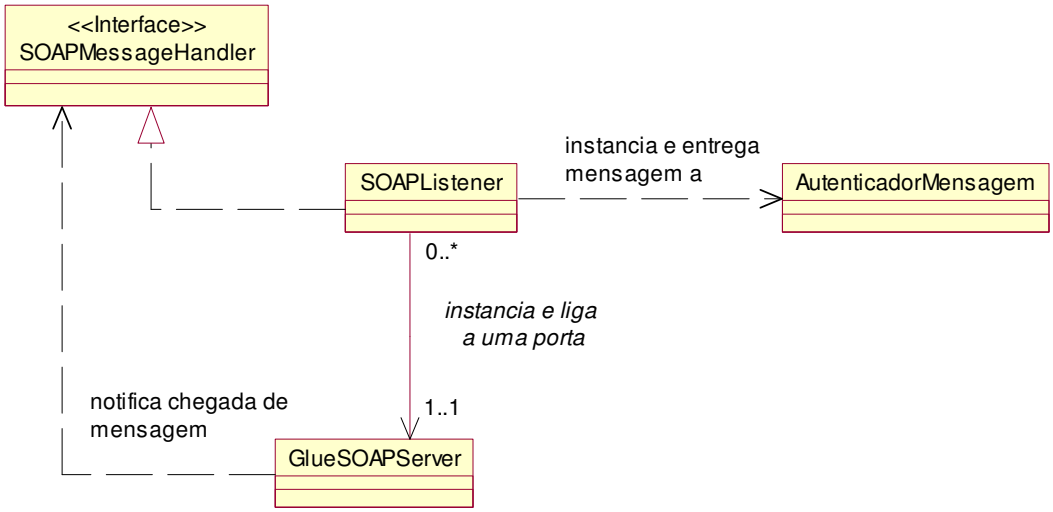

Figura 5.7 - Modelo das Classes Receptoras para Protocolo SOAP

O mecanismo de uso do servidor de comunicação SOAP prevê que um *handler* de uma mensagem SOAP, representado na Figura 5.7 pela interface `SOAPMessageHandler`, deve ser cadastrado no servidor SOAP, representado na Figura 5.7 pela classe `GlueSOAPServer`.

Quando uma mensagem for recebida por esta classe servidora, é encaminhada ao *handler* previamente cadastrado, a fim de ser tratada. A implementação do *handler* é a classe `SOAPListener`, mostrada na Figura 5.7, e o tratamento da mensagem é delegado ao Autenticador de Mensagens. Os controles internos referentes ao protocolo SOAP e ao controle de *threads* fica a cargo da implementação da classe `GlueSOAPServer`.

As inmplementações dos servidores de comunicação *socket* TCP e SOAP são usadas de forma mutuamente exclusiva. Uma instância de um receptor Hermes só pode utilizar servidores de comunicação *socket* TCP ou SOAP, e nunca os dois juntos, sendo

executados em uma mesma instância receptora. O tipo de servidor de comunicação a ser utilizado em uma instância receptora *Hermes* é definido através de uma configuração específica do *Hermes*.

5.5.2 Autenticador de Mensagens

O Autenticador de Mensagens é o elemento do módulo Receptor responsável por validar o conteúdo dos dados recebidos pelo *Listener*, de acordo com padrão estabelecido no modelo de mensagens do *Hermes*. Além destas conversões, o Autenticador de Mensagens converte as respostas de processamento para formatos a serem enviados de volta pelo Receptor ao transmissor da aplicação que envia as mensagens. A Figura 5.8 mostra o modelo do Autenticador de Mensagens.

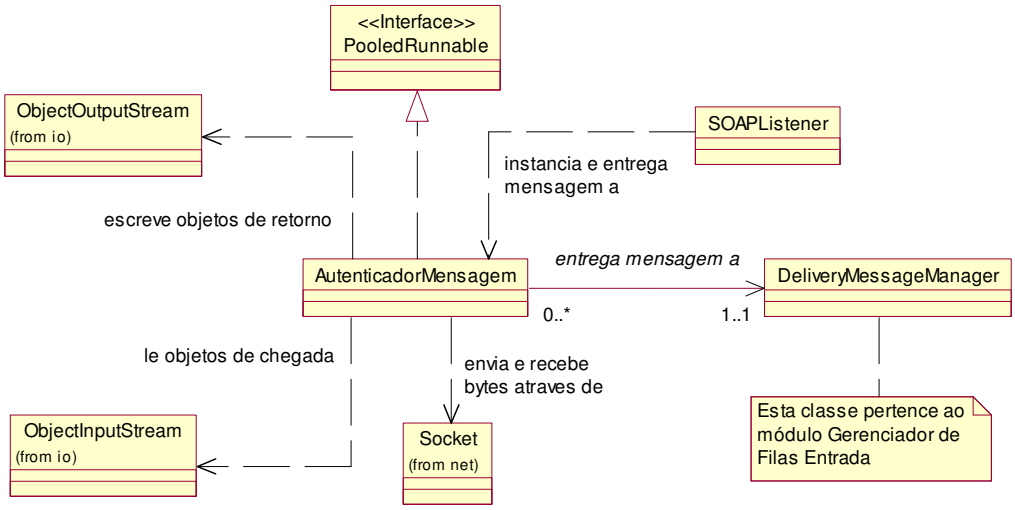

Figura 5.8 – Modelo do Autenticador de Mensagens

O Autenticador de Mensagens, representado na Figura 5.8 pela classe `AutenticadorMensagem`, desempenha diversos papéis na estrutura de recepção de mensagens dentro do módulo Receptor:

- Conversão dos bytes recebidos de uma conexão *socket* TPC para objetos correspondentes na linguagem de programação [35]. Para isto, o autenticador de mensagens usa uma classe da linguagem de programação responsável pelo algoritmo de conversão, `ObjectInputStream`, mostrada na Figura 5.8;

- Verificação de validade da mensagem. Para isto, é checado se o objeto recebido, seja via *socket* ou SOAP, implementa a interface *`Hermes`*`Message [7]`. Este teste é feito por uma checagem do tipo do objeto recebido. Se tal objeto não implementar *`Hermes`*`Message`, um objeto de retorno com esta indicação deve ser enviado de volta via *socket* TCP, ou retornado para a classe `SOAPListener` (mostrada na Figura 5.8). A validação da mensagem é uma simples verificação de sua aderência ao padrão de representação do *Hermes*. Não ocorre nenhum processo de autenticação

baseaado em chaves públicas ou privadas nem qualquer tipo de verificação da origem da mensagem;

- Encaminhamento de objetos, recebidos de uma conexão SOAP ou convertidos pelo próprio autenticador de mensagens, para processamento. O encaminhamento é feito através da entrega da mensagem ao módulo Gerenciador de Filas Entrada, representado na Figura 5.8 pela classe `DeliveryMessageManager`;

- Conversão do objeto de retorno de uma conexão *socket* TCP para um *array* de bytes correspondentes na linguagem de programação. Para isto, o autenticador de mensagens usa uma classe da linguagem de programação responsável pelo algoritmo de conversão [35], `ObjectoutputStream`, mostrada na Figura 5.8, para transformar o objeto retornado pela classe `DeliveryMessageManager`, em um *array* de bytes;
- Interação com o *socket* de uma conexão, a fim de enviar a resposta à solicitação de processamento de uma mensagem. Esta interação se dá através de um objeto do tipo `Socket [28]`, classe mostrada na Figura 5.8 que representa uma conexão *socket* TCP com um cliente *socket*; e

- Retorno para o `SOAPListener` do objeto resultado do processamento da mensagem. O objeto retornado pela classe `DeliveryMessageManager` é também retornado para a classe `SOAPListener`.

5.6 Módulo Gerenciador de Filas Entrada

Neste módulo, estão implementados os mecanismos de controle para processamento síncrono e assíncrono de mensagens. Estes mecanismos contemplam diversos pontos importantes que serão abordados nesta seção: encaminhamento das mensagens síncronas e assíncronas, representação das filas lógicas, criação e gerência dinâmica das filas lógicas associadas a tipos de mensagens ou a tópicos, persistência das mensagens e controle assíncrono de processamento.

Estes pontos estão implementados nos dois componentes lógicos do módulo Gerenciador de Filas Entrada, o Gerenciador de Entrega de Mensagens e o Controlador de Filas de Entrada.

5.6.1 Gerenciador de Entrega de Mensagens

Este componente é responsável pela implementação do encaminhamento de mensagens síncronas e assíncronas e pela criação e gerência dinâmica das filas lógicas associadas a tipos de mensagens ou a tópicos.

Para compreender o funcionamento das classes mostradas no diagrama da Figura 5.9, serão abordados em separado os dois mecanismos de processamento de mensagens possíveis em um receptor *Hermes*: os mecanismos síncrono e assíncrono. A classe `DeliveryMessageManager` da Figura 5.9 identifica, através de um indicador interno da mensagem, se o processamento da mesma será síncrono ou assíncrono.

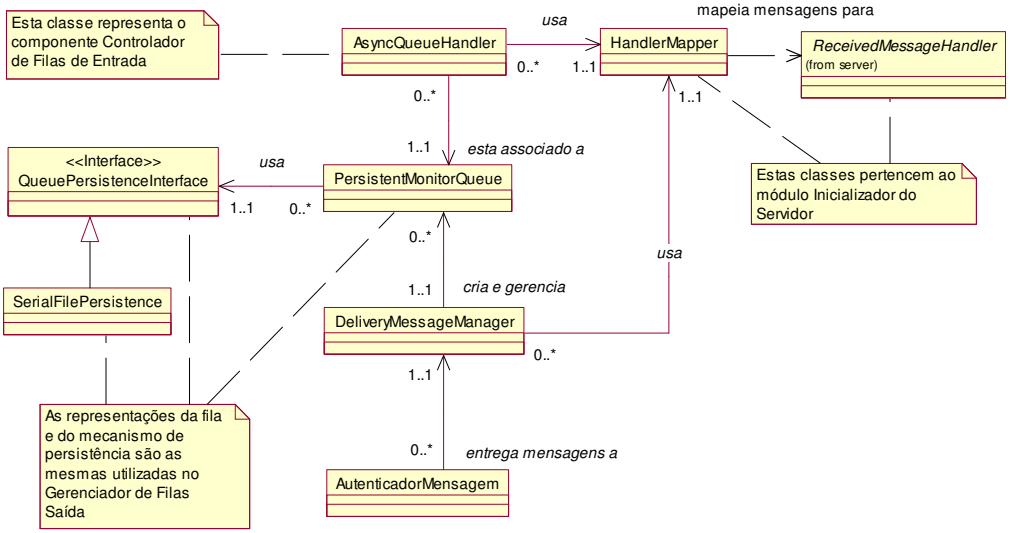

Figura 5.9 – Modelo do Gerenciador de Entrega de Mensagens

No caso do processamento ser síncrono, o `DeliveryMessageManager` interage diretamente com o módulo Controle da Aplicação, representado na Figura 5.9 pela classe `HandlerMapper`, responsável por mapear um *handler* responsável pelo processamento da mensagem, dado o tipo ou o tópico associado à mesma.

O processamento assíncrono é mais elaborado, pois envolve questões relativas ao gerenciamento das filas. A classe `DeliveryMessageManager` possui uma tabela interna populada com diversos objetos do tipo `PersistentMonitorQueue`, a classe que representa uma fila lógica no *Hermes*. Para cada objeto deste tipo constante na tabela, está associada uma chave, que é composta por um nome. Este nome pode representar um tópico ou um tipo.

Assim, para cada tópico ou tipo de mensagem, existe uma fila de entrada. A criação e eventual remoção destas filas é dinâmica, ou seja, à medida que novas solicitações de processamento de mensagens, associadas a novos tópicos ou de novos tipos, ainda não conhecidos pelo `DeliveryMessageManager`, for solicitado, novas filas vão sendo criadas e inseridas na tabela. Após a criação da nova fila, a mensagem é inserida na mesma. Se a chave, composta pelo nome, existir na tabela, a mensagem é inserida na fila que está associada a tal chave. Este mecanismo elimina o trabalho dos usuários do *Hermes* de criarem configurações de tópicos e de filas previamente, como acontece com alguns MOMs existentes.

Não há a preocupação de definição prévia das filas a serem gerenciadas, pois os próprios mecanismos do *Hermes* criam filas dinamicamente. Por outro lado, é necessário existir um controle de filas lógicas ociosas. Para isto, o *Hermes* destrói, periodicamente, as filas que estão vazias e inativas há mais de um determinado tempo.

Este mecanismo está implementado na classe `DeliveryMessageManager` que inicializa um processo independente de "*garbage collector*" para filas lógicas ociosas e vazias. Este processo verifica periodicamente, para cada fila, se ela está vazia e se a diferença entre a data e hora da última operação nela realizada e a data e hora atual é maior que um tempo pré-determinado. Em caso positivo, a fila é retirada da tabela que a relaciona com a chave nome. O critério de expiração das mensagens, anteriormente explicado na seção 4.4.4 que discorre sobre filas de saída, é válido também para as filas de entrada.

A Figura 5.10 mostra um diagrama de objetos que indica as relações entre a instância de `DeliveryMessageManager`, a instância da classe `Hashtable`, que representa uma tabela em memória, as instâncias de `PersistentMonitorQueue` e as `Strings` que representam as chaves das filas. A classe `Hashtable` é a implementação do relacionamento entre as classes `DeliveryMessageManager` e `PersistentMonitorQueue`, mostrado do diagrama da Figura 5.9.

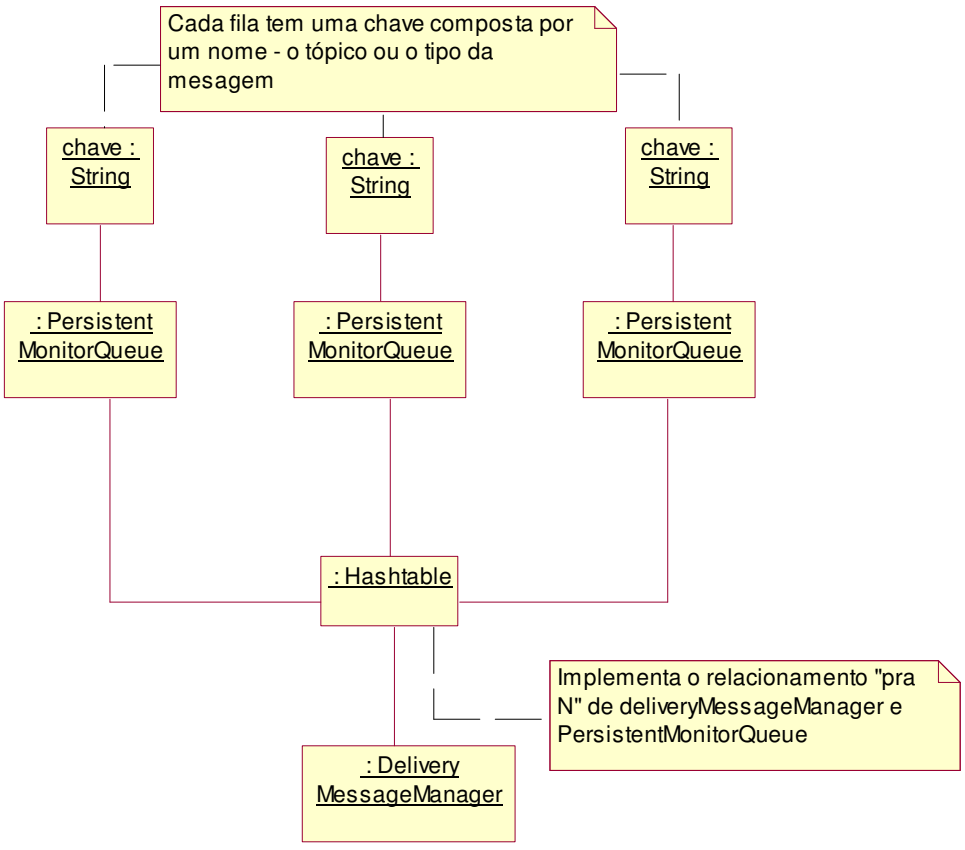

Figura 5.10 - Diagrama de Objetos para `DeliveryMessageManager` e Filas

5.6.2 Controlador de Filas de Entrada

Este componente é responsável pela implementação da representação das filas lógicas, da persistência das mensagens e do controle assíncrono de processamento. A representação das filas lógicas e o mecanismo de persistência são os descritos para o módulo Controlador de Filas Saída, que trata do Gerenciador de Transmissão Assíncrona e descreve as classes `PersistentMonitorQueue` e `SerialFilePersistence`, e a interface `PersistenceQueueInterface`. Estas classes constituem um componente de software independente do contexto do Gerenciador de Transmissão Assíncrona, e podem ser utilizadas como a representação de filas lógicas que monitoram processos e provêem persistência dos elementos a serem armazenados nas filas.

O controle assíncrono de processamento é realizado pela classe `AsyncQueueHandler`, mostrada na Figura 5.9, responsável por ler mensagens das filas de entrada e encaminhar as mesmas para os seus devidos tratadores, mapeados pelo módulo Controle da Aplicação através da classe `HandlerMapper`.

Os processos de inserção, leitura e remoção das filas de entrada, do ponto de vista funcional, é idêntico ao descrito para o módulo Controlador de Filas Saída, que trata do Gerenciador de Transmissão Assíncrona: o mecanismo produtor-consumidor com controle de concorrência no acesso às filas. O que muda em relação ao contexto do Controlador de Filas de Entrada são os papéis desempenhados, mostrados na Tabela 5.7.

Tabela 5.7 – Correspondência dos Papéis nos Processos Produtor-Consumidor

Papéis	Classe no Contexto do Gerenciador de Transmissão Assíncrona	Classe no Contexto do Controlador de Filas de Entrada
Produtor	`OutputQueueManager`	`DeliverYMessageManager`
Monitor	`PersistentMonitorQueue`	`PersistentMonitorQueue`
Consumidor	`AsynchronousSenderThread`	`AsyncQueueHandler`

Tal como foi descrito para o módulo Controlador de Filas Saída, a forma de funcionamento da interação mostrada na Figura 5.11 é a de um esquema produtor – consumidor. Cada fila é provida de mensagens por um processo produtor, a *thread* que executa a chamada do `insert()` pela classe `DeliverYMessageManager`, e tem suas mensagens consumidas por um processo consumidor, a *thread* que executa as chamadas do `read()` e do `remove()` pela classe `AsyncQueueHandler`.

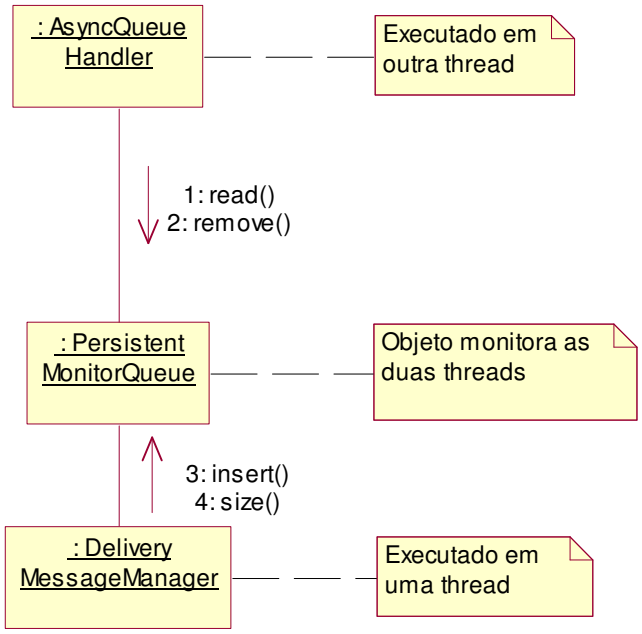

Figura 5.11 – Fila como Monitor de Objetos no Controlador de Filas de Entrada

O controle das chamadas aos métodos de `read()` e de `remove()`, feito pela classe `AsyncQueueHandler`, é implementado dentro de um *loop* infinito, que sempre lê, processa e remove mensagens da fila. O processamento da mensagem é justamente a entrega da mesma para seu *handler* através da classe `HandlerMapper`. A classe `AsyncQueueHandler` é uma *thread* que implementa uma função executada dentro do processo consumidor. Esta forma de interação está mostrada no Quadro 5.16, que apresenta a lógica resumida de funcionamento do processo consumidor da classe `AsyncQueueHandler`.

Uma mensagem é lida da fila e entregue ao `HandlerMapper` para processamento. O `HandlerMapper` pode retornar um dos quatro status listados:

- Processamento realizado com sucesso: A mensagem é removida da fila e o ciclo se repete para a próxima mensagem;

- Erro no processamento: `HandlerMapper` é invocado para avaliar o erro ocorrido e o retorno desta avaliação vai definir se a mensagem, mesmo processada com erro, vai ser retirada da fila ou não. Caso o retorno de `HandlerMapper` no tratamento do erro for retirar a mensagem da fila, aquela é removida e o ciclo inicial se repete para a próxima mensagem. Caso contrário, é verificado se a mensagem já foi expirada. Ocorrendo a expiração da mensagem, ela é inserida no repositório de mensagens mortas, removida da fila e o ciclo inicial se repete para a próxima mensagem. Se a mensagem ainda não estiver expirada, a *thread* que executa o processo descrito dorme por um tempo determinado e, após decorrido este tempo, o ciclo inicial se repete para a mesma mensagem, pois ela não foi removida da fila e será lida de novo;

- *Handler* não encontrado: É verificado se a mensagem já foi expirada. Ocorrendo a expiração da mensagem, ela é inserida no repositório de mensagens mortas, removida da fila e o ciclo inicial se repete para a próxima mensagem. Se a mensagem ainda não estiver expirada, a *thread* que executa o processo descrito dorme por um tempo determinado e, após decorrido este tempo, o ciclo inicial se repete para a mesma mensagem, pois ela não foi removida da fila e será novamente lida; e

- Outro retorno inesperado: A mensagem é inserida no repositório de mensagens mortas com um indicador que explicita ocorrência de erro inesperado, depois é removida da fila e o ciclo inicial se repete para a próxima mensagem.

```
// Loop infinito de processamento. O processo esta sempre
// disponível para processar mensagens
while(true){
     // Chamada ao método bloqueante
     HermesMessage mens = PersistentMonitorQueue.read();
     HandlerMapper.process(mens);
     if (retorno == OK) {
            // Retira mensagem da fila
            PersistentMonitorQueue.remove();
     } else if (retorno == ERRO) {
            // Notifica HandlerMapper de erro no processamento
            // e avalia retorno do tratamento de erro
            HandlerMapper.notifyException(erro,mens);
            if (retornoTratamentoErro == OK) {
                // Se mesmo com erro, aplicação quer
                // retirar mensagem, OK
                PersistentMonitorQueue.remove();
            } else if (mens.isExpired()) {
                insertIntoDeadQueueRepository(mens);
                PersistentMonitorQueue.remove();
            } else {
                // Se não retira, dorme para reprocessar
                // mensagem posteriormente
                sleep(TEMPO_REPROCESSAMENTO);
            }
     } else if (retorno == HANDLER_NAO_ENCONTRADO) {
            if (mens.isExpired()) {
                insertIntoDeadQueueRepository(mens);
                PersistentMonitorQueue.remove();
            } else {
                sleep(TEMPO_HANDLER_NAO_ENCONTRADO);
            }
     } else {
            insertIntoDeadQueueRepository(mens);
            PersistentMonitorQueue.remove();
     }
}
```

Quadro 5.16 – Lógica do Processo Consumidor de `AsyncQueueHandler`

O tratamento das mensagens assíncronas usa a forma de processamento por processo de espera, mas algumas abstrações de controle foram proporcionadas ao usuário do *Hermes*. A forma de processamento por processo de espera provê uma função que bloqueia o processo chamador da mesma, fazendo com que ele fique em estado de repouso (fora do processador) enquanto a fila de entrada estiver vazia.

Quando uma mensagem chega e é inserida na fila de entrada, o processo bloqueado é acordado e o retorno da função é justamente a mensagem a ser processada. Após o processamento da mensagem, o processo retira a mesma da fila e chama novamente a função bloqueante, voltando a ficar em estado de repouso até que outra mensagem seja inserida na fila de entrada. Tal processo fica em *loop* infinito, e em condições normais está sempre em repouso enquanto a fila estiver vazia. O Quadro 5.17 mostra um trecho de código simples que exemplifica o mecanismo descrito. Neste cenário, existem duas preocupações do desenvolvedor que está implementando o código mostrado no Quadro 5.17.

```
// Loop infinito de processamento. O processo esta sempre
// disponível para processar mensagens
while(true){
     // Chamada ao método bloqueante
     Mensagem mens = FILA.readMessage();
     // Código para processar a mensagem
     .........
    // Retira mensagem da fila
    FILA.remove(mens);
}
```

Quadro 5.17 – Lógica de Processamento da Fila de Entrada – Processo em Espera

A primeira preocupação está relacionada com a criação de um processo próprio dentro do programa, seja através da instanciação de objetos que representam *threads* ou processos paralelos de execução, seja através de APIs próprias da linguagem de programação utilizada. É este processo, independente de outros que são executados dentro da aplicação servidora, que vai realizar o processamento das mensagens associadas a uma determinada fila de entrada, executando o código mostrado no Quadro 5.17. Para cada fila de entrada, tem que existir um processo em execução, que executa um *loop* infinito e uma chamada a uma função bloqueante. O programador tem que definir e criar explicitamente todo e qualquer processo que se proponha a tratar mensagens nas diferentes filas de entrada, considerando a referência `FILA` no Quadro 5.17 a representação de uma fila de entrada.

A segunda preocupação diz respeito à lógica de controle implementada dentro do *loop* infinito. Se não houver cuidado na implementação desta lógica, o fluxo do programa pode levar a uma situação onde a mensagem nunca é retirada da fila e o *loop* infinito deixa de ser bloqueante, pois enquanto houver mensagem na fila a função retorna a primeira mensagem da fila, liberando o fluxo de processamento. As conseqüências deste fato são uma maior utilização de CPU e de recursos de máquina e o risco de parar uma fila por não processamento de uma dada mensagem.

A classe `AsyncQueueHandler` faz o papel de um componente que realiza todo o controle de fluxo do *loop* infinito, invocando duas funções de `HandlerMapper`. A primeira se destina a processar efetivamente a mensagem. A segunda trata situações de exceção. Dependendo do retorno dos dois métodos, a mensagem é retirada da fila, é reprocessada após um tempo pré-determinado ou é removida da fila por ter expirado.

Desta forma, o risco de ocorrerem situações onde um caso de exceção provoque uma sobrecarga de processamento fica a cargo de uma rotina genérica, controlada pelo próprio *Hermes*. Esta abordagem do *Hermes* garante a independência do processamento da mensagem em relação ao fluxo que controla as operações de leitura e remoção das mensagens de suas respectivas filas.

A redução de complexidade do código a ser implementado quando se usa o *Hermes* para processar mensagens assíncronas no modo processo em espera é significativa. Tal código é a implementação de duas funções em subclasses da classe `ReceivedMessageHandler`, que representa um *handler* genérico. Como esta classe é abstrata, há a obrigação do programador de implementar sempre estas duas funções [15][35].

No processamento assíncrono, a mensagem é encaminhada pelo mecanismo do `AsyncQueueHandler` para processamento ao `HandlerMapper`, que mapeia o *handler*, implementado na aplicação.

5.7 Módulo Controle da Aplicação

O módulo Controle da Aplicação é responsável por receber mensagens a serem processadas e mapear o *handler* associado, encaminhando ao mesmo a mensagem. Além desta função, o Controle da Aplicação inicializa o Serviço de Monitoramento, que responde a solicitações externas de informações sobre o status das filas de entrada existentes no receptor *Hermes* onde tal serviço é executado, e executa um processo de registro de nomes associado ao receptor.

Do ponto de vista funcional, é possível dividir o Controle da Aplicação em três elementos distintos: o Mapeador de *Handlers*, o Registrador de Nomes e o Serviço de Monitoramento. Cada um destes três elementos desempenha um papel diferente no funcionamento do Controle da Aplicação.

5.7.1 Mapeador de Handlers

Este componente é responsável por realizar o mapeamento de uma mensagem a um *handler* encarregado de processá-la. A classe que representa este componente é `HandlerMapper`, referenciada nos modelos da Figura 5.9. Interagem com ele os elementos do Gerenciador de Filas Entrada responsáveis por encaminhar as mensagens para processamento síncrono ou assíncrono. Nos dois casos, o papel de

`HandlerMapper` é sempre mapear mensagens em tratadores e encaminhar a mensagem a este último, para que haja o efetivo processamento pela aplicação que, em última instância, implementa os *handlers*.

O funcionamento da classe `HandlerMapper` se baseia no modelo de mensagens descrito no Capítulo 4. Para realizar o mapeamento de mensagens em *handlers*, é necessário conhecer previamente as associações entre *handlers* e tipos de mensagens e entre *handlers* e tópicos.

Estas associações estão definidas no arquivo XML (`ConfHandlers.xml`) de configuração descrito no Capítulo 4, que associa um handler a "N" tópicos e ou "N" tipos de mensagens. A classe `HandlerMapper`, quando inicializada, lê este arquivo e instancia todos os *handlers* lá definidos, associando a cada um deles uma lista de tipos de mensagens e uma lista de tópicos.

Estas associações são guardadas em memória na estrutura de duas tabelas. Uma delas tem como chave de suas entradas os nomes dos tópicos e a outra tem como chave de suas entradas os tipos de mensagens. A Figura 5.12 mostra a relação da classe `HandlerMapper` com as duas tabelas.

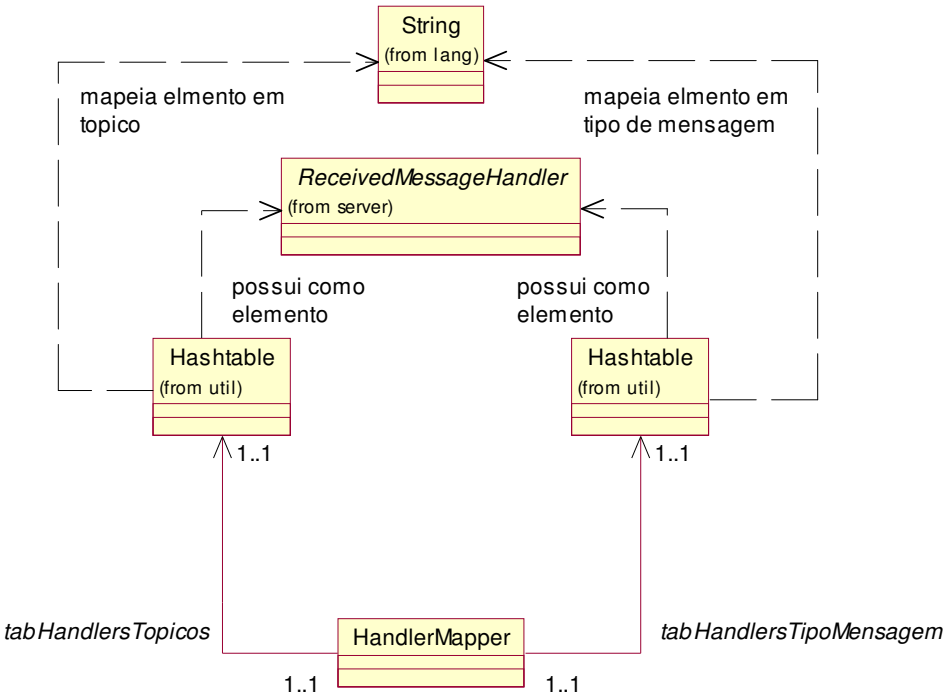

Figura 5.12 – Relação entre `HandlerMapper` e as duas Tabelas de *Handlers*

No modelo da Figura 5.12, a classe `Hashtable` representa uma estrutura de dados do tipo *set*, onde é possível mapear chaves em objetos. `HandlerMapper` popula as duas tabelas com as relações entre *handlers* x tópicos e *handlers* x tipos de mensagens lidas do arquivo XML de configuração. Com as tabelas populadas em memória, é possível

`HandlerMapper` fazer a associação de um *handler* a uma mensagem, avaliando o seu tipo ou tópico associado.

`HandlerMapper` verifica se a mensagem está associada a algum tópico, um dos atributos da mensagem. Em caso positivo, a tabela de *handlers* e tópicos e acessada. Se houver *handler* associado ao tópico, a mensagem é entregue ao primeiro para processamento. Se nenhum *handler* estiver associado ao tópico, a tabela de *handlers* e tipos é acessada. Se houver *handler* associado ao tipo, a mensagem é entregue ao primeiro para processamento. Caso contrário, `HandlerMapper` retorna um status de *handler* não encontrado. No caso da mensagem não estar associada a nenhum tópico, `HandlerMapper` acessa a tabela de *handlers* e tipos é acessada. Se houver *handler* associado ao tipo, a mensagem é entregue ao primeiro para processamento. O Quadro 5.18 mostra a lógica da função `process()` da classe `HandlerMapper`, responsável por implementar a lógica descrita. A função mostrada sinaliza, em qualquer situação, o status de processamento ou de não encaminhamento da mensagem.

```
String topico = HermesMessage.getTopic();
String tipo = HermesMessage.getType();
if (topico != null) {
   if (tabHandlersTopicos.exists(topico)) {
     ReceivedMessageHandler rmh = tabHandlersTopicos.get(topico);
     rmh.process(HermesMessage);
     // Retorna status do processamento
   } else if (tabHandlersTipos.exists(tipo)) {
     ReceivedMessageHandler rmh = tabHandlersTipos.get(tipo);
     rmh.process(HermesMessage);
     // Retorna status do processamento
   } else {
     // Retorna handler não encontrado
   }
} else if (tabHandlersTipos.exists(tipo)) {
     ReceivedMessageHandler rmh = tabHandlersTipos.get(tipo);
     rmh.process(HermesMessage);
     // Retorna status do processamento
} else {
     // Retorna handler não encontrado
}
```

Quadro 5.18 – Lógica de Processamento do `HandlerMapper`

Além de encaminhar mensagens para processamento, o `HandlerMapper` encaminha as solicitações de tratamento de exceções para os *handlers* definirem se a mensagem será retirada da fila de entrada ou se ela será posteriormente reprocessada. A lógica de mapeamento usada pelo `HandlerMapper` é a mesma descrita para o processamento. O que muda são as chamadas às funções do handler, que, em vez de processar, vai avaliar se uma exceção causa a remoção da mensagem da fila de entrada ou uma espera para posterior reprocessamento. A função de `HandlerMapper` que implementa tal lógica é a `notifyException()`, que recebe a exceção ocorrida e a mensagem associada. O Quadro 5.19 mostra a lógica da referida função.

```
String topico = HermesMessage.getTopic();
String tipo = HermesMessage.getType();
if (topico != null) {
   if (tabHandlersTopicos.exists(topico)) {
     ReceivedMessageHandler rmh = tabHandlersTopicos.get(topico);
     rmh.handleException(HermesMessage,Exception);
     // Retorna status do handleException()
   } else if (tabHandlersTipos.exists(tipo)) {
     ReceivedMessageHandler rmh = tabHandlersTipos.get(tipo);
     rmh.handleException(HermesMessage,Exception);
     // Retorna status do handleException()
   } else {
     // Retorna status de reprocessamento da mensagem
   }
} else if (tabHandlersTipos.exists(tipo)) {
     ReceivedMessageHandler rmh = tabHandlersTipos.get(tipo);
     rmh.handleException(HermesMessage,Exception);
     // Retorna status do handleException()
} else {
     // Retorna status de reprocessamento da mensagem
}
```

Quadro 5.19 – Lógica de Tratamento de Exceção do `HandlerMapper`

Em tese, as situações de *handler* não encontrados nunca vão ocorrer, pois uma exceção no processamento só é causada pelo *handler* responsável por processar a mensagem. Ou seja, quando há uma notificação de exceção, já houve, em um momento anterior, a associação da mensagem a um *handler*.

5.7.2 Registrador de Nomes

O Registrador de Nomes é responsável por interagir com o Serviço de Nomes para registrar o seu endereço físico (pares IP x porta) no Serviço de Nomes do *Hermes*. O Serviço de Nomes é uma parte do *Hermes* independente da API transmissora e dos módulos do receptor.

Sendo assim, ele é executado como um programa separado das aplicações que utilizam os módulos do transmissor e do receptor *Hermes*, e é usado para manter o registro de endereços físicos e da forma de endereçamento de mensagens para um dado serviço lógico, unicamente identificado por um nome. Portanto, o Serviço de Nomes provê um servidor que permite a realização de registros de endereços físicos e formas de endereçamento de mensagens associadas a um dado serviço lógico.

O Registrador de Nomes utiliza este servidor do Serviço de Nomes para se registrar, associando seu endereço físico a um nome lógico. A forma de interação entre o Registrador de Nomes e tal servidor é um processo de comunicação entre aplicativos baseado no protocolo CORBA, com um modelo de objetos representativos da semântica

associada a um par endereço físico x endereço lógico e à forma de endereçamento das mensagens. Assim, o Registrador de Nomes é uma implementação de um cliente CORBA. As formas de endereçamento das mensagens são apresentadas com mais detalhes no Capítulo 4, que discorre sobre o Serviço de Nomes.

O Registrador de Nomes realiza a operação de registro na inicialização da aplicação receptora, e pode interagir com "N" serviços de nomes. Para isto, ele lê um arquivo de configuração que possui a lista de endereços dos serviços de nomes e se registra em todos eles. Se nenhum processo de registro for realizado com sucesso, o registrador de nomes sinaliza para a aplicação receptora que não houve registro do serviço *Hermes* em nenhum servidor de nomes cadastrado no arquivo de configuração. Se pelo menos um registro foi feito, não há sinalização de exceção para a aplicação receptora.

5.7.3 Serviço de Monitoramento

O Serviço de Monitoramento é um componente do *Hermes* responsável por prover um servidor de informações sobre os estados das filas de entrada em um receptor *Hermes*. Basicamente, este servidor de informações sobre os estados das filas de entrada tem o papel de responder às consultas que os clientes de monitoramento realizam. Nestas consultas, são solicitadas informações sobre as filas de entrada do receptor *Hermes* onde o Serviço de Monitoramento está sendo executado.

Desta forma, o Serviço de Monitoramento no receptor *Hermes* faz exatamente o mesmo papel que o Serviço de Monitoramento do módulo API *Hermes*, presente na aplicação transmissora, realiza. A única diferença é o componente a ser monitorado, pois os objetos de monitoramento e as informações a serem providas também são rigorosamente iguais. São monitoradas filas de entrada no lado receptor e filas de saída no lado transmissor. Os dados a serem fornecidos são os mesmos para filas de entrada e de saída.

Portanto, o componente de Serviço de Monitoramento usado no módulo Controle da Aplicação é exatamente igual ao utilizado no módulo API *Hermes*. A única mudança é que, em vez deste componente monitorar o Controlador de Filas Saída no lado transmissor, vai monitorar o Controlador de Filas Entrada no lado receptor. Para isto, a interface de monitoramento deve ser implementada pelo Controlador de Filas de Entrada, representado pela classe `AsyncQueueHandler`.

A Figura 5.13 mostra que o modelo do Serviço de Monitoramento para o lado receptor é o mesmo utilizado no lado transmissor. A única mudança diz respeito ao componente que implementa a interface de monitoramento `FifoMonitorable`. No lado receptor, a classe implementadora é `AsyncQueueHandler`. Como o sistema de monitoramento como um todo é independente dos elementos monitorados, todo o modelo é reaproveitado no contexto do lado receptor.

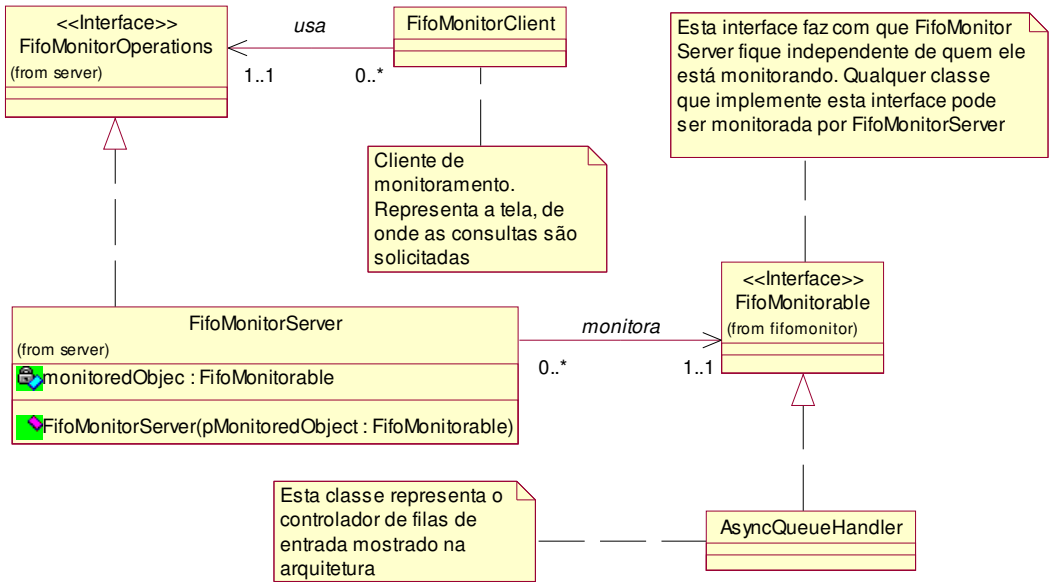

Figura 5.13 – Modelo do Sistema de Monitoramento de Filas – Lado Receptor

5.8 Considerações Finais

Este capítulo apresentou os detalhes e aspectos de implementação do MOM *Hermes*, seguindo a estrutura *top-down* [37] de apresentação proposta e complementando o Capítulo 4, que apresentou os aspectos gerais do *Hermes*, a sua arquitetura e duas de suas características funcionais importantes. O *Hermes* é um componente de software complexo, que utiliza tecnologias relacionadas com sistemas de comunicação, com engenharia de software e com linguagens de programação.

Foram utilizadas três tecnologias de comunicação disponíveis para a construção das funcionalidades do *Hermes*: CORBA, nos serviços de monitoramento e no servidor de nomes; *socket* TPC no processo de comunicação entre transmissores e receptores; e SOAP, como alternativa de protocolo para o *socket* TCP.

A utilização alternativa do SOAP levou o *Hermes*, inicialmente implementado na linguagem de programação Java, a ser convertido para C# .NET [20]. Esta conversão foi realizada por uma ferramenta conversora Java – C#. O resultado foi a corretude de 80% do código convertido. Os 20% restantes foram convertidos manualmente. A versão do *Hermes* em C# possui a limitação de não suportar serviços de monitoramento e servidor de nomes, mas incorpora todas as outras funcionalidades. Versões do *Hermes* em C# e em Java permitem um maior grau de interoperabilidade entre aplicações que fazem uso do *Hermes* como um sistema de troca de mensagens.

Os modelos de classes e de objetos do *Hermes* foram desenvolvidos no padrão UML, seguindo os preceitos modernos de engenharia de software [3][37]. O *Hermes* é modular

e componentizado [14], sendo destacados inclusive alguns componentes independentes do contexto de um MOM: *pool* de *threads*, fila e sistema de monitoramento. Os outros componentes são bem definidos, com atribuições específicas e claramente delimitadas.

Muitas das características do *Hermes* utilizam recursos modernos e inovadores das linguagens de programação Java e C#. Mecanismo de controle de *threads*, representação de objetos em bytes, controle de concorrência por objetos monitores e APIs de comunicação foram alguns dos recursos utilizados.

O *Hermes* apresenta algumas inovações e otimizações das implementações e características de MOMs:

- Funções de balanceamento de carga e de contingenciamento de rotas associadas ao Serviço de Nomes, com algumas características adicionais;

- Flexibilidade nos tipos de transmissão, com a possibilidade de usar qualquer combinação disponível no *Hermes*;

- Tipos de mensagens associados a tópicos;

- Criação e gerência dinâmica de filas associadas a tópicos e a tipos de mensagens;

- Aumento de disponibilidade com o uso de múltiplas portas por instância de receptor, que permite a diminuição da latência de processamento das mensagens nos servidores de comunicação *socket* TCP associados às portas lógicas de uma instância de aplicação; e

- Facilidade de codificar os mecanismos de recepção de mensagens que são processadas por meio de processo em espera.

Capítulo 6

Estudo de Caso

Este capítulo apresenta um estudo de caso do Hermes em um sistema de automação comercial.

6.1 Introdução

Este capítulo apresenta um estudo de caso de uso do *Hermes* em um sistema corporativo de automação comercial. Inicialmente, são apresentados o contexto de uso do *Hermes* e a arquitetura associada a este contexto. Depois, um breve resumo do parque instalado deste contexto é mostrado e são descritos os cenários onde o *Hermes* está presente, analisando-se a aderência das características do *Hermes* aos requisitos de cada cenário. Finalmente, são mostrados os benefícios que as implementações e melhorias do *Hermes* trouxeram nos cenários apresentados.

6.2 Contexto de Uso do Hermes

No contexto do estudo de caso a ser apresentado, o *Hermes* faz o papel de um componente de software dentro de uma solução corporativa voltada para o varejo, doravante denominada SAL (Sistema de Automação de Lojas). A função geral do *Hermes* é a de prover um mecanismo de troca de mensagens entre os diversos módulos da solução.

O SAL possui diversas funcionalidades que exigem um mecanismo de troca de mensagens. Cada uma destas funções tem requisitos e necessidades específicas, tanto do ponto de vista funcional quanto do ponto de vista de performance e de disponibilidade. Para entender o que o *Hermes* deve atender dentro desta solução, é necessário o conhecimento da arquitetura da mesma, e do ambiente onde ela é utilizada.

O SAL contempla um conjunto de aplicativos que atendem a todas as necessidades relativas à gerência de informações e à automação de vendas de uma loja. Soluções desta natureza são designadas como sistemas de automação comercial. Os sistemas de automação comercial são utilizados nos mais variados tipos de estabelecimentos comerciais existentes: supermercados, livrarias, sapatarias, lojas de departamentos, revendas de material de construção, lojas de telefones celulares e muitas outras.

Cada uma destas categorias de estabelecimentos comerciais possui requisitos funcionais específicos a serem atendidos, mas uma grande parte das funcionalidades existentes em uma solução de automação comercial pode ser comumente utilizada por estabelecimentos de todas as categorias existentes.

6.2.1 Características Gerais

O SAL disponibiliza um conjunto de funcionalidades comuns a todos os tipos de estabelecimentos comerciais, permitindo, por conta das características de sua arquitetura, que novas funcionalidades específicas sejam incorporadas à solução sem interferir no que já está disponível. Muitas vezes, a própria versão padrão, a que possui apenas as funcionalidades comuns, atende aos requisitos do contexto onde ela será utilizada. O SAL

precisa executar algumas funcionalidades básicas para prover mecanismos de automação comercial e para permitir a gerência de informações de um estabelecimento comercial. Tais funcionalidades são:

Venda de Produtos e de Serviços
Compreende todos os procedimentos de contabilização das vendas em um PDV (Ponto De Venda), os caixas registradores, presentes em qualquer estabelecimento comercial. Estes procedimentos são: a identificação dos preços dos produtos ou serviços, a contabilização fiscal, a impressão do cupom de venda, a totalização dos produtos e serviços de um cupom de venda, o procedimento de recebimento (em dinheiro, cheque, cartão, etc.) e a transmissão das informações de venda para um servidor central. Em algumas situações, os PDVs realizam operações adicionais ao processo de venda comum, como: consulta *real time* de estoque, carga *on line* de créditos em telefones celulares e outras possíveis operações.

Processamento das Vendas
Compreende a gravação *on line* dos resultados analítico e sintético das vendas, registrando dados como os itens da venda, os valores de cada item vendido, os valores dos impostos recolhidos, as formas de pagamento utilizadas no recebimento, os valores pagos em cada uma das formas e outros dados inerentes ao processo de vendas. Estas informações alimentam um sistema de consulta aos dados das vendas de um estabelecimento, permitindo que relatórios analíticos e gerenciais seja emitidos com os dados processados.

Envio das Vendas para um Sistema Corporativo
Compreende o processo de integração das vendas, que se dá entre o sistema de automação comercial e um sistema de gestão corporativa. Na maioria das vezes, sistemas de automação comercial devem enviar as informações sobre as vendas realizadas e processadas ao sistema de gestão corporativa.

Distribuição de Preços
Compreende o processo de atualização do cadastro de preços no módulo de retaguarda e a distribuição desta atualização para o cadastro de preços existentes nos PDVs.

6.2.2 Arquitetura

Além das funções enumeradas na seção anterior, o SAL contempla as funcionalidades de manutenção dos cadastros, a geração do arquivo de atualização de preços, a emissão de relatórios gerenciais e analíticos, a execução de rotinas operacionais diárias e o processamento das consultas aos estados de todos os sub sistemas da solução. Cada funcionalidade citada está implementada em sub-módulos independentes e distintos.

A arquitetura da solução prevê um servidor central, localizado normalmente no escritório da matriz ou no CPD, capaz de gerenciar, de forma centralizada, "N" lojas, processando vendas e distribuindo preços. A Figura 6.1 mostra a arquitetura da solução.

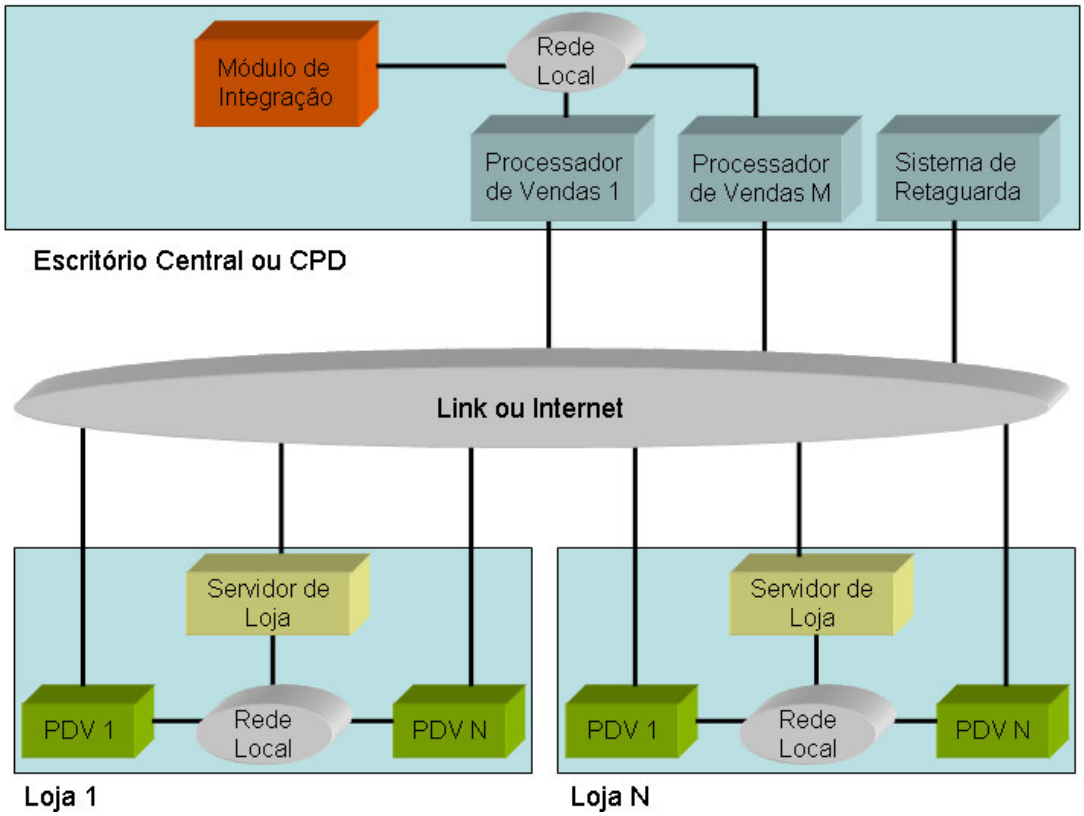

Figura 6.1 – Arquitetura da Solução de Varejo

Com base na arquitetura da Figura 6.1, se destacam os seguintes elementos e suas funções específicas:

- PDV: Realiza o processo de venda e transmite os resultados das vendas para o Processador de Vendas, podendo fazer isto diretamente pelo *link* / Internet, ou através do Servidor de Loja;

- Servidor de Loja: Controla o processo de distribuição de preços para os PDVs e pode, eventualmente, transmitir as vendas de todos os PDVs de uma loja para o Sistema de Retaguarda;

- Processador de Vendas: Recebe e processa as vendas e envia as mesmas para o Módulo de Integração;

- Sistema de Retaguarda: Envia atualizações de preços para os Servidores de Lojas, realiza manutenção dos cadastros, emite relatórios gerenciais e analíticos, executa rotinas operacionais diárias e processa consultas aos estados de todos os sub-sistemas da solução; e

- Módulo de Integração: Recebe as vendas do Processador de Vendas e as integra com o sistema de gestão corporativa.

O papel do *Hermes* nestes elementos é prover os mecanismos de comunicação para realizar as funções de transporte dos dados de um sub-sistema para outro sub-sistema. As funcionalidades gerais onde o *Hermes* está presente são:

- Transmissão de vendas do PDV para o Processador de Vendas;
- Transmissão de vendas do Processador de Vendas para o Módulo de Integração; e
- Transmissão de informações de controle das atualizações de preços do Sistema de Retaguarda para o Servidor de Loja e do Servidor de Loja para os PDVs.

Em cada um destes processos de transmissão de mensagens, existem requisitos distintos que fazem com que o *Hermes* seja utilizado de diversas formas diferentes, aproveitando-se as combinações dos tipos de transmissão e das formas de endereçamento descritas no Capítulo 4.

6.3 Cenários de Uso do Hermes

Esta seção apresenta um resumo do parque instalado do SAL / *Hermes*, e depois descreve os cenários do SAL onde o *Hermes* é utilizado como um mecanismo de troca de mensagens. Três destes cenários, mencionados na seção anterior, são aplicáveis a todo o parque instalado. Um cenário adicional descrito é específico.

6.3.1 Parque Instalado

O SAL é atualmente a solução de automação comercial para onze companhias de varejo de categorias diversas. Em todos estes casos, o *Hermes* é utilizado como mecanismo de troca de mensagens, presente nos três contextos apresentados na Seção 6.2.2. Em alguns casos, o *Hermes* é utilizado em funções específicas que exigem um mecanismo de troca de mensagens. A Tabela 6.1 resume o parque instalado do SAL e do *Hermes*.

Tabela 6.1 – Parque Instalado do SAL / Hermes

Tipo de Companhia	Quantidade de Cupons / Dia	Quantidade Média de Itens do Cupom	Quantidade de Lojas	Quantidade de PDVs
Lojas de telefone celular	3000	02	50	150
Livraria e Papelaria	10500	06	32	1480
Supermercado	7000	16	04	70
Lojas de tecidos	2000	05	08	24
Sapataria	1250	02	06	35
Material de construção	500	08	01	08
Loja de departamentos*	223000	04	315	4825

* Algumas lojas em fase de implantação.

O resumo da Tabela 6.1 indica o volume de vendas e o fluxo de dados que passa pelo *Hermes*. A quantidade de cupons por dia é a quantidade de vendas transmitidas. A quantidade média de itens do cupom indica quantos itens (produtos) uma venda possui. Quanto mais itens, maior volume de dados tem que ser transmitido.

Sem levar em conta as duas lojas de departamento que estão em fase de implantação do SAL, a solução opera em onze companhias diferentes, com características funcionais, de carga e de infra-estrutura diferentes, comportando quatrocentos e cinqüenta PDVs em cento e seis lojas, e processando em torno de vinte e sete mil cupons por dia.

6.3.2 Transmissão de Vendas para o Processador de Vendas

Para transmissão de vendas da loja para o Processador de Vendas, o SAL provê duas alternativas de transmissão: uma rota direta entre os PDVs e o Processador de Vendas ou uma rota de dois pontos – do PDV para o Servidor de Loja e deste para o Processador de Vendas. Dependendo do fluxo de dados previsto e das condições do *link* / Internet, pode ser vantagem optar por uma ou outra alternativa.

Para uma situação onde o *link* / Internet seja de qualidade (velocidade alta) e o fluxo de dados não seja muito intenso, é melhor adotar a rota direta PDV – Processador de Vendas, pois o meio de transmissão não se tornaria um gargalo no processo de envio. Para uma situação onde o *link* / Internet não seja de qualidade (velocidade média ou baixa) e o fluxo de dados mais intenso, é melhor adotar a rota indireta PDV – Servidor de Loja – Processador de Vendas, porque o Servidor de Loja pode atuar como um concentrador e atenuador do fluxo simultâneo de dados no *link* / Internet.

Esta condição da rota indireta implica que o processo de transmissão das vendas de todos os PDVs de uma loja seja concentrado em apenas um ponto, que sequencializa a transmissão das vendas para o Processador de Vendas por ordem de chegada das vendas. Neste caso, tem que haver uma fila de transmissão no Servidor de Loja, caracterizando o *Hermes* como um transmissor assíncrono para vendas.

O processo de venda no PDV requer prioridade total às operações inerentes ao fluxo principal do processo de venda. Este fluxo principal é composto dos passos necessários para liberar o mais rápido possível um consumidor que está aguardando a contabilização de suas compras. Com base nesta premissa, pode-se imaginar que não é importante para um consumidor aguardar que a venda por ele realizada seja transmitida para o Processador de Vendas. E pior, que ele corra o risco de esperar caso este processo de transmissão, por problemas inerentes à rede, interrompa o fluxo principal de venda. Assim, o processo de transmissão de venda no PDV tem que ser obrigatoriamente assíncrono e independente do fluxo principal de vendas.

Além disso, vendas que já tiverem seu consumidor liberado e com o cupom de venda na mão, têm que obrigatoriamente ser transmitidas para o Processador de Vendas, mesmo que a aplicação PDV seja desligada entre o encerramento da venda e o processo de transmissão. Se o *Hermes* for usado como transmissor assíncrono, a única operação que o fluxo principal de venda tem que executar com relação à transmissão da venda é inserir a mesma em uma fila de saída. O *Hermes* então é usado no PDV como um transmissor assíncrono para vendas.

Um dos requisitos do processamento de vendas no Processador de Vendas é que a venda só saia do PDV quando ela for processada, que implica na gravação do cupom eletrônico em um banco de dados, e na atualização das informações sumarizadas das vendas. Desta forma, o processamento da venda no Processador de Vendas tem que ser síncrono. Problemas de acesso ao banco de dados do Processador de Vendas ou problemas de processamento levam as vendas a serem mantidas nos PDVs.

Qualquer problema de transmissão ou de processamento das vendas implica em um processo de retransmissão das mesmas para o Processador de Vendas. O tempo de expiração de uma venda é normalmente de uma semana. Após este período, mecanismos de contingência são acionados para enviar as vendas ao Processador de Vendas.

Na rota PDV – Servidor de Loja – Processador de Vendas, o processamento da venda no Servidor de Loja é inserir a venda na fila de transmissão para o Processador de Vendas. A venda não deve sair do PDV enquanto a inserção dela na fila de transmissão do Servidor de Loja não for confirmada. Assim, o *Hermes* no Servidor de Loja deve ser usado como um receptor síncrono para vendas.

A Figura 6.2 mostra a configuração de filas e as modalidades de uso do *Hermes* na transmissão de vendas do PDV para o Processador de Vendas. A rota da esquerda contempla a transmissão passando pelo Servidor de Loja e a rota da direita contempla a transmissão direta.

O *handler* do Servidor de Loja repassa a venda para uma transmissão assíncrona. O *handler* do Processador de Vendas realiza as operações de processamento de vendas (gravação do cupom eletrônico no banco de dados e atualização de informações sumarizadas sobre as vendas) e envia as vendas para o Módulo de Integração.

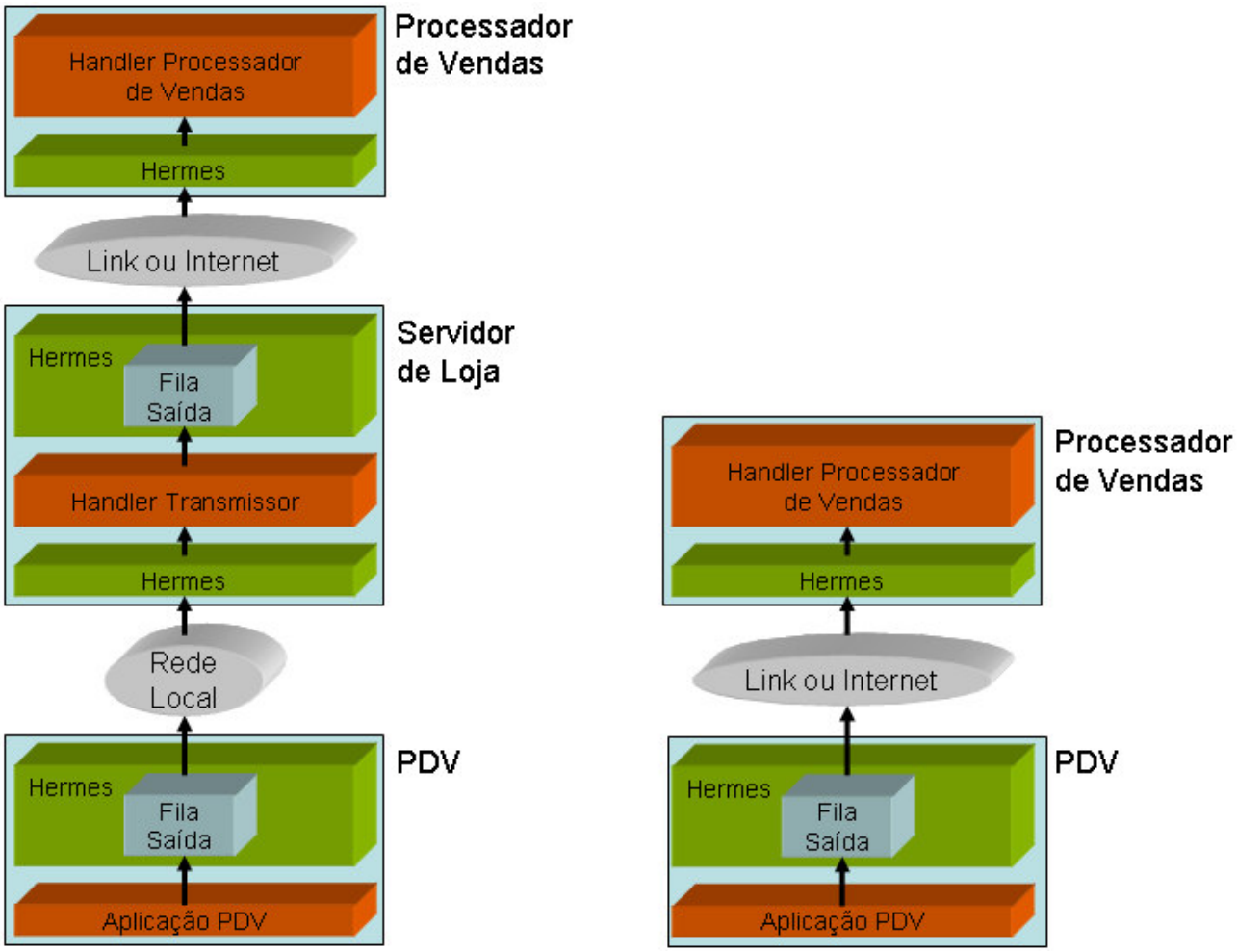

Figura 6.2 – *Hermes* na Transmissão de Vendas para o Processador de Vendas

Como o SAL é uma solução aplicável a qualquer contexto de varejo, é necessário que ele possa ser usado em ambientes com um pequeno volume de vendas e poucas lojas, e em ambientes com um grande volume de vendas e muitas lojas. Assim, é necessário que o mecanismo de transmissão das vendas seja escalável.

A escalabilidade no SAL é provida por dois elementos. O primeiro é a arquitetura do Processador de Vendas. Como ele é um módulo independente, várias instâncias suas podem ser executadas em máquinas diferentes simultaneamente. O segundo é o mecanismo de balanceamento de carga do *Hermes*. Com tal mecanismo, é possível que um Servidor de Nomes disponibilizem sucessivas listas alternadas de endereços dos Processadores de Vendas para PDVs ou para Servidores de Loja, permitindo uma distribuição de carga relativamente eqüitativa entre os Processadores de Vendas.

Além da escalabilidade, é importante que as vendas sejam processadas *on line*, a fim de que os resultados destas sejam avaliados pelo staff gerencial das lojas. Assim, é necessário que o mecanismo de transmissão de vendas seja de alta disponibilidade. A disponibilidade no SAL é provida pelo mecanismo de balanceamento de carga do *Hermes*, que utiliza uma lista de endereços possíveis para transmissão, em vez de utilizar

apenas o endereço mais ocioso. Esta característica permite aos módulos do SAL, que usam o *Hermes* como transmissor, terem sempre rotas alternativas de transmissão das vendas, aumentando a disponibilidade do serviço de processamento de vendas.

Além do mecanismo de balanceamento de carga prover disponibilidade, o *Hermes* trabalha com o conceito de múltiplos servidores *socket* TCP ligados a múltiplas portas nos seus receptores. Esta característica, analisada no Capítulo 5, aumenta a disponibilidade dos receptores, permitindo que portas alternativas de transmissão em um mesmo receptor sejam utilizadas pelos transmissores das vendas.

6.3.3 Transmissão de Vendas para o Módulo de Integração

As vendas, após serem processadas, são enviadas pelo Processador de Vendas ao Módulo de Integração. Este envio é realizado pelo próprio *handler* que processa as vendas, utilizando o *Hermes* como um elemento transmissor que repassa as vendas processadas para o Módulo de Integração.

O processamento de uma venda no Processador de Vendas compreende três passos, um dos quais é a transmissão para o Módulo de Integração. É importante que o mecanismo de integração das vendas seja completamente independente do processamento da mesma dentro do SAL, porque a integração com sistemas corporativos é um processo que depende não só do SAL, mas também do sistema envolvido. Se ocorrer alguma situação de exceção no processo de integração, esta não pode interromper ou atrasar o processamento das vendas dentro do SAL.

Desta forma, o Módulo de Integração é totalmente isolado do Processador de Vendas, e o mecanismo de transmissão das vendas do segundo para o primeiro deve garantir que eventuais problemas no processo de integração como um todo, incluindo a transmissão para o Módulo de Integração, não interfira no processamento das vendas.

Se uma venda estiver processada, ela tem que ser obrigatoriamente entregue para integração, mesmo que haja desligamentos de qualquer sub-sistema envolvido no processo. Com base nestas características, o *Hermes* é utilizado como um transmissor assíncrono no Processador de Vendas e como um receptor assíncrono no Módulo de Integração.

Nas primeiras versões, o SAL utilizava filas únicas de saída e de entrada no Processador de Vendas e no Módulo de Integração. Porém, esta abordagem causou um gargalo no processo de integração, pois todas as vendas de todas as lojas convergiam para uma fila única, e o nível de simultaneidade de processamento na integração era nulo. O Módulo de Integração ficava ocioso e a fila única sempre populada, por conta de um gargalo lógico, causado pela configuração do mecanismo de transmissão das vendas.

A solução foi criar um esquema de múltiplas filas de saída e de entrada no Processador de Vendas e no Módulo de Integração. Como a mensagem associada a uma venda é sempre

do mesmo tipo, foi necessário associar tópico com loja, para que houvesse um par fila de saída - fila de entrada por loja. Desta forma, o enfileiramento das vendas se dá por loja, e o nível de simultaneidade de processamento na integração aumenta na escala da quantidade de lojas. Com esta abordagem, o problema do gargalo foi resolvido.

A Figura 6.3 mostra a configuração de filas e as modalidades de uso do *Hermes* na transmissão de vendas do Processador de Vendas para o Módulo de Integração. A característica do *Hermes* de criar filas dinamicamente torna o processo de configuração das filas por loja bem mais simples.

Cada vez que uma loja nova é inserida no contexto de uso do SAL, não é necessário alterar nenhuma configuração do sistema. Basta que as vendas da nova loja sejam enviadas ao Processador de Vendas que uma fila associada à nova loja é criada. O *handler* integrador de vendas realiza o processo de integração das vendas com o sistema corporativo.

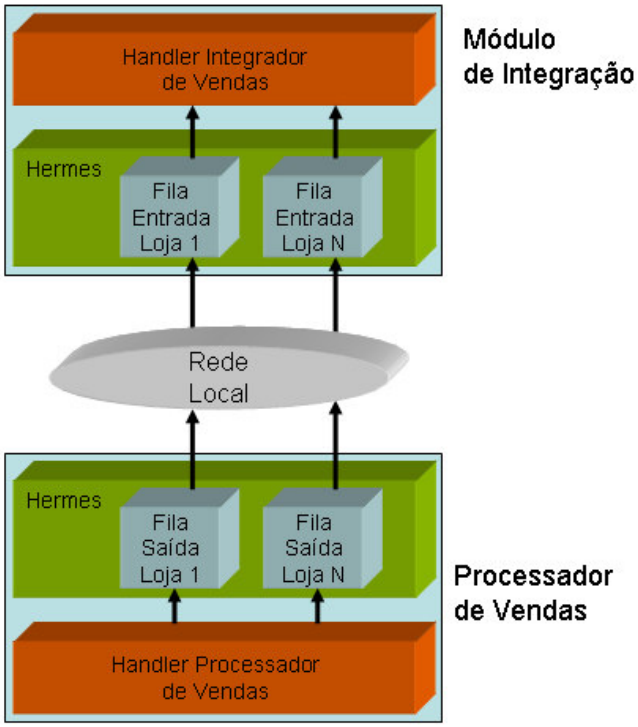

Figura 6.3 - *Hermes* na Transmissão de Vendas para o Módulo de Integração

6.3.4 Transmissão de Dados de Controle na Distribuição de Preços

Cada PDV tem uma réplica da lista de preços existente no Sistema de Retaguarda. Quando os preços são atualizados neste sistema, as réplicas nos PDVs têm que ser obrigatoriamente atualizadas. O mecanismo de atualização destas réplicas é denominado processo de distribuição de preços, que envolve os seguintes passos:

- Geração, pelo Sistema de Retaguarda, de um arquivo com os preços atualizados. Este arquivo é denominado lote;

- Sinalização, pelo Sistema de Retaguarda, aos Servidores de Loja, de que uma atualização de preços foi gerada e está disponível;

- *Download*, pelos Servidores de Loja, do lote;

- Sinalização, pelos Servidores de Loja, aos PDVs, de que uma atualização de preços foi gerada e está disponível; e

- *Download* e processamento do lote pelos PDVs, que lêem a lista dos preços atualizados do lote e remarcam os preços na sua lista local.

Neste cenário, os Servidores de Loja atuam como concentradores locais do lote. Em vez de todos os PDVs de todas as lojas realizarem o *download* do lote diretamente do Sistema de Retaguarda via *link* ou Internet, apenas um Servidor de Loja o faz, e os PDVs realizam o *download* do lote a partir dos Servidores de Loja utilizando uma rede local.

Os processos de sinalização envolvem a transmissão de dados de controle sobre o lote gerado. Número do lote atual, versão do arquivo a ser baixado, usuário e senha da conexão FTP (*File Transfer Protocol*) a ser aberta para baixar o arquivo são algumas das informações de controle enviadas do Sistema de Retaguarda para os Servidores de Loja e deste para os PDVs. A Figura 6.4 mostra as relações entre os sub-sistemas envolvidos no processo de distribuição de preços no SAL.

Se um lote estiver gerado, ele tem que ser obrigatoriamente baixado para atualização nos PDVs, mesmo que haja desligamentos de qualquer sub-sistema envolvido no processo, indisponibilidade temporária dos servidores FTP ou qualquer outra contingência. Uma vez que o processo de sinalização é iniciado, tem que haver a garantia de entrega dos dados de controle, tanto do Sistema de Retaguarda para os Servidores de Loja quanto destes para os PDVs. Com base nestas premissas, o *Hermes* deve ser usado como transmissor assíncrono no Sistema de Retaguarda e no Servidor de Loja, e como receptor assíncrono no Servidor de Loja e nos PDVs.

O processo de sinalização descrito determina que uma mensagem de controle seja enviada de um ponto para N pontos, tanto do Sistema de Retaguarda para os Servidores de Loja quanto destes para os PDVs. É possível usar o *Hermes* com modalidade de endereçamento *publish-subscribe* neste contexto. Uma mesma mensagem, contendo os dados de controle do lote, deve ser enviada a vários destinatários. Cada destinatário interessado em receber a referida mensagem deve se registrar como um *subscriber* do tipo da mensagem.

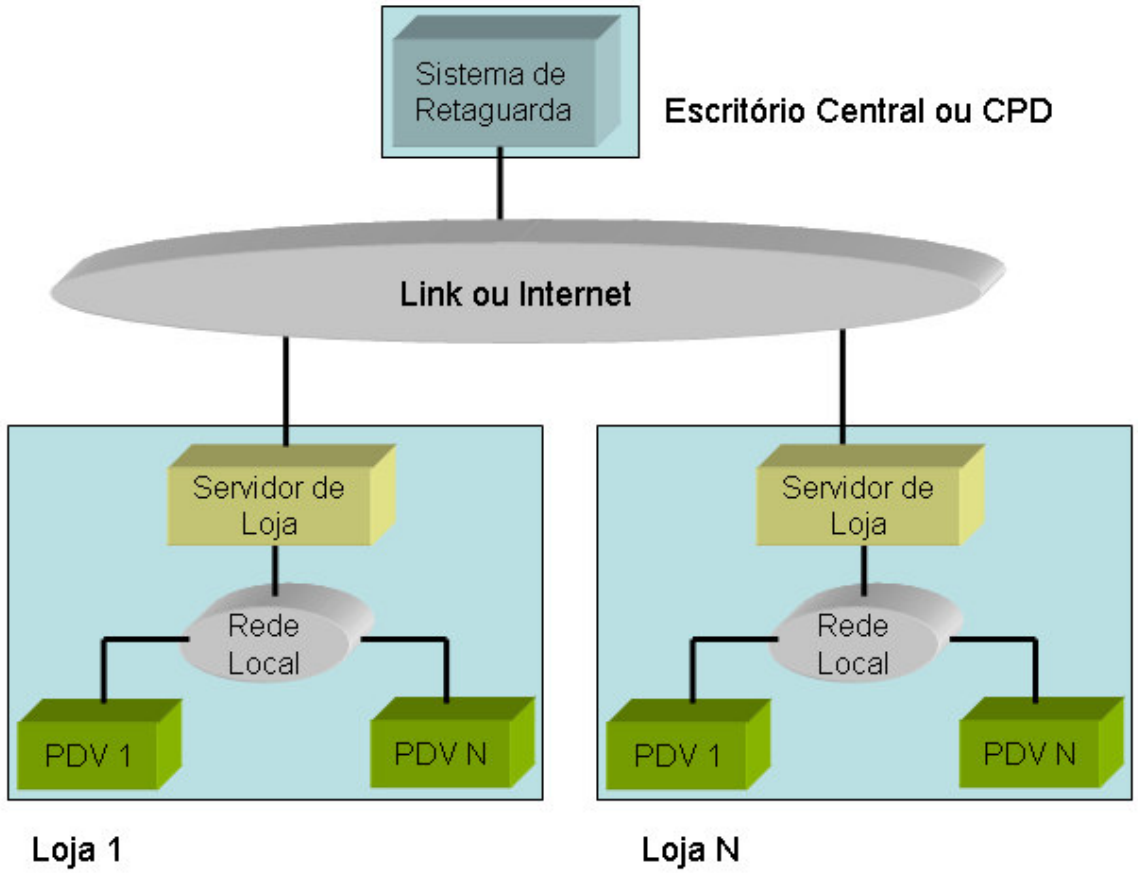

Figura 6.4 – Arquitetura dos Sub Sistemas envolvidos na Distribuição de Preços

O Sistema de Retaguarda usa o *Hermes* como *publisher* da mensagem de lote. Os Servidores de Loja se registram como *subscribers* desta mensagem em um Serviço de Nomes, localizado fisicamente no escritório central. Os Servidores de Loja usam o *Hermes* como *publisher* da mensagem de lote. Os PDVs se registram como *subscribers* desta mensagem nos Serviços de Nomes, localizados fisicamente nas lojas. A Figura 6.5 mostra o *Hermes* no contexto da transmissão de dados de controle na distribuição de preços.

O Sistema de Retaguarda gera o arquivo de lote e sinaliza as "N" lojas, inserindo a mensagem em "N" filas de saída, cada uma das quais correspondendo a uma loja. A modalidade de endereçamento é *publish-subscribe*. O *Hermes* identifica os endereços das lojas solicitando ao Serviço de Nomes do Sistema de Retaguarda todos os *subscribers* da mensagem a ser enviada.

Cada Servidor de Loja possui uma fila única de entrada. A mensagem é processada pelo *handler* de lote, que realiza o *download* do arquivo e sinaliza os "N" PDVs, inserindo a mensagem em "N" filas de saída. A modalidade de endereçamento é *publish-subscribe*. O *Hermes* identifica os endereços dos PDVs solicitando ao Serviço de Nomes da loja todos os *subscribers* da mensagem a ser enviada. Cada PDV possui uma única fila de entrada.

A mensagem é processada pelo *handler* de lote do PDV, que realiza o *download* do arquivo e atualiza a lista local de preços.

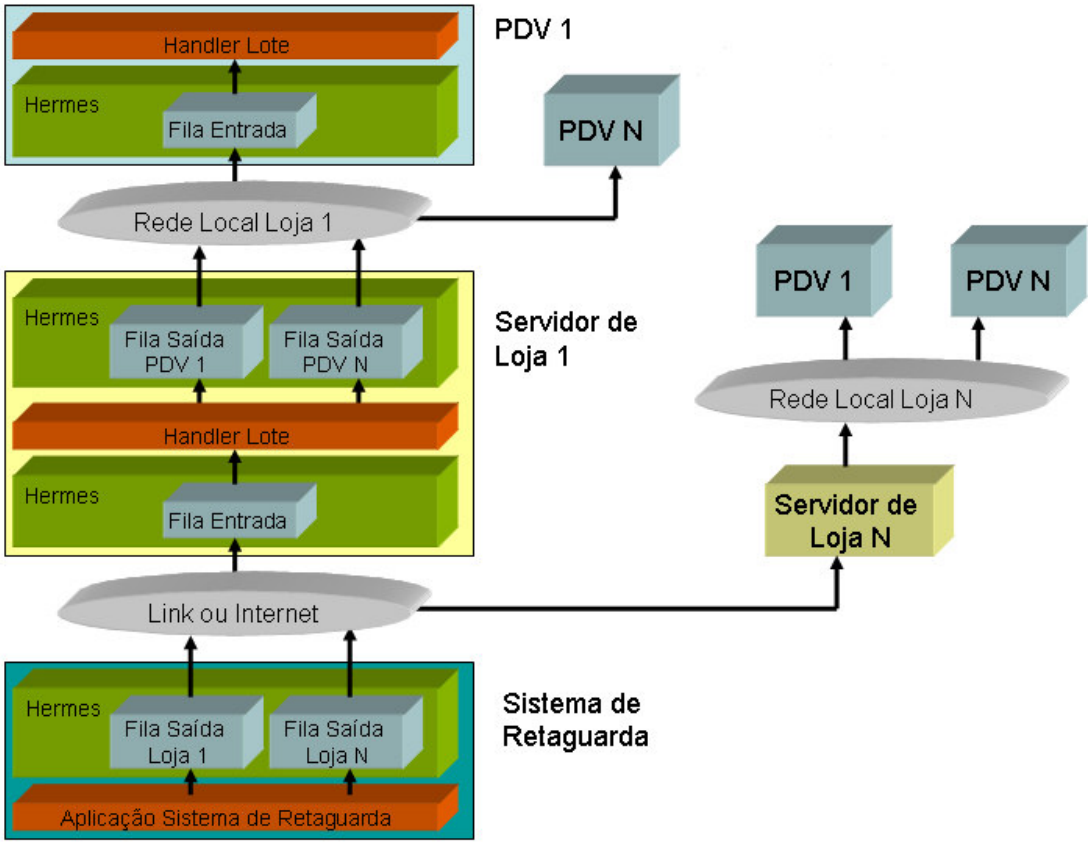

Figura 6.5 - *Hermes* na Transmissão de Dados de Controle na Distribuição de Preços

6.3.5 Consulta em Tempo Real de Estoque

A consulta em tempo real de estoque requer uma verificação no sistema de controle de estoque do sistema corporativo da disponibilidade do produto a ser vendido. Esta operação ocorre quando do ato da venda de um produto no PDV. Como a consulta é feita em tempo real, o processo de venda só se completa se a consulta for realizada com sucesso e tiver uma resposta positiva sobre a posição do estoque do produto a ser vendido. Este requisito torna o processo de consulta de estoque extremamente crítico, pois o serviço de consulta tem que ser em tempo real, rápido e altamente disponível, sob pena de interromper o processo de venda.

A Figura 6.6 mostra a arquitetura da função de consulta em tempo real de estoque e o papel do *Hermes* neste contexto. O PDV se comunica com o Módulo de Consulta de Estoque, que possui um tratador para encaminhar a solicitação de consulta para o sistema corporativo. Este responde à solicitação e a resposta é então retornada para o PDV. O *Hermes* neste contexto é usado como transmissor síncrono no PDV e como receptor síncrono no Sistema de Retaguarda.

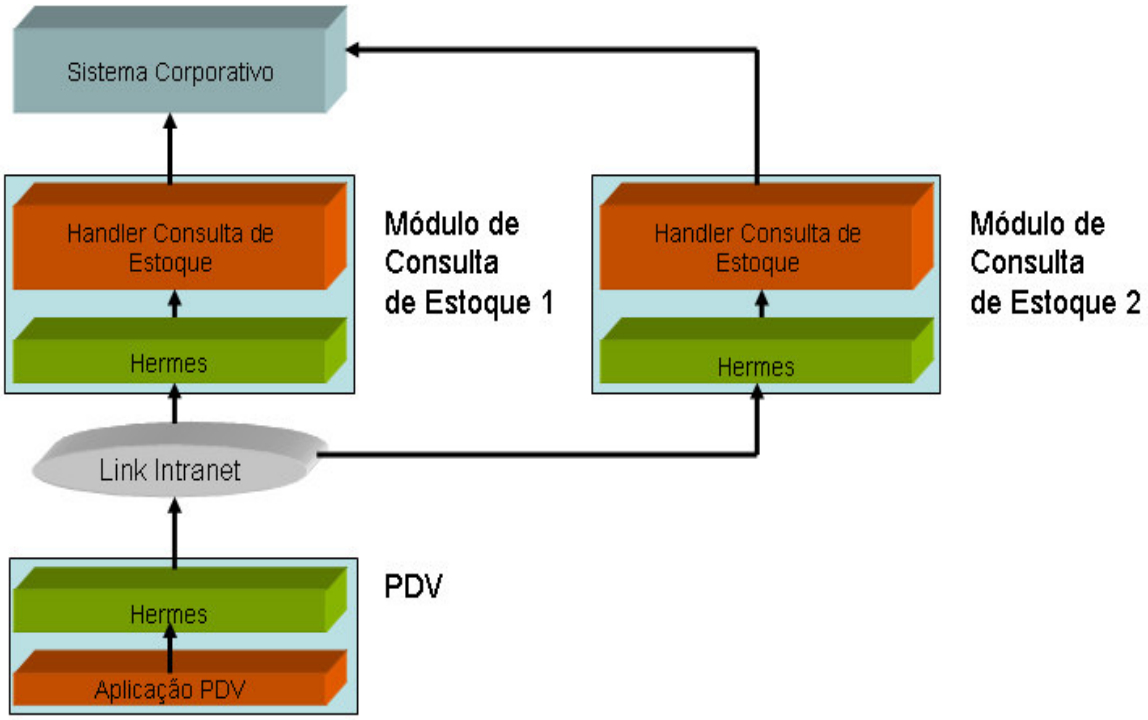

Figura 6.6 – *Hermes* na Consulta em Tempo Real de Estoque

Como a consulta exige alta disponibilidade, houve a necessidade de utilizar algumas características do *Hermes* que permitem aumento de disponibilidade e rotas alternativas. O Módulo de Consulta de Estoque é executado em duas máquinas diferentes, com endereços diferentes. A modalidade de endereçamento usado nos transmissores *Hermes* dos PDVs é o contingenciamento de rotas.

As lojas foram divididas em dois grupos, com quantidades aproximadamente iguais de solicitações diárias de consultas. O primeiro grupo tem como servidor preferencial o número 1 e servidor alternativo o número 2. O segundo grupo tem como servidor preferencial o número 2 e servidor alternativo o número 1.

Cada receptor *Hermes* do Sistema de Retaguarda foi configurado para se ligar a cinco portas lógicas diferentes. O grupo de lojas com servidor preferencial número 1, com 25 lojas, foi dividido em 05 sub-grupos de 05 lojas cada grupo. Cada sub-grupo de 05 lojas tem uma lista preferencial de acesso a portas lógicas diferente, onde as portas preferenciais em uma lista são as alternativas em outras. Para o grupo de lojas com servidor preferencial número 2 também foi usada a mesma sistemática.

A Tabela 6.2 mostra a forma de distribuição dos grupos e sub-grupos de lojas em função dos servidores 1 e 2 e das portas lógicas disponíveis em cada servidor. Desta forma, foi possível distribuir eqüitativamente a carga de solicitação de consultas entre servidores e portas e criar rotas e portas alternativas para a realização das consultas. As rotas alternativas seguem o mesmo padrão para a lista de portas.

Tabela 6.2 – Distribuição de Servidores e Portas Lógicas entre Grupos de Lojas

Servidor Número 1 – Rota Preferencial	
Lojas	**Lista de Portas Preferenciais**
Lojas 1-5	4000, 4001, 4002, 4003, 4004
Lojas 6-10	4001, 4002, 4003, 4004, 4000
Lojas 11-15	4002, 4003, 4004, 4000, 4001
Lojas 16-20	4003, 4004, 4000, 4001, 4002
Lojas 20-25	4004, 4000, 4001, 4002, 4003
Servidor Número 2 – Rota Preferencial	
Lojas	**Lista de Portas Preferenciais**
Lojas 26-30	4000, 4001, 4002, 4003, 4004
Lojas 31-35	4001, 4002, 4003, 4004, 4000
Lojas 36-40	4002, 4003, 4004, 4000, 4001
Lojas 41-45	4003, 4004, 4000, 4001, 4002
Lojas 46-50	4004, 4000, 4001, 4002, 4003
Servidor Número 2 – Rota Alternativa	
Lojas	**Lista de Portas Preferenciais**
Lojas 1-5	4000, 4001, 4002, 4003, 4004
Lojas 6-10	4001, 4002, 4003, 4004, 4000
Lojas 11-15	4002, 4003, 4004, 4000, 4001
Lojas 16-20	4003, 4004, 4000, 4001, 4002
Lojas 20-25	4004, 4000, 4001, 4002, 4003
Servidor Número 1 – Rota Alternativa	
Lojas	**Lista de Portas Preferenciais**
Lojas 26-30	4000, 4001, 4002, 4003, 4004
Lojas 31-35	4001, 4002, 4003, 4004, 4000
Lojas 36-40	4002, 4003, 4004, 4000, 4001
Lojas 41-45	4003, 4004, 4000, 4001, 4002
Lojas 46-50	4004, 4000, 4001, 4002, 4003

6.4 Considerações Finais

Este estudo de caso contemplou a análise da utilização do *Hermes* em um contexto de aplicação corporativa, utilizada em diversos ambientes com características funcionais distintas e com requisitos de carga e de performance distintos.

De uma maneira geral, as características do *Hermes* se mostraram aderentes aos requisitos de troca de mensagens exigidos no contexto de utilização observado. Em cada cenário de aplicação do *Hermes*, algumas características deste contribuíram para que os requisitos funcionais relativos à transmissão de dados fossem atendidos.

No cenário de transmissão de vendas para o Processador de Vendas, as seguintes características do *Hermes* foram diferenciais:

- A possibilidade de combinação dos diversos tipos de transmissão possíveis em um sistema de troca de mensagens. O uso de uma destas combinações atendeu a um requisito da aplicação. Foi utilizado o mecanismo de transmissão assíncrona e processamento síncrono para transmissão das vendas dos PDVs para o Servidor de Loja, deste para o Processador de Vendas e dos PDVs para o Processador de Vendas. Esta característica é uma otimização do modelo de transmissão proposto pela maioria dos MOMs;

- A possibilidade de ligação de um receptor *Hermes* com múltiplas portas, que permitiu, em alguns casos onde a carga de processamento de vendas é alta, uma melhora significativa na disponibilidade do receptor. A utilização de múltiplas portas reduziu, nestas situações, o índice de conexões recusadas e o tempo médio de resposta na transmissão de uma venda. Esta característica é uma otimização do modelo de implementação de um servidor *socket* TCP; e

- O mecanismo de balanceamento de carga, que possibilitou o processamento escalável e mais disponível das vendas. Este mecanismo prevê a determinação de um servidor preferencial mais uma lista de servidores alternativos, permitindo a combinação de balanceamento de carga e aumento de disponibillidade.

No cenário de transmissão de vendas para o Módulo de Integração, foi diferencial o fato do *Hermes* permitir a criação dinâmica de filas. O problema do gargalo e da ociosidade foi resolvido com a utilização de uma fila por loja. Como no *Hermes* não existe o ônus de configurar e gerenciar filas, pois estas são criadas dinamicamente e mantidas pelo próprio *Hermes*, não houve custo na resolução do problema do gargalo.

No cenário de transmissão de dados de controle para distribuição de preços, foi diferencial o mecanismo *publish-subscribe*, que permitiu a replicação das mensagens de controle de uma forma direta, simplificando de forma significativa a implementação da rotina de envio na aplicação. Além desta simplificação, o mecanismo *publish-subscribe*, junto como Serviço de Nomes, permite que novas lojas e PDVs sejam adicionados ao contexto sem necessidade de configurações adicionais. Para que PDVs e Servidores de Lojas recebam mensagens de controle referentes à distribuição de preços, basta que eles se registrem nos seus respectivos Servidores de Nomes, o que já naturalmente feito na inicialização dos mesmos.

No cenário de consultas em tempo real de estoque, as seguintes características do *Hermes* foram diferenciais:

- O mecanismo de contingenciamento de rotas, que permitiu o aumento da disponibilidade de um serviço crítico. O uso de rotas alternativas de acesso a uma função que, por força do requisito específico da aplicação, pode interromper ou até mesmo inviabilizar um processo de venda, diminuiu significativamente o índice de consultas mal sucedidas, e conseqüentemente o índice de vendas não realizadas; e

- A possibilidade de ligação de um receptor *Hermes* com múltiplas portas, que permitiu uma melhora significativa na disponibilidade dos serviços de consulta. A combinação do esquema de contingenciamento de rotas com a utilização de múltiplas portas lógicas por instância de receptor possibilitou a redução do índice de consultas mal sucedidas para um valor próximo de zero.

Um diferencial do *Hermes* observado em todos os cenários descritos é a facilidade de utilização do mesmo. Em especial, a redução de complexidade de código para implementar rotinas de processamento por processo em espera é significativa, pois se resume a implementação de duas funções em uma classe que representa um *handler*.

A possibilidade de configurar as associações entre *handlers*, tipos de mensagens e tópicos é outra facilidade na utilização do *Hermes*, pois não é necessário alterar código para adicionar, remover ou alterar estas associações. Durante a realização desse estudo de caso, foram verificadas algumas questões que serão consideradas o ponto de partida para realizar trabalhos futuros a fim de melhorar o *Hermes*:

- Há uma necessidade de mecanismos de monitoramento e de controle para os Serviços de Nomes em operação, quando a quantidade destes servidores é grande;

- Dentro de um contexto onde a quantidade de lojas é grande (maior que 100), é importante a presença de uma ferramenta de monitoramento dos receptores mais sofisticada, com funções adicionais às que hoje estão disponíveis; e

- Alguns controles e atributos referentes a mensagens poderiam estar disponíveis em arquivos de configuração, em vez de serem alterados diretamente no código da aplicação.

Capítulo 7

Conclusão

Este capítulo apresenta as principais contribuições desta dissertação e algumas direções de trabalhos futuros.

7.1 Contribuições

Esta dissertação apresentou um *middleware* orientado a mensagem chamado *Hermes*. As principais contribuições deste trabalho estão relacionadas à engenharia de software de *middleware* e ao desenvolvimento e implementação de mecanismos específicos de MOMs. As contribuições relacionadas a estes dois pontos são apresentadas a seguir:

Engenharia de Software de *Middleware*
A apresentação detalhada das etapas de desenvolvimento de um *middleware*, incluindo o uso de padrões de implementação, tais como: uso e controle de *threads*, controle de concorrência por objetos monitores e outros. Adicionalmente, os conceitos de componentização e de modularização foram amplamente utilizados no *Hermes*. Foram apresentados alguns modelos e padrões reusáveis. Alguns deles, de uso geral, são independentes dos conceitos relacionados aos MOMs. Dentre estes, podemos destacar o *pool* de *threads*, a fila e o sistema de monitoramento. Outros são de uso exclusivo em projetos de MOMs, mas possuem papéis e atribuições claramente definidos. Dentre estes, podemos destacar o mapeador de *handlers*, o *listener* e o transmissor síncrono. Estes dois últimos podem ser utilizados em contextos que requerem um processo de comunicação cliente-servidor.

Balanceamento de Carga e Contingenciamento de Rotas no Serviço de Nomes
No *Hermes*, o Serviço de Nomes faz o controle destas duas características, permitindo que as configurações de endereços para balanceamento de carga e para contingenciamento de rotas seja realizada de forma centralizada, diminuindo o ônus de se gerenciar a configuração das diversas aplicações que usam o *Hermes* como transmissor. Além desta facilidade, o balanceamento de carga controlado pelo Servidor de Nomes do *Hermes* permite que sejam retornadas para o transmissor rotas alternativas, além do endereço a ser acessado no momento. Desta forma, aumenta-se potencialmente a disponibilidade dos serviços registrados no Servidor de Nome sob esta modalidade de endereçamento.

Flexibilidade nos Tipos de Transmissão
No *Hermes*, é possível usar qualquer combinação dos tipos de transmissão apresentados no Capítulo 2. Normalmente, os MOMs disponibilizam transmissões e processamentos assíncronos, com filas de saída e de entrada. O *Hermes* possibilita quatro combinações: transmissão síncrona - processamento síncrono, transmissão assíncrona – processamento síncrono, transmissão síncrona – processamento assíncrono e transmissão assíncrona – processamento assíncrono. Com esta flexibilidade, aplicações não necessitam implementar controles para ter tais opções, o próprio *Hermes* se encarrega desta implementação, só sendo necessário à aplicação escolher qual o tipo de transmissão e o tipo de processamento.

Tipos de Mensagens Associados a Tópicos
O *Hermes* permite que o próprio tipo da mensagem seja associado a um tópico, que representa uma fila lógica no contexto de um MOM, caso não seja informado o tópico

associado à mensagem. Esta facilidade foi incorporada porque diversas aplicações usam como critério de formação dos seus conjuntos de tópicos os tipos das mensagens usadas. Esta inovação beneficia especialmente aplicações que usam o *Hermes* como um solicitador de processamento ou de serviços remotos.

Criação e Gerência Dinâmica de Filas
O *Hermes* gerencia dinamicamente as filas lógicas, criando filas novas quando necessário e destruindo filas lógicas ociosas e vazias. Esta inovação poupa as aplicações do trabalho de criação de um esquema de filas previamente definido e da posterior manutenção deste esquema.

Múltiplas Portas por Instância de Receptor
O *Hermes* permite que uma instância receptora tenha diversas portas lógicas de comunicação disponíveis em vez de uma só. O aumento de portas lógicas ligadas a servidores *socket* TCP aumenta consideravelmente a disponibilidade de conexão, permitindo a redução dos índices de conexões recusadas.

Simplificação no Processamento de Mensagens por Processo em Espera
O *Hermes* permite que os controles inerentes ao processamento de mensagens por processo em espera, normalmente implementados pela aplicação, sejam retirados do conjunto de responsabilidades dos programadores. Esta facilidade faz com que tarefas como criação e controle de *threads* fiquem a cargo do *Hermes*, cabendo à aplicação apenas a implementação de duas funções.

Facilidade de administração de Filas
A complexidade de administração foi reduzida com a característica de criação e gerência dinâmica de filas, tirando de administradores e de operadores do sistema as atribuições de criação de uma estrutura prévia de filas e tópicos associados.

O *Hermes* implementa a maioria das características dos MOMs, apresentadas no Capítulo 2. As formas de implementação destas características no MOM *Hemes* estão descritas abaixo.

Enfileiramento e sincronismo das mensagens
O *Hermes* disponibiliza sincronismo e enfileiramento de mensagens através da implementação das filas, utilizando um componente que representa filas lógicas e alguns elementos que gerenciam o acesso a estas filas. O sincronismo de entrega é coordenado tanto pelo transmissor, nas filas de saída, quanto pelo receptor, nas filas de entrada.

Assincronismo de transmissão e de processamento das mensagens
Filas coordenadas pelos transmissores e receptores permitem que as funcionalidades de assincronismo de processamento e de transmissão sejam apreciadas pelos usuários do *Hermes*. O assincronismo de transmissão se dá tanto no transmissor, com a possibilidade de assincronismo de processamento no lado receptor. Além da modalidade usual de transmissão e de processamento de mensagens, o *Hermes* suporta combinações síncronas e assíncronas de transmissão e de processamento de mensagens.

Persistência das mensagens enfileiradas
A implementação de persistência das mensagens no *Hermes* usa recursos próprios, sendo utilizado o mecanismo de persistência de objetos das lingugagens de programação utilizadas, embora a arquitetura da implementação permita que novos mecanismos de persistência sejam facilmente incorporados como alternativas adicionais.

Tolerância a falhas no cliente
O enfileiramento de mensagens realizado no transmissor, com persistência das mensagens a serem transmitidas, provê tolerância a falhas no cliente. Se uma mensagem for inserida na fila de saída, o *Hermes* garante a entrega aos destinatários, mesmo que haja desligamentos da aplicação receptora.

Filas e tópicos
A política de tópicos no *Hermes* segue o padrão JMS. Cada tópico é associado a uma fila de mensagens e na hora do envio de uma mensagem a um ou mais destinatários, esta é colocada na fila associada ao tópico. Além deste padrão, o *Hermes* permite associação automática entre um tópico e um tipo de mensagem, representado pelo nome completo da classe que representa uma dada mensagem. O *Hermes* ainda provê gerência dinâmica das filas, criando novas filas lógicas quando necessário e removendo filas lógicas ociosas de tempos em tempos.

Formas de tratamento das mensagens
O *Hermes* realiza tratamento de mensagens por notificação de eventos. Quando uma mensagem chega ao receptor, o *handler* a ela associado é identificado e a mensagem é entregue a este último para processamento. O *handler* deve informar ao *Hermes* o resultado do processamento que, em caso de uma transmissão síncrona e de um processamento síncrono, deve ser retornado à aplicação transmissora. Em caso de transmissões e de processamentos assíncronos, o resultado do processamento é avaliado para que o *Hermes* decida se remove a mensagem da fila ou se a mantém enfileirada até que um novo processamento ou uma nova transmissão aconteça. No caso de processamento assíncrono, uma *thread* bloqueante é associada a cada fila lógica e ela é responsável por invocar o *handler* de processamento e de controlar o processo de manutenção ou de remoção das mensagens na fila, evitando que o programador das aplicações que utilizam o *Hermes* realize implementações referentes ao controle do processo associado à fila e à lógica de remoção ou manutenção das mensagens na fila.

Tolerância a falhas na rede
O enfileiramento de mensagens realizado no transmissor, com persistência das mensagens a serem transmitidas, provê tolerância a falhas na rede. Se uma mensagem for inserida na fila de saída, o *Hermes* garante a entrega aos destinatários, mesmo que haja problemas de transmissão causados por questões referentes à rede.

Tolerância a falhas no servidor
O enfileiramento de mensagens realizado no transmissor, com persistência das mensagens a serem transmitidas, provê tolerância a falhas na servidor. Se uma mensagem for inserida na fila de saída, o *Hermes* garante a entrega aos destinatários mesmo que haja

problemas de transmissão, causados por questões de servidor inativo ou sem comunicação. Por sua vez, o enfileiramento de mensagens realizado no receptor, com persistência das mensagens a serem transmitidas, também provê tolerância a falhas no servidor. Se uma mensagem for inserida na fila de entrada, o *Hermes* garante o processamento, mesmo que haja desligamentos da aplicação receptora.

Balanceamento de carga
O *Hermes* realiza balanceamento de carga baseado em uma lógica simples de alternância de endereçamento de serviços, disponível no Serviço de Nomes. Serviços e endereços associados à função de balanceamento de carga devem ser cadastrados em uma lista, carregada na inicialização do Serviço de Nomes.

Transparência de localização
O Serviço de Nomes do *Hermes* garante a transparência de localização de serviços para transmissores *Hermes,* que utilizam a função de envio de mensagens para um determinado nome. O endereço ou a lista de endereços, associados ao serviço identificado por um nome, é retornada do Serviço de Nomes para os transmissores *Hermes*.

Transparência de migração
O *Hermes* implementa transparência de migração através do modo de endereçamento por contingenciamento de rotas, disponível no Serviço de Nomes. Quando um determinado endereço associado a um serviço não está disponível, podem existir endereços alternativos que disponibilizem o mesmo serviço. A lista de endereços associada a um serviço deve ser carregada na inicialização do Serviço de Nomes.

Transparência de replicação
O *Hermes* não implementa nenhum mecanismo de transparência de replicação.

Segurança – autenticação e criptografia
O *Hermes* não implementa nenhum mecanismo de autenticação e segurança.

Heterogeneidade
Do ponto de vista de plataforma / sistema operacional, o *Hermes* suporta 100% de heterogeneidade, desde que o sistema operacional possua uma JVM. Dentre os que possuem JVM, podemos destacar: família *Windows* (95, 98, 2000, *Millenium*, XP, *Server* 2000, *Server* 2003), *Linux*, família *Unix* (HP UX, SCO, *Solaris*, etc.), OS2 e *Mac*. Do ponto de vista de plataforma / linguagem de programação, o *Hermes* pode interagir com aplicações que suportam protocolo SOAP. A API do *Hermes* está disponível em JAVA e em C# .NET (com algumas restrições).

7.2 Trabalhos Futuros

As propostas de trabalhos futuros estão relacionadas principalmente à avaliação de desempenho de *middleware* [2][45], à implementação de mecanismos de monitoramento e à adoção do *Hermes* em outros domínios de aplicações. Estas propostas são apresentadas a seguir:

Avaliação de desempenho: avaliação comparativa do desempenho do *Hermes* com outros MOMs [29][36], como o *MQ Series* e o JORAM. Os cenários poderão ser os mesmos do estudo de caso apresentado neste trabalho, pois já existem parâmetros relacionados à qualidade de serviços definidos para aqueles. O *Hermes* possui pontos internos para medições de alguns parâmetros necessários a um estudo de avaliação de desempenho, facilitando a coleta de dados.

Aplicações em outros domínios: realização de um estudo de caso do *Hermes* em contexto diferente do apresentado neste trabalho. Como MOMs são componentes de uso geral, o *Hermes* deve ser incorporado à outra aplicação de domínio diferente das soluções de automação comercial. O que se espera como resultado deste estudo de caso é a repetição das conclusões sobre o uso do *Hermes*: aderência a requisitos funcionais e não funcionais relacionados a processos de troca de mensagens, flexibilidade e facilidade de uso.

Monitoramento e controle para os serviços de nomes: em contextos onde a quantidade de servidores de nomes é grande, faz-se necessária a utilização de uma ferramenta de monitoramento remoto dos Serviços de Nomes. Esta ferramenta deve contemplar ativação / desativação de um Serviço de Nomes em um determinado endereço, consulta às informações manipuladas no serviço de nomes, ativação / desativação de um tópico e suas associações.

Melhorias no Sistema de Monitoramento: em contextos com uma grande quantidade de transmissores e de receptores *Hermes*, algumas melhorias no sistema de monitoramento são necessárias: monitoramento via web, limpeza remota de filas, consulta / manutenção remota ao repositório de mensagens mortas (expiradas ou não processadas por conta da ocorrência de uma exceção não prevista) e ativação e desativação de receptores.

Melhorias no Modelo de Mensagens: o modelo de mensagens poderá contemplar algumas configurações adicionais: tempo e critério de expiração da mensagem, definidas na implementação da classe que representa a mensagem, serão configurações definidas em arquivos; associação entre mensagem e tópico, também definida na implementação da classe que representa a mensagem, será também uma configuração definida em arquivo.

Manutenção dos Parâmetros e das Configurações do Hermes: construção de interfaces gráficas web e de infra-estrutura para alterar e consultar, remotamente, os parâmetros de configuração e de funcionamento do *Hermes*. Em ambientes cuja

capilaridade do *Hermes* é grande, a manutenção remota, *on line* e centralizada de parâmetros é extremamente útil. Para este propósito, serão utilizadas as funcionalidades de comunicação do próprio *Hermes*.

Autenticação e Criptografia: implementações adicionais, relativas a funções de checagem de autenticidade de origem e de criptografia poderão ser incorporadas ao autenticador de mensagens como *plug-ins* independentes. A própria linguagem de programação Java, possui classes que realizam tais funções. Estas classes podem ser utilizadas dentro do autenticador para que uma mensagem possa ter sua origem autenticada através de um sistema de verificação de conteúdo e geração de chaves públicas e privadas. Um esquema de criptografia e decriptografia da mensagem, também baseado em chaves públicas e privadas de 128 bits, está disponível e pronto para uso na linguagem Java.

Mecanismos Adicionais de Persistência das Filas: Poderá ser incorporado um mecanismo alternativo de persistência das filas de saída e de entrada que utiliza um banco de dados para armazenar mensagens. Em Java e C# (.NET) é possível criar o mecanismo independente do banco de dados utilizado, pois estas linguagens trabalham com padrões de acesso que suportam SQL padrão, e criam uma abstração entre a aplicação e o banco utilizado. Esta funcionalidade permitirá que, no futuro, instâncias diferentes de transmissores e receptores *Hermes* compartilhem fisicamente a mesma fila de mensagens.

Prioridade de Mensagens: Poderá ser incorporado um mecanismo para definição de prioridade de processamento para as mensagens do *Hermes* que são inseridas nas filas de entrada no lado receptor.

Bibliografia

[1] Aaron E Walsh. **UDDI, SOAP, and WSDL: The Web Services Specification Reference Book**. UDDI Org, 2002.

[2] Abdul-Fatah Istabrak, Majumdar Shikharesh. **Performance of CORBA-Based Client-Server Architectures**. IEEE Trans. Parallel and Distributed Systems, Vol.13, No.2, pg.111-126, February 2002.

[3] Alan Dennis, Barbara Haley Wixom, David Tegarden. **Systems Analysis and Design: An Object-Oriented Approach with UML**. JohnWiley & Sons, 2001.

[4] Andreas Vogel, Madhavan Rangarao. **Programming with Enterprise JavaBeans, JTS, and OTS: Building Distributed Transactions with Java and C++**. John Wiley & Sons, 1998.

[5] Andrew T. Campbell, Geoff Coulson, Michael E. Kounavis. **Managing Complexity: Middleware Explained**. IT Professional, IEEE Computer Society, Vol 1(5), pp. 22-28, Outubro, 1999.

[6] Bobby Woolf. **Learn how the new API will help you write more reusable JMS clients:** http://www-106.ibm.com/developerworks/Java/library/j-jms11/. IBM Publishng, 2003.

[7] Cay S. Horstmann, Gary Cornell. **Core Java 1.1 Volume I – Fundamentals**. The Sunsoft Press, 1997.

[8] David Hunter, Kurt Cagle, Chris Dix, Roger Kovack, Jonathan Pinnock, Jeff Rafter. **Beginning XML, Second Edition**. Wrox, 2001.

[9] Douglas Schmidt. **Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects**. John Wiley & Sons, 2001.

[10] Fiorano Inc.: **Fiorano MQ Documentation:** http://www.fiorano.com/devzone/doc_fmq.htm.

[11] Fiorano Inc.: **Fiorano MQ: Meeting the Needs of Technology and Business:** http://www.fiorano.com/whitepapers/whitepapers_fmq.pdf. Fiorano, 2000.

[12] Fiorano Inc.: **Fiorano Study Cases:** http://www.fiorano.com/solutions/solutions.htm. Fiorano, 2000.

[13] Fiorano Inc.: **Highly Scalable Java Messaging - Understanding FioranoMQ's Pluggable, Scalable Connection Management (SCM) Architecture:** http://www.fiorano.com/whitepapers/fioranomq_scalability.pdf. Fiorano, 2000.

[14] George T. Heineman, William T. Councill. **Component-Based Software Engineering: Putting the Pieces Together**. Addison-Wesley, 2001.

[15] Grady Booch, James Rumbaugh, Ivar Jacobson .**The Unified Modeling Language User Guide**. Addison-Wesley, 1998.

[16] Guruduth Banavar, Tushar D. Chandra, R.obert E. Strom, Daniel C. Sturman. **A Case for Message-Oriented Middleware**. Thirteenth International Symposium on Distributed System, LNCS 1693, pp. 1-18, 1999.

[17] Ian East. **Parallel Processing with Communicating Process Architecture**. UCL Press, 1995.

[18] IBM Inc.: **MQ Family:** http://www-306.ibm.com/software/integration/mqfamily/. IBM Corporation, 2003.

[19] Jeff Magee, Jeff Kramer**. Concurrency: State Models & Java Programs**. John Wiley & Sons, 1999.

[20] Jeff Prosise. **Programming Microsoft .NET**. Wintellect, 2002.
[21] John Siegel. **CORBA 3 Fundamentals and Programming, 2nd Edition**. John Wiley & Sons, 2000.
[22] John T. Bell, James Lambros, Stan Ng. **J2EE Open Source Toolkit: Building an Enterprise Platform with Open Source Tools (Java Open Source Library)**. John Wiley & Sons, 2003.
[23] Kenneth Calvert, Michael Donahoo. **TCP/IP Sockets in Java: Practical Guide for Programmers.** MK / Elsevier, 2001.
[24] Kyle Gabhart. **J2EE messaging solutions for your enterprise:** http://www-106.ibm.com/developerworks/Java/library/j-pj2ee5/. IBM Publishing, 2003.
[25] Leonard Gilman, Richard Schreiber**. Distributed Computing with IBM(r) MQSeries.** John Wiley & Sons, 1997.
[26] Mark Cade, Simon Roberts. **Sun Certified Enterpreise Architect for J2EE Technology**. Sun Microsystems Press, 2002.
[27] Mark Wotka. **Special Edition Using Java 2 Enterprise Edition (J2EE): With JSP, Servlets, EJB 2.0, JNDI, JMS, JDBC, CORBA, XML and RMI**. QUE, 2001.
[28] Merlin Hughes, Michael Shoffner, Derek Hamner. **Java Network Programming: A Complete Guide to Networking, Streams, and Distributed Computing**. Manning Publications, 1999.
[29] Michael Pang, Piyush Maheshwari. **Benchmarking Message-Oriented Middleware – TIB/RV vs. SonicMQ**. 2002
[30] Object Management Group**:** http://www.omg.org/.
[31] ObjectWeb Inc.: **JORAM Manuals:** http://joram.objectweb.org/doc/index.html. ObjectWeb Consortium, INRA, 2004.
[32] Paul Hyde. **Java Thread Programming**. Sams Publishing, 1999.
[33] Paul Perrone. **J2EE Developer's Handbook**. Sams Publishing, 2003.
[34] Philip A.Bernstein. **Middleware: A Model for Distributed System Services**. Communications of the ACM, Vol 39 (2), pp. 87-98, fevereiro, 1996.
[35] Philip Heller, Simon Roberts. **Complete Java 2 Certification Study Guide, 3rd Edition**. Sybex, 2003.
[36] Phong Tran, Paul Greenfield. **Behaviour and Performance of Message-Oriented Middleware Systems**. Proc. 2nd IEEE International Conf. on Distributed Computing Systems Workshops, 2002
[37] Roger S. Pressman. **Software Engineering, a Practitioner's Approach, 4th Edition**. McGraw Hill, 1997.
[38] Roland Barcia. **JMS Application Architectures:** http://www.theserverside.com/articles/article.jsp?l=JMSArchitecture. TheServerSide Publishing, 2003.
[39] Ron Ben-Natan, Ori Sasson. **IBM Websphere Application Server: The Complete Reference**. McGraw Hill, 2002.
[40] Scaleagent Inc.: **Design and Deploy A JORAM Message System:** http://www.scalagent.com/pages/en/services/training02-3.htm. Scaleagent, 2000.
[41] Scott Grant, Michael P. Kovacs, Meeraj Kunnumpurath, Silvano Maffeis, K. Scott Morrison, Gopalan Suresh Raj, Paul Giotta, James McGovern**. Professional JMS**. Wrox, 2001.
[42] Shaun Terry. **Enterprise JMS Programming**. John Wiley & Sons, 2002.

[43] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Stearns. **The J2EE Tutorial**. Sun Microsystems Press, 2002.

[44] Steve Vinoski. **Where is Middleware?** IEEE Internet Computing, Vol. 6(2), pp. 83-85, 2002.

[45] Steve Vinoski. **The Performance Presumption**. IEEE Internet Computing, April 2003, pp. 88-91.

[46] Sun Microsystems Inc.: **JMS Tutorial:** http://Java.sun.com/products/jms/tutorial/index.html. Sun Microsystems, 2002.

[47] Sun Microsystems Inc: **Java Message Service Specification:** http://Java.sun.com/products/jms/. Sun Microsystems, Março, 2002.

[48] Theparticle Inc.: **Java Data Structures:** http://www.theparticle.com/Javadata2.html. Theparticle, 1999.

[49] Vlada Matena, Mark Hapner. **Enterprise JavaBeans**. Sun Microsystems, 1998.

[50] Wolfgang Emmerich. **Software Engineering and Middleware: A Roadmap**. Second International Workshop on Software Engineering and Middleware, Limerick. Ireland, pp. 119-129, Junho, 2000.

[51] Worldwide Websphere Group Inc.: **Worldwide Websphere User Groups, Study Cases**: http://www.websphere.org/case_studies/case_studies.html. Worldwide Websphere Group, 2004.

[52] Wrox Author Team. **Professional Java Server Programming: with Servlets, JavaServer Pages (JSP), XML, Enterprise JavaBeans (EJB), JNDI, CORBA, Jini and Javaspaces**. Wrox, 2003.

Dissertação de Mestrado apresentada por **Eduardo Gonçalves Calábria** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título , **"Hermes - Um Middleware Orientado a Mensagem para Ambientes Corporativos",** orientada pelo **Prof. Nelson Souto Rosa** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Carlos André Guimarães Ferraz
Centro de Informática / UFPE

Prof. Gledson Elias da Silveira
Departamento de Informática e Matemática Aplicada / UFRN

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 8 de março de 2004..

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.