PUC-Rio - Certificação Digital Nº 0321107/CA

2
Middlewares Adaptáveis

Middlewares, na sua essência, são *softwares* que auxiliam a gerir diferenças e complexidades entre infra-estruturas distribuídas e que facilitam a comunicação entre sistemas heterogêneos. Com eles, a tarefa de se criar sistemas onde se utiliza de maneira mais eficiente recursos de várias máquinas é extremamente facilitada. Algumas vezes, até mesmo a demanda por um serviço inviabiliza a utilização de uma solução centralizada, criando a necessidade de se usar um sistema escalável, distribuindo partes do sistema por diversas máquinas. Outro uso comum é a integração de sistemas já estabelecidos, que podem estar sendo executados em plataformas de *hardware* e *software* muito díspares e cuja reimplementação seria inviável.

A criação de sistemas distribuídos diretamente através da utilização das primitivas em sistemas operacionais de rede pode tomar muito tempo e, com isso, tornar o processo de desenvolvimento mais caro e sujeito a erros. Com os *middlewares*, a tarefa de construir esses sistemas se torna mais fácil, por eles proverem um ambiente de programação mais amigável para a construção dos mesmos. Além disso, a idéia de um *middleware* é prover ao desenvolvedor final uma interface de programação de um nível mais alto. Com isso, o desenvolvedor pode se preocupar mais com as funcionalidades requeridas pelo sistema final. Portanto, uma definição em (08) é que o *middleware* é qualquer camada de *software* que é colocada sobre a infra-estrutura do sistema distribuído, isto é, o sistema operacional, a rede e a interface de programação de ambos, e abaixo da camada de aplicação. O *middleware* fica então responsável por esconder a comunicação entre os processos, mascarar as diferenças de *hardware*, sistema operacional e protocolos de comunicação e, se possível, oferecer mais de um modelo de programação, em diferentes linguagens.

As dificuldades de construção de um sistema distribuído são melhor apresentadas em (09). O mesmo trabalho propõe uma classificação para *middlewares* tradicionais, dividindo-os em quatro grupos: *middlewares* transacionais, *middlewares* orientados a mensagens, *middlewares* procedurais e *middlewares* orientados a objetos.

Um *middleware* transacional (*transactional middleware*) implementa

transações entre componentes que se situam em diferentes sistemas, normalmente ambientes heterogêneos. É usado normalmente para fazer a integração de sistemas de bancos de dados. Exemplos desse tipo de arquitetura são o CICS da IBM (10) e o Tuxedo da BEA (11).

Já um *middleware* orientado a mensagens (*message-oriented middleware*) facilita a comunicação entre processos no ambiente distribuído com a implementação de um fluxo de mensagens entre eles através de uma fila de mensagens. É uma boa opção para a implementação de arquiteturas *publish-subscribe* e serviços de eventos. Normalmente o envio de mensagem é assíncrono, tornando a implementação de chamadas síncronas um pouco mais complicada. No entanto, uma maior escalabilidade em um *middleware* orientado a mensagens é mais fácil de ser obtida. Os sistemas MQSeries da IBM (12) e Java Message Queue da Sun, uma implementação do Java Message Service (13) são exemplos de *middlewares* orientados a mensagens.

Em contrapartida, um *middleware* procedural (*procedural middleware*) estende uma chamada local para uma máquina remota, onde está situada a implementação do procedimento, através de uma técnica denominada *chamadas remotas de procedimento* (RPC). Dessa forma, para o cliente, a chamada na verdade é uma ilusão criada pelo *middleware*. Os parâmetros são codificados do lado do cliente através de um *stub*, enviados ao servidor que os decodifica, executa o procedimento, recodifica os resultados e os repassa ao cliente. Normalmente, as chamadas remotas de procedimento são operações síncronas, ao contrário dos *middlewares* orientados a mensagens. A maioria dos sistemas operacionais modernos adota chamadas remotas de procedimento.

Finalmente, considerados evoluções dos *middlewares* procedurais, os *middlewares* orientados a objetos (*objected-oriented middleware*) combinam as facilidades de uma linguagem orientada a objetos, tais como herança e polimorfismo, e a dinâmica do *middleware* procedural, onde uma chamada local é mapeada, através do middleware, para um objeto remoto, que é onde se encontra a implementação desejada. Dessa forma, um cliente obtém uma referência para um objeto remoto e faz chamadas a essa referência como se o objeto residisse no ambiente local. A infra-estrutura intermedia essa comunicação, cuidando dos detalhes dos níveis mais baixos, tais como codificação e decodificação dos parâmetros e valores de retorno, manutenção da conexão entre o servidor e cliente e transmissão das chamadas e retornos das chamadas entre eles. Exemplos desse tipo de *middleware* são o DCOM (14), CORBA (15) (16), e o Java RMI (17).

PUC-Rio - Certificação Digital Nº 0321107/CA

2.1
Suporte à adaptação em middlewares

Um dos pontos em comum dos *middlewares* tradicionais é que a sua implementação é na forma de uma caixa preta, justamente para facilitar o trabalho do desenvolvedor final na construção do sistema distribuído e permitir que o desenvolvedor da infra-estrutura básica fique livre para fazer qualquer tipo de modificação para aumento de desempenho e confiabilidade. Entretanto, caso um desenvolvedor queira inserir alguma funcionalidade diferente em sua aplicação, funcionalidade que dependa de um comportamento não usual do *middleware*, ele deve ou implementá-la em sua aplicação ou usar alguma estrutura disponibilizada pelo próprio *middleware* para fazê-lo. Por exemplo, alguns dos *middlewares* comerciais permitem interceptações no fluxo de troca de informações da infra-estrutura, criando a possibilidade de se implementar comportamentos diferentes em alguns casos, como tolerância a falhas ou aspectos de segurança (18) .

Muitas vezes, no entanto, pode ser mais interessante mexer na estrutura interna do *middleware* para fazer a modificação, principalmente quando se trata de modificações em níveis mais baixos, como a codificação de dados a serem transmitidos pela rede. Para isso, é necessário que o *middleware* seja construído de forma a permitir essas adaptações.

Em (19) e (20), os autores propõem uma classificação que leva em consideração três dimensões: *onde*, *como* e *quando* é feita a adaptação no *middleware*. Na primeira dimensão, *onde*, o *middleware* é classificado de acordo com a camada do *middleware* em que acontece a modificação, usando a divisão mostrada na seção 2.1.1. A segunda dimensão, *como*, é ligada às técnicas de adaptação descritas na seção 2.1.2. Finalmente, a dimensão *quando* sugere em que momento a modificação de comportamento é feita em um sistema distribuído. A classificação segundo o momento da adaptação é descrita na seção 2.1.3.

2.1.1
Camadas de um middleware

Em (21), D. Schmidt divide um *middleware* em camadas, como mostradas na figura 2.1. Usamos essas camadas como base para a classificação, mostrando em que parte do *middleware* a adaptação pode ser feita.

– Infra-estrutura: Nesta camada o *middleware* encapsula mecanismos de comunicação e concorrência do sistema operacional, criando componentes reutilizáveis, abstraindo diferenças entre sistemas operacionais.

PUC-Rio - Certificação Digital Nº 0321107/CA

Figura 2.1: Camadas de um middleware (figura retirada de(21)).

– Distribuição: Esta camada fornece ao desenvolvedor uma abstração de programação mais alto-nível, como as chamadas remotas de procedimento descritas anteriormente. Situada acima da camada de infra-estrutura, a camada de distribuição esconde ainda mais os ambientes heterogêneos onde o *middleware* é executado, tais como protocolos de comunicação e linguagens de programação.

– Serviços comuns: Alguns serviços comuns a vários tipos de aplicação, como serviços de eventos, serviços de nomes, transações e persistência de dados podem ser implementados no middleware, deixando o desenvolvedor livre para tratar da lógica do sistema.

– Serviços de domínio: Enquanto os serviços comuns descritos acima podem ser utilizados em vários tipos de aplicação, serviços de domínio são os serviços específicos para um tipo particular de aplicação. Um exemplo são middlewares específicos para sistemas de aviação, que possuem informações e métodos reutilizáveis apenas para outros sistemas ligados à essa área.

PUC-Rio - Certificação Digital Nº 0321107/CA

2.1.2
Técnicas de adaptação

A forma mais simples de adaptação de um *middleware* é a chamada adaptação por parâmetros. Durante a implementação, o desenvolvedor do *middleware* pode adicionar comportamentos variáveis de acordo com parâmetros configurados no momento da execução. Dessa forma, a adaptação é limitada a esses pontos de configuração e a um grupo finito de parâmetros possíveis definidos pelo desenvolvedor.

Uma forma mais flexível de adaptação é através da chamada adaptação por composição. Esta forma se diferencia da adaptação por parâmetros por permitir a modificação e a adição de funcionalidades não antecipadas pelo desenvolvedor do *middleware*. Além da técnica de orientação a objetos, três tecnologias se tornaram importantes para a adaptação por composição: a reflexão computacional, o projeto baseado em componentes e a programação orientada a aspectos.

Reflexão computacional

A reflexão computacional é a técnica através da qual um programa pode descobrir detalhes de sua própria implementação em tempo de execução e, em alguns casos, modificar seu próprio comportamento de acordo com essas informações. As técnicas de reflexão tornam possível mostrar a implementação sem revelar detalhes desnecessários nem comprometer a portabilidade dela (22). O acesso a essa implementação é feito através de um *protocolo de meta-objeto* (MOP), que a disponibiliza em um meta-nível, como representado na figura 2.2.

Alguns exemplos de serviços são a inclusão de ações antes ou depois de invocações de métodos, ou mudanças de parâmetros internos de objetos. O estudo em reflexão é feito em linguagens de programação (22) (23) (24), sistemas operacionais (25) e em sistemas distribuídos (03).

A reflexão pode ser dividida em dois tipos: a *introspecção*, que é a capacidade de uma aplicação observar sua estrutura e comportamento; e a *intercessão*, que é a modificação da estrutura ou comportamento de acordo com as informações obtidas da introspecção ou mesmo de informações externas. Algumas linguagens de programação e *middlewares* que fazem uso da reflexão computacional separam explicitamente essas duas partes (26). Também há a divisão da reflexão em *estrutural*, que atua na interconexão entre objetos, nos seus tipos e na sua hierarquia, e *comportamental*, que tem o foco na semântica da aplicação.

PUC-Rio - Certificação Digital Nº 0321107/CA

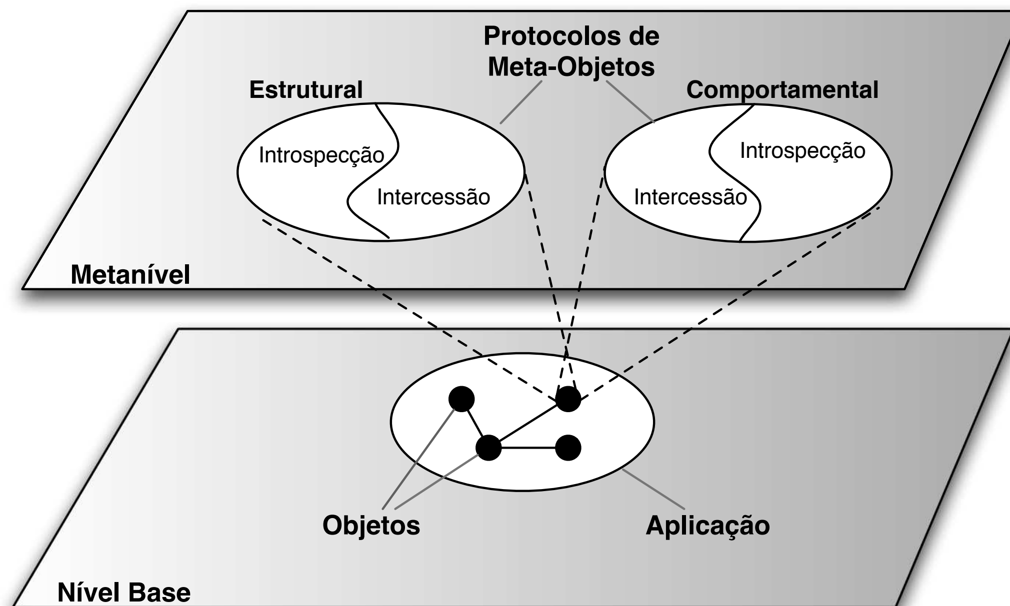

Figura 2.2: Relação entre o nível base e o meta-nível na reflexão computacional (figura retirada de (20)).

PUC-Rio - Certificação Digital Nº 0321107/CA

Projeto baseado em componentes

Uma extensão da orientação a objetos é a programação orientada a componentes (27). A idéia é separar ainda mais as unidades funcionais de código umas das outras, criando objetos que têm tanto a interface exportada, i.e. as funções que este objeto executa, quanto a declaração de quais outros objetos eles dependem, criando assim os pré-requisitos para o bom funcionamento do sistema. Os componentes são criados de forma a serem produzidos independentemente uns dos outros, possibilitando o reuso dos mesmos quando da implementação de um sistema. Em muitos modelos de componentes, o reuso é feito com componentes binários, mesmo sendo gerados a partir de linguagens de programação diferentes.

A construção de um sistema baseado em componentes pode ser feita em tempo de compilação ou em tempo de execução, desde que os pré-requisitos dos componentes sejam satisfeitos. Através da ligação de componentes em tempo de execução, a ligação (*binding*) de novos componentes no lugar de antigos pode ser usada para fazer adaptações. Exemplos de *middlewares* baseados em componentes são o DCOM, EJB (28) e o CCM (29).

Aspectos

A utilização de técnicas como a programação orientada a objetos ou a programação orientada a componentes visa ajudar na construção de sistemas

devido à possibilidade de reuso de partes de código já testadas e usadas em outros sistemas. No entanto, há alguns problemas de programação que não podem ser atacados por essas técnicas, porque existem funcionalidades que, por definição, estão espalhadas por todo um sistema, tornando difícil a sua manutenção. Essas funcionalidades que permeiam o sistema como um todo são chamados de *aspectos*. A programação orientada a aspectos (30) foi criada de forma a expressar claramente quais são os aspectos e como descrever a composição e reuso deles.

Em outros paradigmas de programação, através de um processo conhecido como separação de interesses [1], decompõe-se o sistema em unidades funcionais, ou seja, unidades que, dada uma entrada produzem uma saída bem definida. Para se criar operações mais complexas, basta se utilizar de outras unidades funcionais mais simples e combiná-las de forma a se ter a saída desejada. Entretanto, ao fazer isso, o sistema pode estar sendo ineficiente, pois não há reuso de funções entre os componentes, já que cada unidade funcional no processo trabalha como se não existisse nada além da estrutura interna dela. No entanto, se houver a tentativa de fazer o processo ficar mais eficiente através de uma visão global, o encapsulamento das unidades funcionais pode ser quebrado.

A programação orientada a aspectos tenta resolver esse problema. Através da definição de pedaços de código e uma lista de pontos onde esses pedaços de código podem ser inseridos, um compilador de aspectos pode fazer a inserção desses pedaços de código nos pontos específicos, adicionando funcionalidades antes não existentes. Este processo é chamado *weaving*, que pode ser automatizado para diminuir a possibilidade de erro. A primeira implementação de uma linguagem orientada a aspectos, AspectJ (31), demonstra como se pode alterar o comportamento do código usando *advices*, que são os comportamentos adicionais, em *join points*, pontos em um código, e que são especificados de acordo com uma expressão chamada *pointcut*, que é a forma de detectar se naquele ponto deve-se ou não inserir o novo comportamento. Usando esse modelo, a manutenção do código ficaria mais fácil, pois ela pode ser feita no pedaço de código original ao invés de ir em cada um dos pontos onde ele seria inserido e modificá-lo.

2.1.3 Momento da adaptação

De acordo com o momento da adaptação, um *middleware* pode ser mais ou menos dinâmico. Normalmente, uma adaptação feita em um momento

[1] Do inglês *separation of concerns*.

PUC-Rio - Certificação Digital Nº 0321107/CA

posterior, isto é, durante a execução, permite formas mais poderosas de mudanças de comportamento, mas cria problemas de manutenção da corretude do programa e de lidar com possíveis inconsistências na sua execução. Os diferentes momentos possíveis, representados na figura 2.3, são discutidos a seguir.

Adaptação estática

Chama-se de adaptação estática aquela que é feita durante o tempo de desenvolvimento, tempo de compilação ou no tempo de carregamento do programa. Um *middleware* adaptável em tempo de desenvolvimento é considerado *hardwired* e depende de uma modificação no código para acontecer.

Já um *middleware* adaptável em tempo de compilação é chamado de customizável. Nessa categoria se incluem a maioria dos *middlewares* baseados em aspectos (32) (33), que utilizam o processo de *weaving* para inclusão de novas funcionalidades ou funcionalidades específicas para o ambiente em que o *middleware* vai ser executado.

Finalmente, o último momento de adaptação considerado estático é o que acontece no carregamento do sistema distribuído, chamando o *middleware* de configurável. De acordo com uma configuração pré-estabelecida, o *middleware* pode ser customizado para carregar só os componentes mais importantes quando ele estiver sendo executado em um ambiente mais limitado, enquanto pode carregar serviços menos importantes quando os recursos disponíveis forem mais fartos. Em alguns ambientes, como em Kava (34), a própria implementação de um componente pode ser modificada em tempo de carregamento através da modificação dos *bytecodes* das classes cujo comportamento o sistema queira mudar.

PUC-Rio - Certificação Digital Nº 0321107/CA

Adaptação dinâmica

A implementação de mudanças no *middleware* durante o tempo de execução nos fornece uma flexibilidade maior. Pode-se dividir em duas categorias, cuja diferença é a possibilidade ou não de mudança da lógica de execução de um componente depois que ele começa a ser usado.

Em *middlewares* ajustáveis (*tunable*), o *middleware* tem a capacidade de modificar-se depois do começo da sua execução, mas antes de a parte modificada estar sendo usada ativamente. Dessa forma, o *middleware* se mantém intacto durante essas modificações, porque componentes que estão sendo utilizados não podem ser alterados. A diferença dessa categoria para os *middlewares* configuráveis é que, enquanto os configuráveis instanciam o seu conjunto de componentes durante a inicialização do sistema, os ajustáveis podem instan-

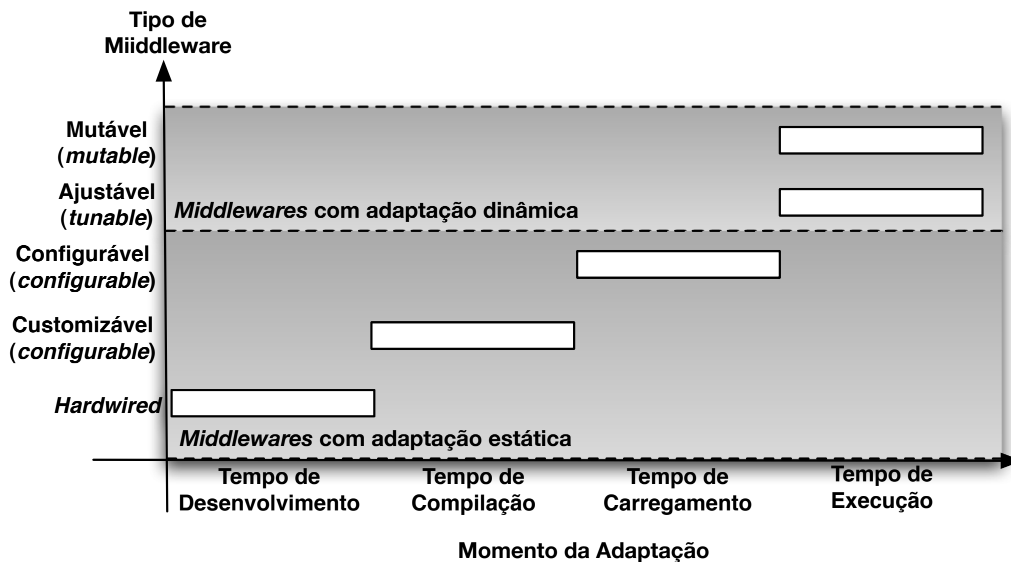

Figura 2.3: Classificação de acordo com o momento da adaptação.

ciar os componentes à medida que eles forem utilizados. Por exemplo, se um objeto requisitar um determinado protocolo para a comunicação, o *middleware* ajustável pode carregá-lo durante a execução do sistema.

Já em *middlewares* mutáveis, tanto o *middleware* quanto a aplicação podem ser adaptados durante a execução, enquanto estão sendo usados. A adaptação pode inclusive inserir mudanças na implementação dos componentes em uso. Embora essa possibilidade seja mais interessante do ponto de vista da flexibilidade, a manutenção de um estado consistente do *middleware* se torna mais complicada, sendo necessários mecanismos mais avançados para isso, como a reflexão computacional e *weaving* dinâmico de aspectos.

PUC-Rio - Certificação Digital Nº 0321107/CA

2.2 Middlewares com adaptação estática

Os *middlewares* com adaptação estática já estão presentes em soluções comerciais, provando sua eficiência como soluções de adaptação. No entanto, as modificações possíveis tendem a ser mais limitadas que os *middlewares* com adaptação dinâmica. Elas incluem, normalmente, configurações estáticas do middleware, de modo a iniciá-lo com o comportamento desejado, mas não permitem que o comportamento seja dinamicamente modificado durante a sua execução. Os *middlewares* apresentados a seguir foram classificados como possuindo adaptação estática devido a abranger adaptações no tempo de compilação ou no momento da sua inicialização.

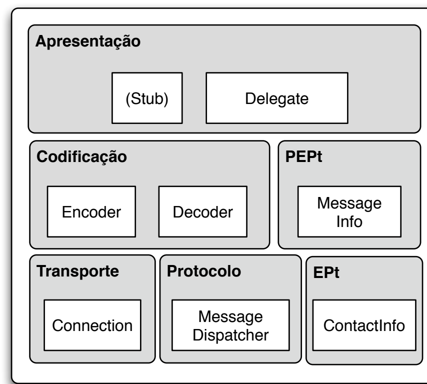

Figura 2.4: Arquitetura PEPt do lado do cliente.

2.2.1 PEPt

O PEPt (01) é uma forma em alto-nível de estruturar o projeto e implementação de sistemas RPC. Ele o faz através da criação de uma visão, independente de linguagem, dos elementos básicos que compõem um sistema remoto: apresentação, codificação, protocolo e transporte – PEPt [2]. O modelo PEPt assume que a troca de informações nos sistemas RPC se dá através de mensagens, podendo assim ser utilizado como um modelo de construção de um *middleware* baseado em mensagens. Um modelo RPC, então, nada mais é que uma interface que encapsula a chamada remota em uma mensagem e a respectiva resposta está contida em uma mensagem recebida. O PEPt é base da implementação de CORBA e JAX-XML no Java 1.5 e está presente no SunOne Application Server.

O bloco de apresentação é definido como as APIs para a interação com o serviço remoto, os tipos de dados a serem transmitidos e eventuais tratamentos de erro; o bloco de codificação consiste na representação de dados em baixo-nível a serem transmitidos pela rede, acordada entre o cliente e o servidor, assim como a tradução de dados alto-nível para esse baixo-nível e vice-versa; o bloco de protocolo é responsável por encapsular os dados em uma mensagem, colocando nela informações de destino e quaisquer outras informações necessárias para o funcionamento da chamada; finalmente, o bloco de transporte é quem move a mensagem de um ponto a outro. Desta forma, um sistema distribuído pode ser criado de tal forma que ele seja compatível com padrões pré-existentes ou crie protocolos novos, apenas por definir uma implementação para um ou mais blocos.

Há dois outros blocos que controlam o fluxo de execução da comunicação

[2]do inglês *presentation*, *encoding*, *protocol* e *transport*

PUC-Rio - Certificação Digital Nº 0321107/CA

e servem como fábricas dos blocos descritos acima. O bloco fábrica, chamado *ContactInfo* (35) no lado do cliente e *Acceptor* (36) do lado do servidor, contém uma composição dos três tipos de blocos: codificação, protocolo e transporte (também chamado de EPt). Para cada composição de blocos diferente, é criado um EPt na inicialização, dando a possibilidade de escolha por parte do *middleware* de qual bloco EPt ele irá utilizar na comunicação em uma chamada remota.

Cada um dos blocos do PEPt é responsável por uma parte da operação de um sistema RPC e contém objetos que o representam ou o ligam com os outros blocos da arquitetura, mostrando assim os contratos explícitos entre eles. Implicitamente ou explicitamente, todos os sistemas RPC possuem as operações descritas por esses blocos. Entretanto, há às vezes a necessidade de se oferecer funcionalidades que precisam de uma integração maior entre os blocos. Um exemplo é o suporte à fragmentação por parte do protocolo, pois nesse caso o bloco de codificação deve avisar ao protocolo quando seu *buffer* está completo, mesmo que o processo inteiro ainda não tenha acabado. Dessa forma, a interação não será apenas o protocolo pedir a codificação de uma mensagem para o codificador, mas uma interação mais complicada. O modelo PEPt deixa em aberto esse tipo de integração.

O fluxo de comunicação de um cliente fazendo uma requisição (figura 2.5) passa por todos os blocos, na seguinte ordem: o cliente obtém a referência do objeto remoto, criando um objeto *Delegate*. Quando é feita a requisição a esse *Delegate*, a arquitetura PEPt obtém uma conexão de transporte. Para isso, é necessário decidir qual o tipo de conexão e, com isso, usar a fábrica *ContactInfo* para criar o EPt correspondente. Nesse ponto, o *ContactInfo* pode escolher qual o melhor transporte para a dada referência remota, se ela possuir mais de um tipo de transporte. A conexão obtida, então, é usada para enviar e receber dados, que devem ser previamente codificados a partir dos dados alto-nível do bloco apresentação. O bloco de codificação também é obtido da fábrica *ContactInfo*, não havendo assim uma ligação direta entre codificação e conexão. Com um codificador, os argumentos da chamada remota são passados para a representação escolhida e, nesse ponto, devem ser transmitidos pelo transporte. Portanto, os dados codificados são encapsulados nas estruturas de mensagem do bloco protocolo, colocando os cabeçalhos necessários. O bloco protocolo também é obtido da fábrica *ContactInfo*, sendo assim o último bloco do grupo EPt a ser escolhido.

O bloco protocolo, nesse momento, envia a mensagem e é responsável por esperar por uma resposta. Essa operação é dependente do protocolo utilizado, que pode ser simplesmente ficar bloqueado na conexão esperando uma resposta

PUC-Rio - Certificação Digital Nº 0321107/CA

ou tratar múltiplas mensagens em uma única conexão, o que nesse caso cria um tratador através do *ContactInfo* para cada mensagem recebida. Dada essa resposta, o tratador, que faz parte do bloco protocolo, pede ao *ContactInfo* um objeto decodificador, que abre a mensagem e transforma os dados na representação mais alto nível, repassa-os ao bloco apresentação e retorna o controle à aplicação.

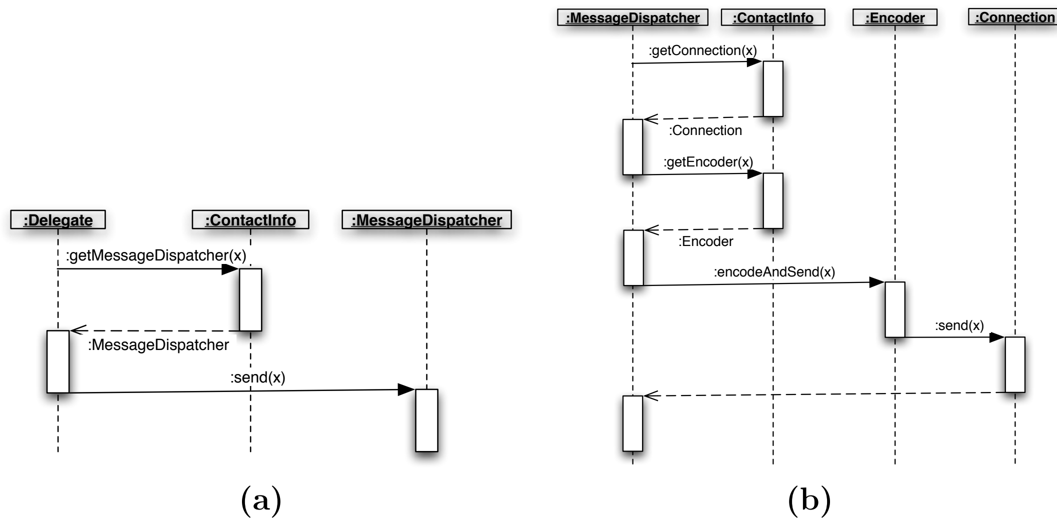

Figura 2.5: Diagramas de seqüência da comunicação do lado do cliente no PEPt (retirados de (35).)

A comunicação do lado do servidor usa praticamente os mesmos passos no sentido contrário. Uma descrição mais detalhada desses passos está disponível em (01) e (36). Podemos ver que o modelo do PEPt é geral o suficiente para a estruturação de sistemas RPC de uma forma simples e, ao mesmo tempo, deixa em aberto a possibilidade da implementação de funcionalidades particulares para um protocolo específico. Mesmo assim, a idéia do modelo é que o bloco de apresentação não precise saber das características da conexão, transporte ou protocolo escolhidos, podendo até mesmo, se necessário, trocar esses três blocos entre duas chamadas sem que o bloco de apresentação o saiba. Isso pode, por exemplo, criar a possibilidade de uma adaptação dinâmica do protocolo utilizado entre duas chamadas diferentes.

2.2.2
.NET Remoting

O .NET Remoting (37), parte do .NET (02), é um *framework* que permite objetos de aplicações diferentes interagirem entre si. Os serviços oferecidos pelo *framework* incluem meios de ativação de objetos a partir do cliente ou a partir do servidor, canais de comunicação para transportar mensagens entre

PUC-Rio - Certificação Digital Nº 0321107/CA

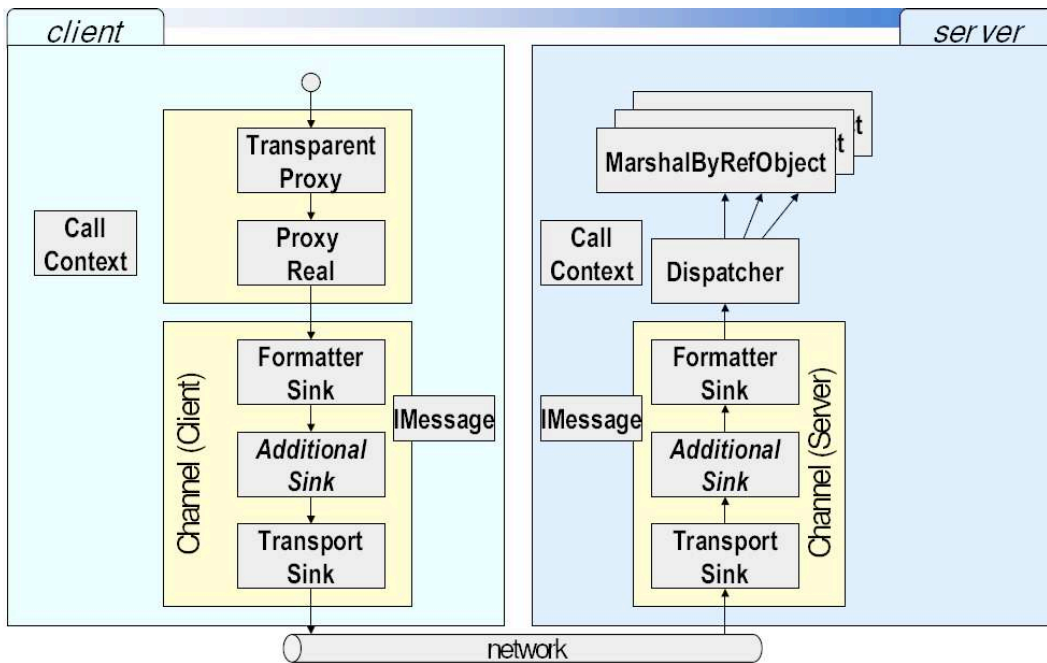

Figura 2.6: Arquitetura .NET Remoting

PUC-Rio - Certificação Digital Nº 0321107/CA

aplicações remotas além de oferecer serviços de controle do tempo de vida dos objetos remotos, facilitando a implementação de serviços distribuídos. Provê também formatadores para a codificação e decodificação de mensagens antes de serem transmitidas por um canal de transporte e maneiras de interceptar o canal para se ter acesso à mensagem antes de ser efetivamente enviada. A arquitetura do .NET Remoting é representada na figura 2.6.

O *framework* diferencia objetos remotos e objetos locais. Os objetos, mesmo que estejam situados na mesma máquina, mas em espaços de endereçamento diferentes, são considerados remotos. Objetos locais são passados por referência, mas, por essas referências serem válidas apenas dentro do mesmo espaço de endereçamento, elas não podem ser passadas como parâmetro ou retornadas como resultado em chamadas remotas. Todos os objetos locais que forem utilizados em chamadas remotas são passados por valor e, para isso, são marcados com um atributo *serializable* ou implementam a interface *ISerializable*. Um objeto local pode ser transformado em um objeto remoto derivando de *MarshalByRefObject*, que cria uma camada de indireção de acesso ao objeto. Assim, qualquer chamada a esse novo objeto é na verdade uma chamada a um *proxy* que transforma as chamadas em mensagens, enviadas através da infra-estrutura ao objeto remoto, e que recebe as respostas, retornando-as ao chamador. A infra-estrutura também é responsável por verificar se o objeto remoto na verdade está situado na mesma aplicação do chamador e, com isso, evitar custo adicional de envio e recebimento de mensagens, invocando o objeto

localmente.

No lado do cliente, o *proxy* cuida para que as chamadas feitas a ele sejam direcionadas ao objeto remoto correspondente. Quando ele é criado, um objeto *TransparentProxy* é retornado ao cliente, contendo informações sobre a interface do objeto remoto. Todas as chamadas a este *proxy* são interceptadas e validadas como funções existentes. Nesse momento, a infra-estrutura descobre se o objeto é local ou não. Se for local, a chamada é efetuada diretamente. Senão, os parâmetros da chamada são inseridos em um objeto *IMessage* e repassados a um *RealProxy*, que se responsabiliza pela comunicação com o objeto remoto propriamente dito.

No .NET Remoting, objetos remotos podem ser criados tanto a partir do cliente quanto a partir do servidor. Se ele for criado a partir do cliente, é ele quem controla o tempo de vida do objeto através de um sistema de *lease*. Já no servidor, ele é registrado normalmente no começo da aplicação, através das informações de referência (onde está a implementação do objeto, o tipo do objeto, a localização dele em uma rede (URI) e o modo de ativação), e continua até ela ser terminada. Uma maior discussão sobre o tempo de vida dos objetos pode ser encontrada em (38).

A parte mais importante do .NET Remoting sob o ponto de vista da adaptação consiste nos chamados *Channels*, que são os objetos responsáveis pelo transporte dos dados entre objetos remotos. Um servidor, em sua inicialização, pode registrar um ou mais canais associados a ele e, havendo mais de um, um cliente pode escolher qual a melhor opção de canal para fazer a chamada remota. Além de poderem ser usados os canais já incluídos na distribuição do .NET Remoting, novos canais podem ser implementados, usando estruturas de mensagem ou protocolos diferentes quando necessário.

Todos os canais derivam de uma interface padrão, chamada *IChannel*, e devem implementar as funções de envio, as de recebimento, ou ambas, dependendo do objetivo do canal. Canais são registrados antes dos objetos que os utilizam, e é necessário haver pelo menos um canal registrado na infra-estrutura. A chamada, quando feita, é enviada pelo *RealProxy* correspondente a uma cadeia de objetos que recebe os parâmetros, processa-os, serializa-os e finalmente os envia pela conexão de transporte. O primeiro dos objetos dessa corrente normalmente é um formatador que transforma as mensagens em um *stream* de *bytes* e o último é um objeto que cuida da conexão entre o cliente e o servidor. Alguns canais já são implementados no .NET Remoting, como um canal TCP e um HTTP, e formatadores, como um formatador SOAP e um formatador binário. Há também a possibilidade de se combinar canais e formatadores, como por exemplo usar um canal TCP e um formatador SOAP,

PUC-Rio - Certificação Digital Nº 0321107/CA

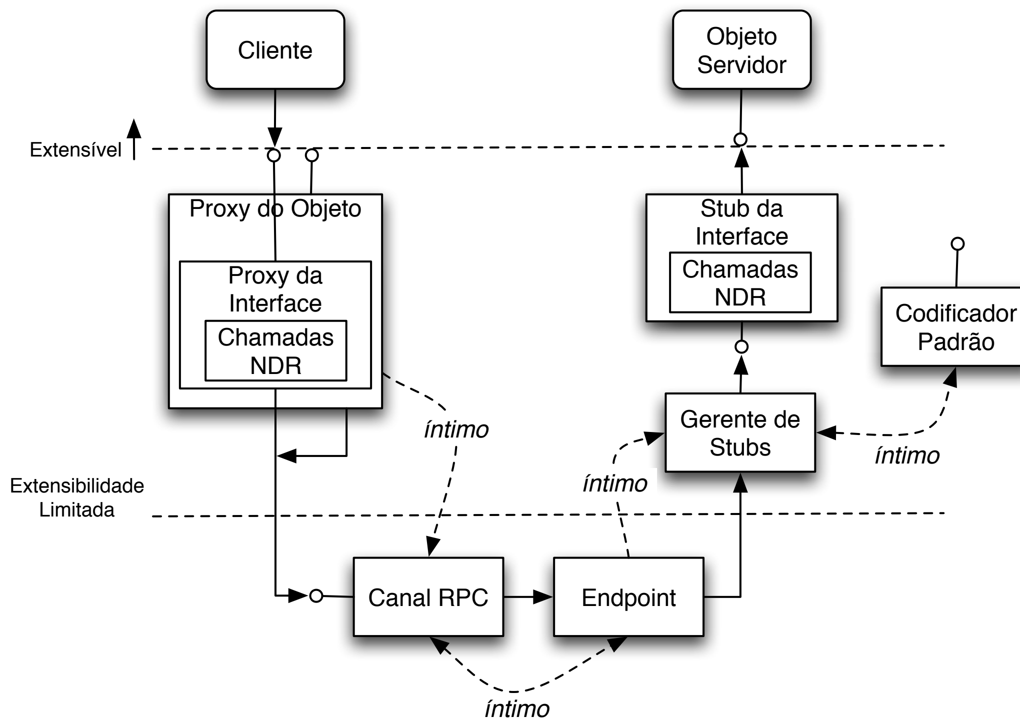

Figura 2.7: Arquitetura COM (figura retirada de (41)).

ou um canal HTTP com um formatador binário.

O projeto Remoting.CORBA (39) é um exemplo de infra-estrutura que usa a implementação de novos objetos *Channels* para estender o *framework* .NET de forma que objetos .NET possam se comunicar com aplicações CORBA usando o protocolo IIOP e vice-versa.

2.2.3 COMERA

O COMERA propõe uma arquitetura remota extensível baseando-se no COM (40), aproveitando dois pontos principais: o COM já possui a possibilidade de extensão, usando um mecanismo chamado *marshalling* customizado, podendo um objeto criar uma nova maneira de comunicação sem modificar a implementação da arquitetura padrão; o COM também já é componentizado, fornecendo a base para construir aplicações baseadas em componentes. Portanto, a arquitetura pode ser construída em tempo de execução através da instanciação e conexão de componentes dinâmicos.

No COM, cada componente criado deve oferecer uma interface *IMarshal*. Se essa interface não for oferecida explicitamente, o próprio COM se encarrega de usar o *marshaller* padrão, que cria uma referência para o objeto para passagem ao cliente. O *unmarshaller* do cliente, por sua vez, cria um *proxy* padrão a partir dessa referência, para a utilização do objeto remoto de forma transpa-

PUC-Rio - Certificação Digital Nº 0321107/CA

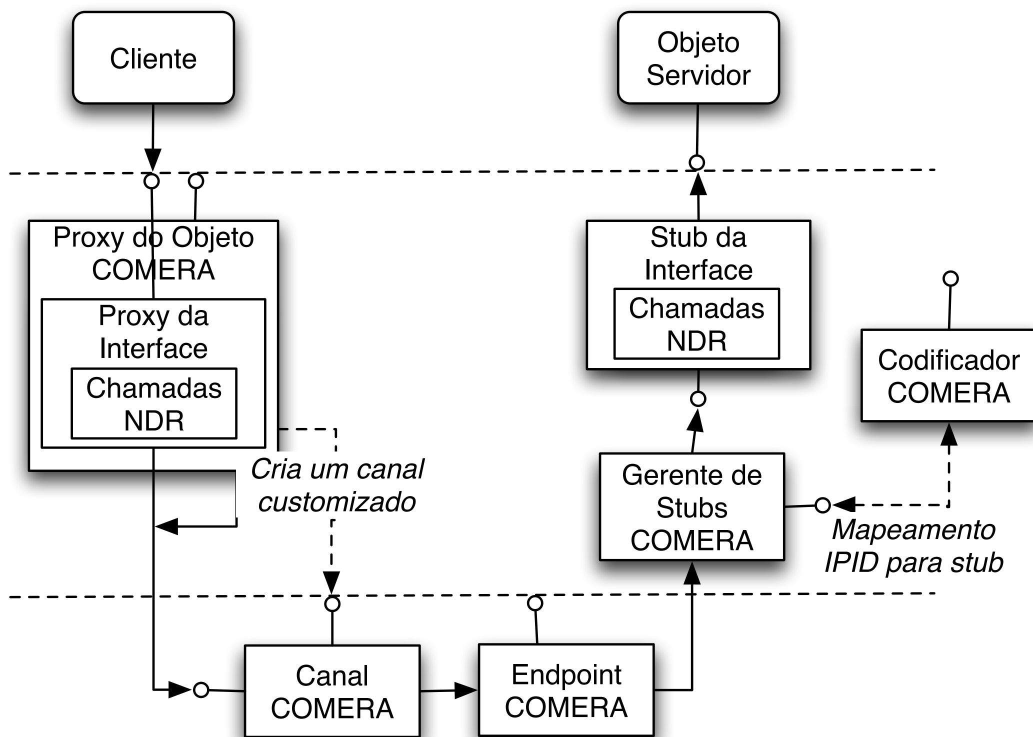

Figura 2.8: Arquitetura COMERA (figura retirada de (41)).

PUC-Rio - Certificação Digital Nº 0321107/CA

rente. Associado a esse *proxy*, é carregado pelo COM um objeto canal RPC, que usa a informação do *endpoint* contida na referência para se comunicar com o servidor. Cada chamada a esta referência remota entra no *proxy*, é codificada pelo *marshaller* para o formato Network Data Representation (42), enviada pelo objeto canal para o servidor, decodificada e entregue ao objeto correspondente no servidor. Um objeto pode mudar a implementação da codificação, implementando a interface *IMarshal* especialmente para o fim desejado. No entanto, para isto, o objeto tem que implementar não só a nova codificação, mas também os *proxies* e *stubs* que utilizam essa codificação. A figura 2.7 mostra a relação pouco extensível entre os componentes da arquitetura.

Aplicações podem não querer reimplementar todas as partes de uma arquitetura remota para modificar uma parte pequena da arquitetura, preferindo usar partes do sistema padrão. Para isso, é necessário que a própria arquitetura remota seja componentizada, para que o reuso de partes e peças seja possível em um nível mais baixo. O COMERA propõe para isso usar um novo *marshaller* e redesenhar partes da arquitetura para aumentar a extensibilidade.

É criado portanto um objeto COM que funcione como gerente de *stubs* e que, se necessário, possa ser substituído, como por exemplo para mudar forma de despacho de chamadas, sem ser preciso mudar o *marshaller*. O *endpoint* no

COMERA também é um objeto COM, podendo ser substituído para a inclusão de algum gancho antes da transmissão. Finalmente, os canais são considerados como componentes, tanto para deixar que um canal customizado seja utilizado por implementações feitas com componentes padrão, como para componentes customizados poderem fazer uso do canal padrão do COM.

A figura 2.8 apresenta a arquitetura criada pelo COMERA. Em (41), o autor apresenta essa arquitetura e propõe três exemplos que demonstram os aspectos de adaptação. O primeiro exemplo é a utilização de um novo *marshaller* e um novo *channel* em um sistema para, com isso, criar uma tolerância a falhas. Esses novos objetos são na verdade representantes de um ou mais componentes reais, que podem estar em máquinas diferentes. Assim, caso um dos componentes reais esteja indisponível, esse representante consegue direcionar a chamada para um outro componente. Um segundo exemplo é a mudança do transporte envolvido na comunicação entre componentes. Em redes onde o tráfego pode ser prejudicado pela presença de *firewalls*, às vezes é necessário que o protocolo de comunicação entre os componentes seja diferente. Com o COMERA, o *endpoint* pode ser trocado por um específico para o transporte desejado, encapsulando a implementação desse transporte e a escondendo de camadas superiores da arquitetura. O terceiro exemplo se relaciona com a migração de objetos. Para isso, no lado do cliente troca-se o componente *channel* para se comunicar com o novo objeto.

PUC-Rio - Certificação Digital Nº 0321107/CA

2.2.4 AspectJRMI

O AspectJRMI (33) é uma implementação de chamadas remotas de métodos usando conceitos de programação orientada a aspectos e configurada em tempo de compilação. Através de um compilador de aspectos, programadores podem adicionar e retirar interesses chamados transversais (*crosscutting concerns*), criando um *middleware* específico para uma determinada aplicação distribuída, apenas com as funcionalidades de que ela necessita. Tenta-se assim diminuir a sobrecarga criada pela adição de componentes ao *middleware* e que não são utilizados pela aplicação final. A adaptação é feita em tempo de compilação e os aspectos são implementados usando-se a linguagem AspectJ.

O AspectJRMI é baseado no sistema RME (43) um *middleware* orientado a objetos simples, desenvolvido sobre o perfil MIDP/CLDC da plataforma J2ME, que foi desenhado originalmente para dispositivos com recursos mais limitados. Este *middleware* já fornece componentes que implementam interesses não-transversais, como protocolos de comunicação, protocolos de troca de mensagem e serializadores de dados.

Em cima do núcleo simples, o AspectJRMI implementa novas funcionalidades transversais ao sistema original. Uma delas, os interceptadores de chamadas remotas, adiciona elementos no fluxo de execução das chamadas remotas através de um decorador (44) que implementa o novo elemento. Por exemplo, uma forma de saber quando uma chamada foi feita pode ser incluída dessa maneira com o uso de um decorador que implemente uma função de *log*.

Chamadas *oneway*, i.e. chamadas que não precisam de resposta, e chamadas assíncronas também podem ser especificadas em tempo de compilação. No primeiro caso, usa-se um *pointcut* dizendo qual função será transformada em *oneway*. O compilador AspectJ, usando esse *pointcut*, insere no código da função as diretivas necessárias para essa transformação. Já o segundo caso, a chamada original, síncrona, é trocada por uma implementação vazia que retorna imediatamente um objeto `Future`, que possui um método, *getResult*. Assim, o cliente que utiliza essa chamada remota pode continuar executando até que precise do resultado. Nesse momento, o cliente chama *getResult*, que fornece os valores de retorno da chamada assíncrona, ou uma exceção em caso de erro.

Em (33) há também uma discussão sobre o tamanho do código gerado a partir dos aspectos escolhidos, demonstrando os resultados desse método na comparação do *overhead* do programa final em relação às funcionalidades escolhidas em tempo de compilação.

2.3 Middlewares com adaptação dinâmica

Em algumas aplicações, a adaptação somente no tempo de compilação ou no momento do carregamento do *middleware* pode não ser o mais indicado. Torna-se então necessário fazer adaptações durante a execução do *middleware*, que pode ser feita de tal forma que o programa que é executado sobre o *middleware* esteja ciente dessa modificação, ou mesmo que trocas de comportamento do *middleware* sejam transparentes do ponto de vista da aplicação. No entanto, para chegar a esse nível de customização, a infra-estrutura deve apresentar mecanismos que ajudem a manter a integridade na execução de um programa, principalmente durante o processo de adaptação. Apresentamos a seguir alguns *middlewares* pesquisados que propõem soluções para esses problemas.

PUC-Rio - Certificação Digital Nº 0321107/CA

2.3.1 Fractal

Fractal é um modelo de componentes modular e extensível criado pelo grupo ObjectWeb (45) que pode ser usado com várias linguagens de programação diferentes para implementar aplicações de diversos domínios, inclusive de plataformas de *middleware*. O modelo possui três características principais: a separação da interface e da implementação; programação orientada a componentes; e inversão de controle, que separa um componente em dois aspectos, o funcional e o de configuração, e tira do componente a responsabilidade de se configurar, fazendo com que uma entidade externa seja o agente de uma eventual mudança de seus atributos. A implementação de um *middleware* baseado no modelo do Fractal é completamente livre para escolher os elementos designados nele. O Fractal classifica o *middleware* de acordo com níveis hierárquicos de conformidade, indo desde um modelo de nível 0, um *middleware* implementado somente segundo as práticas da orientação a objetos, até um modelo 3.3, que usa a especificação inteira do modelo.

Os níveis de conformidade com o modelo do Fractal ligados à adaptação dinâmica passam por dois conceitos principais: a introspecção e a configuração. A introspecção é a capacidade de um componente de prover informações sobre as suas características internas. O componente nesse nível tem apenas interfaces externas visíveis, podendo ser interfaces do tipo *cliente*, que invocam operações em outros componentes, e interfaces do tipo *servidor*, que recebem invocações. Pode-se descobrir mais sobre as interfaces do componente através de duas funções: a primeira lista todas as interfaces externas do componente e permite obter uma interface específica; a segunda permite fazer a introspecção de uma interface específica, podendo obter o seu nome, o tipo e a qual componente pertence.

Outro conceito, a configuração, leva em consideração que um componente no Fractal (figura 2.9) é dividido em duas partes: o *controller*, o envólucro exterior, e o *content*, que está contido no *controller*. O *controller* provê níveis de supervisão das unidades internas do componente e implementa as ligações entre o componente que ela engloba com os componentes externos, através de suas interfaces externas. Possui também interfaces internas, que se conectam ao *content*, que é formado por subcomponentes, que recursivamente podem conter outros subcomponentes.

O *controller* pode simplesmente exportar funcionalidades do componente ou interceptar as chamadas feitas para subcomponentes, controlando de alguma forma o comportamento do componente em si, através de uma indireção criada entre interface externa do componente e a interface interna. Além disso,

PUC-Rio - Certificação Digital Nº 0321107/CA

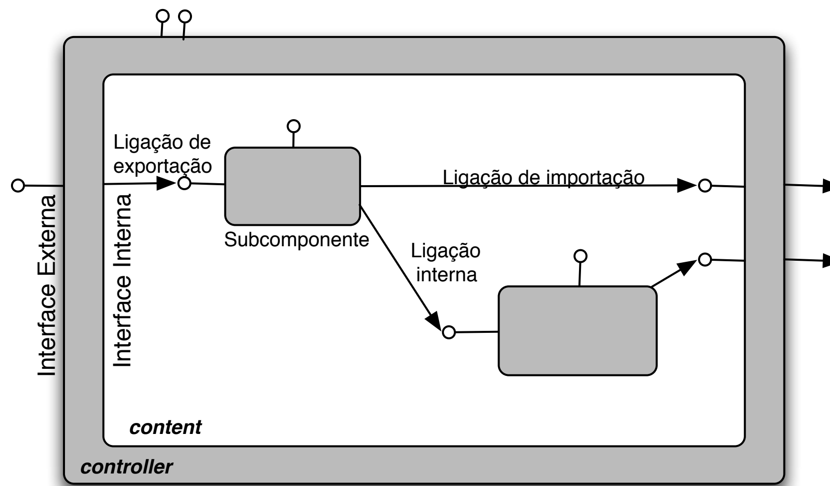

Figura 2.9: Visão interna de um componente no Fractal (figura retirada de (45)).

PUC-Rio - Certificação Digital Nº 0321107/CA

pode exportar uma interface de controle, que não está ligada necessariamente a um subcomponente interno, mas que fornece operações para mudar a maneira de agir dos subcomponentes, tais como controle do seu tempo de vida, ou das ligações entre os componentes internos.

Para mudar um atributo, uma ligação entre componentes ou para remover um subcomponente, é necessário que haja um suporte para que o estado da aplicação continue consistente. Por isso, o componente deve implementar uma interface chamada *LifeCycleController*, que oferece funções para parar a execução do componente, avisando-o de que a configuração vai ser efetuada e, posteriormente, para executá-lo novamente. Para que esse suporte a adaptações dinâmicas seja efetivo, é preciso que o componente implemente essa interface, gerindo os recursos internos de modo a liberar quaisquer entraves que possam acontecer durante a reconfiguração. Além disso, essa interface deve ser implementada de forma recursiva quando necessário, pois, no momento em que uma requisição é feita a um componente dizendo que chegou o momento de uma reconfiguração, seus subcomponentes devem também receber a mesma requisição.

O Julia (46), uma implementação do modelo de componentes do Fractal mantida pelo ObjectWeb, fornece um exemplo de configurações possíveis com esse modelo, principalmente na gerência do ciclo de vida do componente através da interface *LifeCycleController*. Outra implementação, o AOKell (47), usa duas noções um pouco diferentes para a realização de adaptações dinâmicas. Uma delas é o uso de componentes para a construção do próprio *controller*, facilitando reconfigurações no nível de controle e supervisão. Outra facilidade

apresentada pelo AOKell é o uso de programação orientada a aspectos para fazer a cola do nível de controle com a aplicação, separando a implementação do controle e a integração do controle com a aplicação em si.

2.3.2 OpenORB

O OpenORB é uma iniciativa do grupo Next Generation Middleware na Universidade de Lancaster para construir um *middleware* baseado em componentes. As motivação do OpenORB é fornecer suporte a diferentes tipos de aplicação, através de mudanças em vários níveis do middleware. Para isso, é necessário que o *middleware* não seja apenas configurável, mas também dinamicamente configurável e com a possibilidade de evolução à medida que novos problemas forem surgindo.

A idéia por trás dessa configuração dinâmica é a reflexão, isto é, a capacidade de um programa ter acesso e poder modificar a sua representação interna de acordo com um conjunto de regras e com mudanças no ambiente externo. Essa capacidade é dividida em duas partes: a introspecção, que significa a possibilidade de examinar a estrutura interna, como os componentes que formam a estrutura, ou o comportamento, como quais são os recursos utilizados no momento; e a adaptação, seja ela uma pequena reconfiguração, como mudança de um protocolo de comunicação, ou uma evolução, como a adição de serviços que não estavam disponíveis anteriormente.

O OpenORB utiliza como base o modelo de componentes OpenCOM (48), que usa princípios do COM, mas adiciona funcionalidades como: dependências explícitas entre componentes, facilitando a reconfiguração de conexões entre eles; mecanismos de exclusão mútua para auxiliar a reconfiguração; e interceptação antes e depois de chamadas de métodos, permitindo a inserção de controles nessas chamadas que não existiam anteriormente. A arquitetura do OpenCOM é representada na figura 2.10.

No OpenCOM utiliza-se o conceito de interfaces, receptáculos e conexões. A interface representa o que o componente exporta de funcionalidades, enquanto o receptáculo representa uma necessidade, forma pela qual o modelo mostra a dependência entre os componentes. Dessa maneira, uma conexão entre dois componentes só pode ser efetuada durante o tempo de execução se a interface e o receptáculo forem do mesmo tipo. O OpenCOM também possui um componente comum a todos os outros, denominado *OpenCOM* e que gere a conexão e a desconexão de interfaces e receptáculos. Um componente precisa implementar duas interfaces: *IReceptacles*, que oferece as operações para alterar quais interfaces estão conectadas atualmente nos receptáculos do com-

PUC-Rio - Certificação Digital Nº 0321107/CA

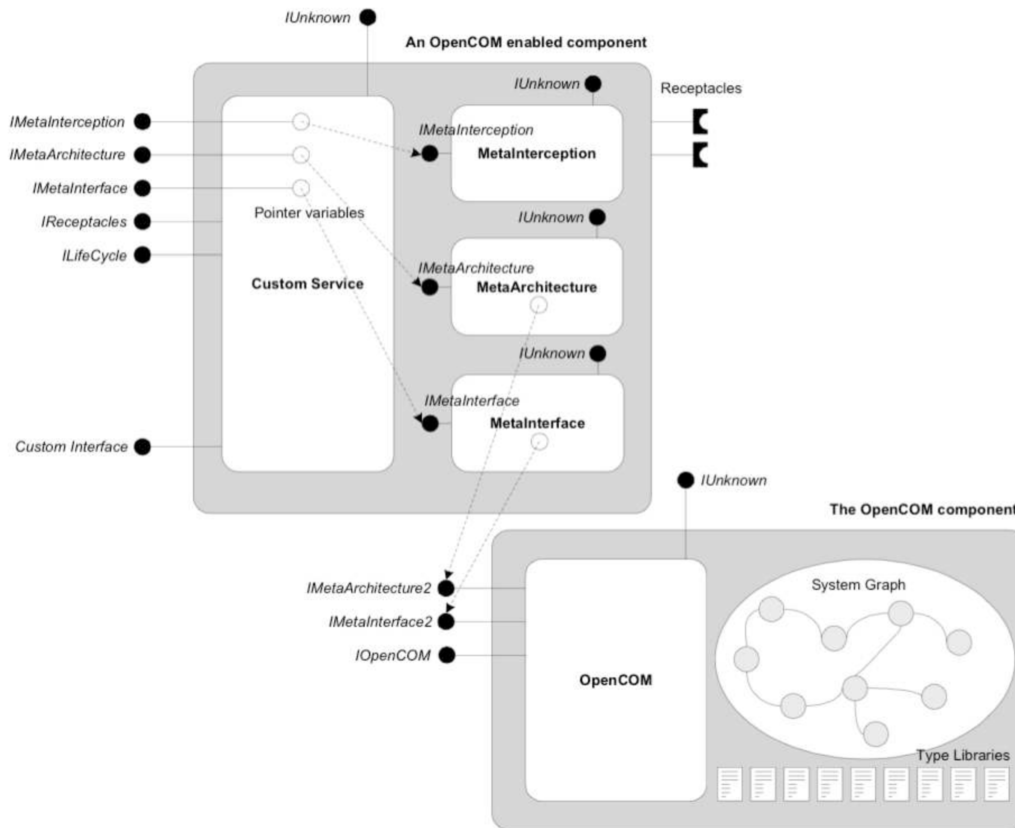

Figura 2.10: Arquitetura do modelo de componentes OpenCOM (figura retirada de (48)).

ponente; e *ILifeCycle*, que oferece as operações equivalentes a um construtor e a um destrutor em uma linguagem orientada a objetos. Ambas as interfaces são usadas pelo gerente único do sistema, que antes de usá-las tem que obter acesso exclusivo para evitar conflitos com invocações que estejam acontecendo no momento do seu uso.

Cada objeto ou interface tem associado a si mesmo um meta-espaço, que é um mecanismo através do qual a reflexão é implementada. Através dele, operações de introspecção e adaptação tornam-se disponíveis. Objetos podem ser agrupados e possuir um meta-espaço associado a esse grupo, onde operações sobre esses objetos podem ser feitas de uma só vez. Um meta-espaço pode ainda ter associado a ele um meta-meta-espaço, criando-se tantos níveis de recursão quanto sejam necessários, de forma a espelhar a reflexão desejada. A implementação dessas funcionalidades é feita a partir de três subcomponentes, presentes em cada componente do OpenCOM, que respectivamente implementam as interfaces *IMetaInterception*, *IMetaArchitecture* e *IMetaInterface*. A primeira permite a inserção de interceptadores em uma interface, e que podem ser invocados antes ou depois de uma chamada nessa interface específica,

PUC-Rio - Certificação Digital Nº 0321107/CA

além de oferecer suporte a mais de um interceptador, se desejado. A segunda fornece identificadores das conexões entre os receptáculos do componente e interfaces de componentes externos. A última permite a inspeção das interfaces e receptáculos declarados pelo componente.

A arquitetura do OpenORB usa o conceito de *framework* de componentes, ou CF, do inglês *component framework*, que é um conjunto de componentes ligados a um domínio específico e de regras que governam a interação entre eles. Um exemplo é um CF de protocolo, que possui os componentes necessários para a composição e reconfiguração de protocolos oferecidos por esse CF. Os CFs são organizados de forma hierárquica no OpenORB, sendo que a arquitetura completa é um CF que engloba outros três níveis de CFs: ligação (*binding*), comunicação (*communication*) e recursos (*resource*), como demonstrado na figura 2.11. A cada CF só é permitido acesso a interfaces de CFs do mesmo nível ou de nível mais baixo.

Com as informações obtidas das interfaces do OpenCOM e os mecanismos apresentados pelo modelo de componentes, o OpenORB usa um processo de gerência da reconfiguração dinâmica, mantendo a integridade durante essa reconfiguração. Um CF mantém meta-informações sobre o estado de sua configuração atual, ou seja, dos componentes que estão sendo utilizados dentro de si próprio, monitora os eventos emitidos por eles e, se for necessário, efetua modificações nos seus atributos ou em conexões entre os componentes. O CF utiliza, durante essas modificações, verificações de conformidade, tais como descobrir que uma instância de um componente não está sendo utilizada antes de removê-la. Muitas vezes, o CF provê uma função de transferência de estado de um componente removido para o novo componente que será colocado em seu lugar.

Em (48) há também uma discussão sobre o desempenho do OpenORB em relação a ORBs tradicionais. De acordo com os resultados, os autores demostram que, mesmo com a inclusão de adaptação dinâmica e indireções necessárias para tanto, o desempenho do OpenORB, e em parte o desempenho do modelo OpenCOM, não é prejudicado.

2.3.3 AspectIX

O AspectIX (49)(50) é uma implementação de CORBA que adota a programação orientada a aspectos como uma maneira de adicionar novas funcionalidades ou de especificar dinamicamente comportamentos diferentes no middleware. Um objeto remoto no AspectIX consiste de vários fragmentos (51) que interagem entre si, cada um com uma descrição derivada de interfaces

PUC-Rio - Certificação Digital Nº 0321107/CA

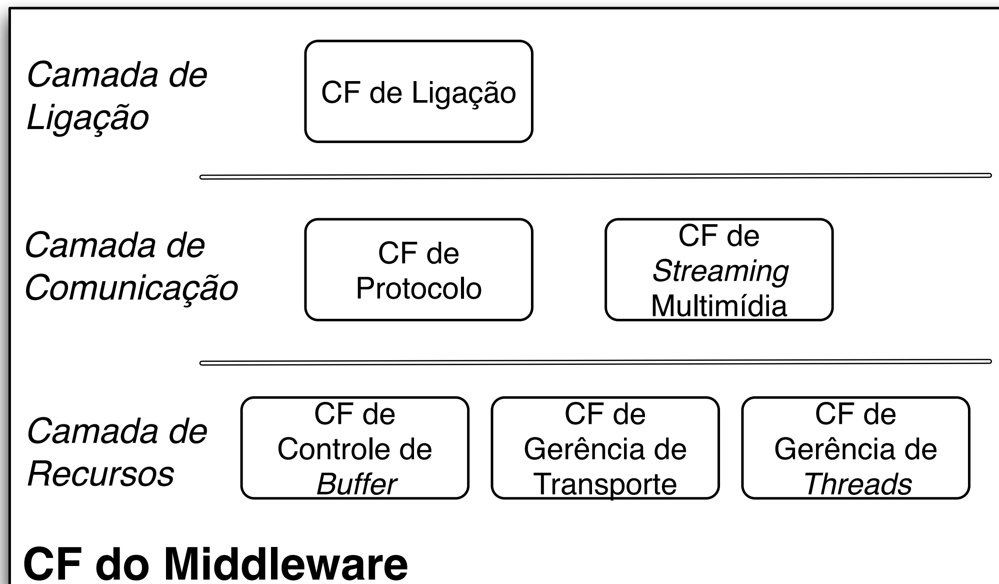

Figura 2.11: Arquitetura do OpenORB (figura retirada de (48)).

PUC-Rio - Certificação Digital Nº 0321107/CA

IDL de CORBA (figura 2.12). Um cliente, por exemplo, precisa de um fragmento local para realizar as chamadas a outros objetos. Um fragmento pode ser um apenas um *stub*, como acontece em um *middleware* CORBA tradicional, que conecta em um outro fragmento, como por exemplo um objeto que contém a implementação do objeto remoto. No entanto, um fragmento pode compreender mais funcionalidades, como guardar dados obtidos do objeto remoto para minimizar os custos de fazer chamadas através da rede repetidamente ou mesmo ser capaz de trocar seu comportamento através da inserção dinâmica de novos aspectos.

Se um fragmento não implementa a configuração desejada, ele pode carregar um outro fragmento que seja capaz de fazê-lo. Além disso, novas implementações de fragmentos podem ser carregadas durante a execução do programa, através de um *weaving* dinâmico de aspectos. O processo de *weaving* é modularizado de uma forma hierárquica. Usam-se unidades chamadas *weavelets*, cujas formas mais simples adicionam código antes ou depois de uma função, adicionam variáveis, mudam parâmetros etc. *Weavelets* mais complexos são declarados pela composição de outros *weavelets* mais simples. No fim do processo, tem-se um *weavelet* que descreve uma implementação específica de um aspecto, bem como as regras necessárias para a decisão de se fazer ou não o *weaving*.

O carregamento de novos fragmentos é configurado através dos *weavelets* escritos pelo desenvolvedor. Quando uma adaptação é requerida, um gerente

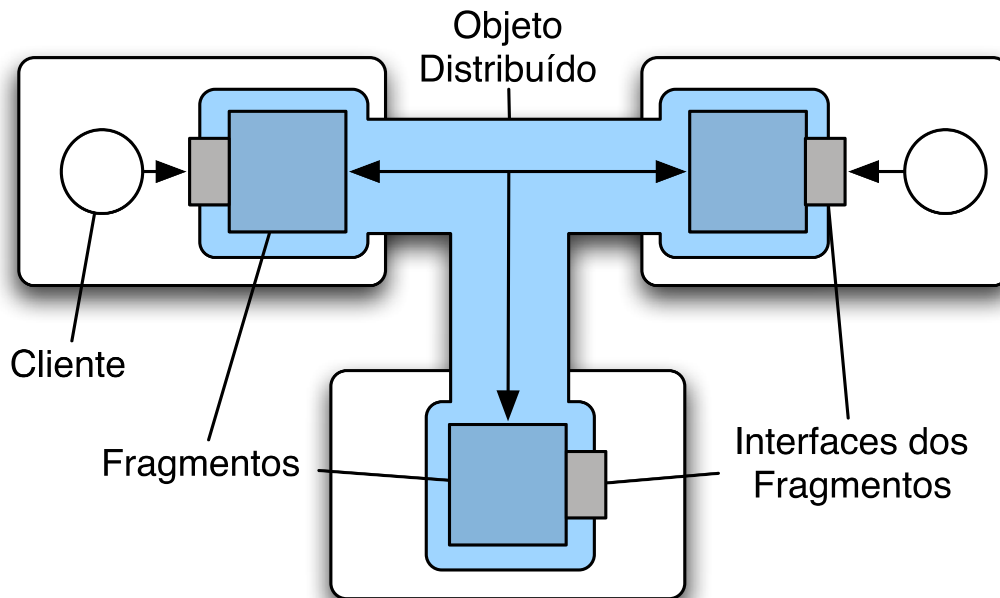

Figura 2.12: Um objeto no AspectIX composto de três fragmentos (figura retirada de (49).

PUC-Rio - Certificação Digital Nº 0321107/CA

de políticas procura o *weavlet* que melhor encaixa com essa requisição, recursivamente resolve as dependências do fragmento de acordo com a política e assim o carrega, de forma transparente para a aplicação.

Fragmentos se comunicam através de *endpoints*, que podem ser sem conexão, com conexão ou baseados em RPC. Um *endpoint* pode, por exemplo, possuir a pilha de protocolo do IIOP para implementar um *stub* padrão de CORBA, permitindo uma interoperabilidade com outros ORBs, ou usar protocolos proprietários se assim o desejar.

2.4 Considerações finais

Os *middlewares* estudados neste capítulo exibem algumas características importantes que serão utilizadas na definição da arquitetura para o OiL no capítulo 4.1. Pudemos observar que todos os *middlewares* neste capítulo utilizam pelo menos uma das técnicas apresentadas na seção 2.1.2, sendo que a maioria usa a separação de interesses através da divisão das partes do sistema em componentes. O projeto baseado em componentes é usado tanto para adaptações estáticas quanto para dinâmicas.

Outra técnica que é utilizada por *middlewares* tanto para adaptações estáticas quanto dinâmicas é a orientação a aspectos. Pudemos notar também que, apesar de a orientação a aspectos auxiliar na adaptação em *middlewa-*

	Como				Quando				O que		
	Orientação a Objetos	Reflexão	Componentes	Aspectos	Customizável	Configurável	Ajustável	Mutável	Troca de protocolos	Troca de transporte	Interceptadores
PEPt	✓					✓			✓	✓	✓
.NET Remoting	✓				✓	✓			✓	✓	✓
COMERA	✓		✓			✓			✓	✓	✓
AspectJRMI	✓			✓	✓						✓
Fractal (Julia)	✓	✓	✓				✓	✓	✓	✓	✓
OpenORB	✓	✓	✓				✓	✓	✓	✓	✓
AspectIX	✓	✓		✓			✓	✓	✓	✓	✓

Figura 2.13: Tabela comparativa entre os *middlewares* estudados

PUC-Rio - Certificação Digital Nº 0321107/CA

res, linguagens disponíveis atualmente com suporte a aspectos, tais como o AspectJ, não oferecem mecanismos para um *weaving* dinâmico. Por isso, *middlewares* dinamicamente adaptáveis e orientados a aspecto precisam construir mecanismos específicos para tal fim.

Os *middlewares* com adaptações em tempo de execução têm a reflexão computacional como a ferramenta mais importante, já que eles necessitam descobrir o estado atual do *middleware* antes de fazer qualquer reconexão entre componentes do sistema. A combinação da reflexão computacional com as outras técnicas parece ser a opção da maioria dos trabalhos quando se trata de *middlewares* adaptáveis dinamicamente. Na figura 2.13 mostramos uma tabela comparativa entre os *middlewares* estudados, de acordo com as categorias que apresentamos neste capítulo e de acordo com as informações encontradas nos artigos relacionados a esses *middlewares*.

Nos próximos capítulos, usaremos algumas das características estudadas, tais como a divisão do sistema em componentes e a separação de interesses nos níveis de protocolo e transporte do PEPt, entre outras.