Adaptação dinâmica no Open-ORB: detalhes de implementação

Nelio Cacho , Thais Batista

[1]Universidade Federal do Rio Grande do Norte (UFRN),
Departmento de Informatica,
Campus Universitário Lagoa Nova, 59072-970, Natal-RN, Brazil

cacho@consiste.dimap.ufrn.br, thais@ufrnet.br

Resumo. *Esse artigo apresenta estratégias para implementação de middlewares reflexivos, em particular para o Open-ORB, visando aumentar o suporte para adaptação dinâmica. Para isso são definidas abstrações de alto nível para se expressar a estrutura interna do middleware e aproveitados os mecanismos reflexivos definidos pelo Open-ORB que irão atuar sobre tal estrutura. O artigo detalha uma implementação que emprega as estratégias definidas e também apresenta a avaliação de desempenho dessa nova implementação, desenvolvida em Lua, comparando-a com a anterior, desenvolvida em Python.*

Abstract. *This paper presents implementation strategies to reflective middlewares, in particular to Open-ORB, in order to improve its support for dynamic adaptation. These strategies include the definition of high-level abstractions to express the middleware internal structure that are combined with the reflective mechanisms defined by Open-ORB. Such mechanisms act on the internal structure. The paper gives details about an implementation that uses the strategies defined in this paper. It also presents a performance evaluation of this new implementation, developed using Lua, comparing it with a previous one, developed using Python.*

1. Introdução

No contexto de desenvolvimento de aplicações distribuídas, plataformas de middleware [Bernstein, 1996] têm sido utilizadas como infra-estrutura subjacente oferecendo transparência de distribuição, de heterogeneidade e disponibilizando serviços comuns. Tipicamente, a estruturação das plataformas de middleware segue uma arquitetura monolítica agregando diversas características para atender a uma gama de aplicações. Tal arquitetura é inadequada face à diversidade de aplicações e de ambientes de hardware e de software com características distintas que exigem que o middleware ofereça serviços especializados determinados caso a caso. Portanto, o comportamento dinâmico é um requisito fundamental para o middleware de forma a permitir que protocolos e serviços sejam inseridos e removidos dinamicamente na sua arquitetura, conforme as necessidades das aplicações e do ambiente.

Para endereçar esse problema surgiu a idéia de *middleware de próxima geração* [Blair et al., 1998, Agha, 2002, Tripathi, 2002]. Segundo [Agha, 2002], existem dois aspectos fundamentais relacionados ao desenvolvimento de middleware de próxima geração: adaptação dinâmica e abstrações de alto nível. Adaptação dinâmica oferece a flexibilidade necessária para adaptar o middleware de acordo com os requisitos da aplicação enquanto que abstrações de alto nível simplificam o problema de expressar padrões complexos de interação entre os elementos que compõem o middleware.

O Open-ORB [Blair et al., 1998, Blair et al., 2001] foi um dos precursores dessa nova geração propondo uma arquitetura com estilo diferente das tradicionais arquiteturas *caixa-preta* que dificultam adaptação dinâmica. Para isso, o Open-ORB utiliza reflexão computacional [Smith, 1982] e define meta-modelos. A reflexão é usada para permitir a inspeção do comportamento interno do sistema e, conseqüentemente, para permitir a adaptação de acordo com as necessidades da aplicação. Uma das primeiras implementações desse middleware foi o Open-ORB Python Prototype(OOPP)[Andersen et al., 2000]. O OOPP concentra todo seu poder de adaptação na utilização da reflexão. No entanto, não segue amplamente a idéia de estruturar o middleware explorando abstrações que ampliem o poder de adaptação do mesmo. Para isso, o middleware reflexivo deve ser implementado como uma coleção de componentes [Szyperski, 2002] que possam ser configurados e reconfigurados pela aplicação [Kon et al., 2002]. Portanto, o modelo de componentes utilizado para estruturar internamente o middleware é fundamental para permitir sua adaptação assim como a estratégia de implementação de cada um dos elementos.

O OOPP apresenta várias limitações decorrentes de uma má estruturação interna: (1) a abordagem utilizada para a implementação dos *binding* locais limita o poder de reflexão do meta-modelo que expõe a composição interna do componente; (2) a inexistência de um conjunto único de classes para implementar o recebimento e tratamento das conexões implica na necessidade de um código muito especifico que poderia perfeitamente ser generalizado; (3) a definição de um limitado número de níveis de abstrações aproxima as camadas superiores do middleware das camadas de transporte de dados impedindo, por exemplo, a troca da camada de transporte.

No sentido de superar essas limitações e definir estratégias para implementação de middlewares reflexivos que facilitem a adaptação dinâmica, este trabalho propõe uma implementação do Open-ORB que amplia o suporte a adaptação fornecida pelo OOPP. Para isso são definidas abstrações de alto nível para se expressar a estrutura interna do middleware e aproveitados os mecanismos reflexivos definidos pelo Open-ORB que irão atuar sobre tal estrutura. Ou seja, é definido um conjunto mínimo de elementos com suporte a reflexão que, combinados, irão gerar um middleware reflexivo. Essa nova implementação do Open-ORB, chamada de LOpenORB, foi desenvolvida em Lua por possuir características semelhantes a linguagem Python, usada na implementação do OOPP: facilidades reflexivas, natureza interpretada e sistema de tipos dinâmico. A similaridade entre as linguagens facilita as comparações entre LOpenORB e OOPP tanto em termos estruturais quanto em relação ao desempenho. Nesse artigo a abordagem utilizada na construção do LOpenORB será detalhada e comparada com a utilizada no OOPP.

Esse artigo está estruturado da seguinte forma. A seção 2 apresenta alguns conceitos relativos a middlewares reflexivos e os princípios gerais da arquitetura do Open-ORB. A seção 3 apresenta a arquitetura do LOpenORB, contrastando com a do OOPP, bem como suas funcionalidades. A seção 4 descreve a infra-estrutura de comunicação. A seção 5 apresenta a avaliação de desempenho do LOpenORB comparada com a do OOPP. A seção 6 contém as conclusões.

2. Open-ORB

2.1. Middleware Reflexivos

O principal objetivo da reflexão em sistemas computacionais é permitir que um sistema possa executar algum processamento em benefício próprio, para modificar ou ajustar sua própria estrutura ou comportamento. Com base nessa idéia, a reflexão vem sendo aplicada com sucesso em áreas como linguagem de programação, sistemas operacionais e

em middleware. Neste último caso, a utilização da reflexão deu origem ao termo "*middleware reflexivo*". O objetivo de um middleware reflexivo é dispor de um maior poder de adaptação utilizando a reflexão computacional para esse propósito.

A reflexão está diretamente ligada aos detalhes de implementação de um sistema, uma vez que promove a separação entre as funcionalidades fornecidas pelos objetos, e os detalhes de sua implementação. Para isso o *middleware* é dividido em dois níveis: nível-base e meta-nível. No *nível-base* estão as funcionalidades do sistema, disponíveis através das interfaces bases. No *meta-nível* estão os meta-objetos usados para descrever e manipular os objetos do nível-base. No meta-nível existem ainda duas abordagens: *reflexão estrutural* e *comportamental*. A *reflexão estrutural* permite a inspeção e a manipulação da estrutura do middleware, além de expor detalhes de composição dos elementos internos. Na *reflexão comportamental*, o comportamento das funções básicas do middleware pode ser monitorado e controlado no meta-nível. Dessa forma, operações como a manipulação de variáveis (leitura e escrita) e invocações de métodos podem ser interceptadas e desviadas para o meta-nível.

Middlewares construídos a partir dessa abordagem utilizam-se de módulos compostos por *objetos base* e por *meta-objetos*. O meta-objeto possui uma interface com o objeto base, através da qual pode-se obter informações internas do objeto base ou interceptar chamadas que deveriam ir diretamente para o mesmo. Uma invocação interceptada pode ser manipulada pelo meta-objeto e ser reencaminhada para o objeto original.

2.2. Arquitetura do Open-ORB

Diferentemente dos middlewares tradicionais, onde a arquitetura segue um estilo caixa-preta, o Open-ORB propõe uma arquitetura aberta baseada em reflexão computacional. Nas subseções seguintes descreveremos seus principais elementos.

2.2.1. Elementos do nível-base

Os principais elementos do nível-base da arquitetura Open-ORB são: Interfaces, *Bindings* locais e Componentes. *Interfaces* representam os pontos de acesso do componente. Cada interface pode exportar ou importar métodos. Os métodos exportados são os disponibilizados pelos objetos e os importados são os requisitados pelos objetos. Para estabelecer a relação entre as interfaces que exportam e as que importam é utilizado um *binding* local. Este *binding* associa interfaces compatíveis, ou seja, um método importado por uma interface deve ter seu correspondente na interface exportadora. A figura 3 ilustra a presença de vários *bindings* locais que realizam a conexão entre componentes. Um desses *bindings*, por exemplo, permite a utilização por parte do componente *Capsule*, dos métodos *server* e *newPort* fornecidos pelo componente *NodeMngr*.

Na arquitetura Open-ORB um componente é uma unidade de composição e distribuição independente [Szyperski, 2002] que especifica serviços requisitados e fornecidos através de uma ou mais interfaces. Todas as interações entre componentes são realizadas através de suas interfaces. Este é um fator que facilita o desenvolvimento, além de permitir uma maior independência. Como um componente é uma unidade de composição, existem dois tipos de componentes, o *primitivo* e o *composto*. O componente primitivo encapsula apenas um objeto e fornece acesso a uma ou mais interfaces. O componente composto contém um conjunto de outros componentes interligados, através de *bindings* locais, por suas interfaces. Cada um desses componentes que compõe um componente composto também pode ser um componente composto. Esta recursão de composição é finalizada por componentes primitivos. A figura 3 ilustra a presença de três componentes compostos, sendo o primeiro deles formado pelos componentes *Caspule*, *NodeMngr*

e *Accept Factory*. Pode-se observar que o conjunto de interfaces externas de um componente composto é um sub-conjunto das interfaces dos componentes que o compõem. Os componentes que compõem um componente composto não necessariamente devem estar no mesmo espaço de endereçamento (*Capsule*). Podem existir componentes distribuídos que, através de *bindings* implícitos/explícitos [Fitzpatrick et al., 1998], fornecem a composição de componentes em diferentes *Capsules*. Um *binding* implícito é um *binding* criado entre duas interfaces sem o conhecimento do programador. Este tipo de *binding* é normalmente usado quando uma interface é importada. O *binding* explícito, por sua vez, é criado pelo programador e pode ser de três diferentes tipos: *Operational, Signal, Stream*.

Um *binding Operational* é usado para enviar invocações de métodos de interfaces importadoras para interfaces exportadoras e, opcionalmente, retornar um resultado. Um *Signal binding* suporta um conjunto de sinais *one-way* onde não se obtém resposta. O fluxo de um sinal origina-se no *Source* e destina-se ao *Sink*. O *Stream binding* opera semelhantemente ao *Signal*, diferenciando-se apenas no suporte ao envio de dados contínuos como áudio e vídeo.

2.2.2. Elementos do meta-nível

Como foi visto na seção 2, a reflexão é composta por aspectos estruturais e comportamentais. Para simplificar as interfaces providas por cada um desses aspectos, a arquitetura Open-ORB dividiu o meta-nível em quatro diferentes meta-modelos [Okamura et al., 1992]. Os aspectos estruturais são representados pelos meta-modelos *encapsulation* e *composition* e os aspectos comportamentais são representados pelos meta-modelos *environment* e *resource*. Cada meta modelo apresenta as seguintes funcionalidades:

- *Encapsulation*: O objetivo deste meta-modelo é expor o encapsulamento provido pelos objetos, tornando possível a inspeção, modificação e extensão da implementação de um objeto. Este meta-modelo pode ser usado para monitorar e controlar todos os acessos aos objetos, incluindo os seus atributos e métodos.
- *Composition*: Este meta-modelo provê acesso ao grafo de *bindings* de um componente. Através desse meta-modelo é possível remover, inserir e substituir componentes que façam parte de um grafo de componentes.
- *Environment*: Este meta-modelo expõe o ambiente de execução de cada interface. Isto inclui todas as etapas do processo de invocação de um método, tanto o lado cliente (*marshall*, envio) como o lado servidor (*unmarshall*, processamento).
- *Resource*: O objetivo deste meta-modelo é expor a alocação e gerenciar os recursos associados com cada um dos objetos ou interfaces.

3. LOpenORB

3.1. Arquitetura

A figura 1 ilustra os elementos que compõem a arquitetura de implementação do LOpenORB. Nela estão descritas as abstrações de alto nível definidas para facilitar o desenvolvimento e a adaptação de um middleware reflexivo. A primeira delas é a classe *Interface*. Ela representa o ponto de acesso para os objetos e componentes. Uma interface possui vários métodos importados e exportados representados pela classe *Methods*. Os métodos exportados são implementados pelo elemento associado ao atributo *Object*. Cada método, representado pela classe *Methods*, possui como atributo o nome do método, seu tipo (importado ou exportado) e um *Binding* usado para invocar um método relacionado a um

respectivo *BindCtrl*. A classe *BindCtrl* é um elemento chave na vinculação de interfaces e na composição de componentes. Ela é criada pelo método *localBind* da classe *LocalBinding*. Um *BindCtrl* possui um *BindingGraphElement* que estabelece um vínculo entre os *node_a* e *node_b*. Cada nó é uma instância da classe *GraphNode* que possui como atributos uma referência da classe *Interface*, o nome da interface e uma referência da classe *Component*.

A composição das Interfaces é realizada pela classe *Component* e *CompositeComponent*. A classe *Component* representa um componente primitivo e armazena o nome do componente e as interfaces externas fornecidas pelo componente. A classe *CompositeComponent* representa um componente composto formado pelos componentes armazenados no atributo *components*.

A classe *MetaMethods* é criada quando uma interface possui um meta-objeto associado. Cada método da interface possui uma instância de *MetaMethods*. Essa classe registra os métodos *Pre, Pos e Wrap*, que serão executados respectivamente antes, depois e durante a execução do método original. Além disso, possui a propriedade *Factory* responsável por determinar uma lista de possíveis *Factories* que podem ser usados para construir o caminho de invocação de cada método. Cada *Factory* é representado pela classe *Factories* que possui o nome do *Factory* e um atributo *method*, da classe *Method*, usado para invocar o construtor.

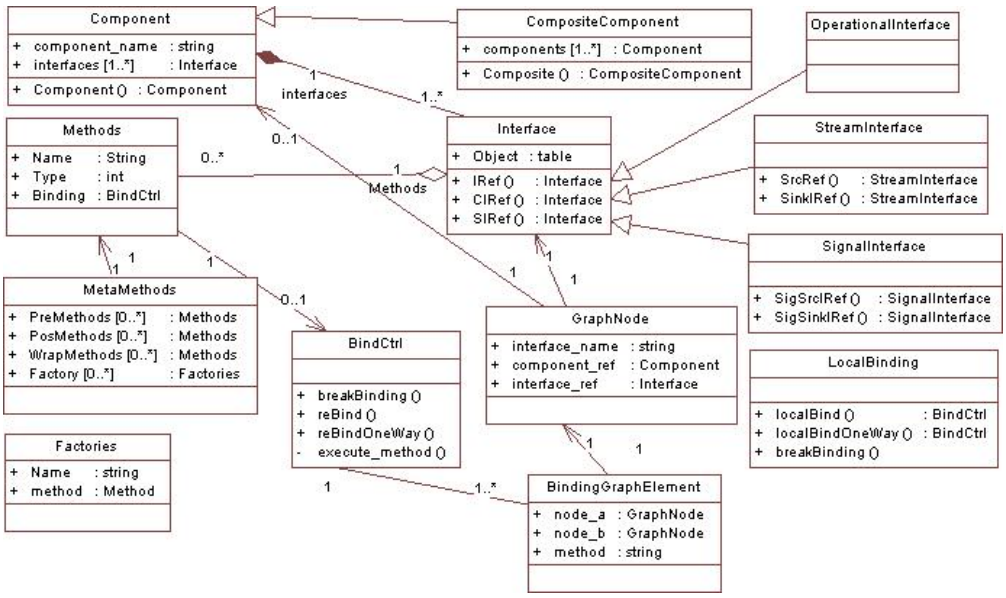

Figura 1: Arquitetura do LOpenORB

De forma geral, a diferença entre as arquiteturas do LOpenORB e do OOPP, está no papel dos *bindings* locais. No OOPP, quando um *binding* local é criado, a interface importadora relaciona-se diretamente, através das classes *IObj e IMethod*, com o objeto que implementa a interface exportadora, e não com a própria interface. Dessa forma, não é possível, através da interface importadora, chegar até a exportadora. Essa abordagem limita o poder de representação dos grafos de nós a apenas os componentes compostos, que o fazem por implementação interna. No LOpenORB, todos os componentes e interfaces são vinculados através dos *BindCtrl*, ou seja, uma interface importadora possui uma referência do *BindCtrl* que a vincula a interface exportadora. Dessa forma, quando um método é invocado em uma interface importadora, esse método é encaminhado através da função *execute_method* para o *BindCtrl* correspondente. Na classe *BindCtrl*, o *BindingGraphElement* é recuperado e verifica a existência de um meta-objeto para aquela interface. Caso exista, os métodos *Pre, Pos, Wrap* e *Factory* são invocados. Os *Factories*

são um tipo especial de Pre-métodos, adicionados pela abordagem do LOpenORB que, ao invés de receber os parâmetros da função invocada, sempre recebe uma referência do *BindCtrl* utilizado. Dessa forma, é possível, dentro das funcionalidades do *Factory*, modificar os elementos que compõem o *BindingGraphElement* e assim alterar o caminho de execução da função. Podem existir vários *Factories* para um mesmo método, no caso do *BindCtrl*, é utilizado o *Factory* com nome *FactoryInvocation*.

3.2. Funcionalidades Básicas

As funcionalidades básicas do LOpenORB são as mesmas do OOPP: criação de interfaces, *bindings* locais, Componentes e *Composites*. O LOpenORB também tentou manter a mesma sintaxe de invocação do OOPP. Dessa forma, funções como *IRef*, *CIRef*, *SIRef*, responsáveis por criar as interfaces, mantiveram os mesmos argumentos dos utilizados no OOPP. O mesmo não ocorreu com as funções que criam *bindings* locais, componentes e componentes compostos.

A função *localBind*, responsável por criar os *bindings* locais, recebe apenas duas interfaces na abordagem OOPP. No caso do LOpenORB, essa função mantém o número de parâmetros, diferenciando-se pelo tipo informado. São passados dois elementos do tipo *GraphNode*, onde cada um deve ter a referência da interface, o nome da interface e a referência do componente que fornece essa interface. Os dois últimos parâmetros são opcionais e buscam ampliar o alcance do grafo de componentes, pois ao atingir uma interface, se a mesma fizer parte de um componente, as demais interfaces desse componente também poderão ser incluídas no grafo.

```
1 local objLoja = {}
2 function objLoja:vender(produto) . . .  end
3 objLoja.InterfaceLoja = Interface:IRef(objLoja, {"vender"}, {})
4 local InterfaceVendedor = Interface:IRef({}, {}, {"vender"})
5 LocalBinding:localBind({objLoja.InterfaceLoja},{InterfaceVendedor})
6 InterfaceVendedor:vender("145214")
```

Figura 2: Criando um binding local entre duas interfaces

A figura 2 ilustra a utilização de um *binding* local para vincular as interfaces Loja e Vendedor. Nas linhas 3 e 4, as interfaces exportadora e importadora são criadas. Na linha 5, o *binding* local é criado. O método *vender* da interface importadora é invocado na linha 6.

Assim como no OOPP, um componente pode ser criado de duas maneiras. A primeira é através do método *Component(name, interfaces, objeto)*, onde são necessários o nome do componente, suas interfaces externas e o objeto do componente. Pode-se, por exemplo, criar um componente Loja, através da invocação *Component("Loja",{{"DepVendas", objLoja.InterfaceLoja}},objLoja)*. A segunda maneira é através do método *Composite*, que cria um componente composto por vários outros componentes. Este método recebe como parâmetro o nome do novo componente composto, as interfaces externas, os componentes que irão compor o componente composto, as interfaces internas e uma lista com a combinação das interfaces internas que serão vinculadas através de *bindings* locais.

3.3. Funcionalidades do Meta-nível

Assim como o OOPP, o LOpenORB fornece dois meta-modelos: *encapsulation* e *composition*. No LOpenORB estes meta-modelos são fornecidos por um componente composto chamado *MetaModelComponent*. Este componente possui duas interfaces externas: *EncapsulationMetaModel* e *CompositionMetaModel*. A primeira interface exporta o método

encapsulation que recebe como parâmetro a referência de uma interface ou componente e retorna um meta-objeto *encapsulation* para o respectivo elemento. A segunda interface exporta o método *composition* que recebe como parâmetro a referência de uma interface ou componente e retorna um meta-objeto *composition* para o respectivo elemento.

Os métodos fornecidos pelo meta-objeto *encapsulation* do LOpenORB são os mesmos fornecidos pelo OOPP. Como exemplo, *inspect()* retorna a descrição de uma interface ou componente; *addPreMethod(m, f)* adiciona o método *f*, da classe *Method*, para ser executado antes do método de nome *m*. Existe ainda os fornecidos para o meta-objeto *encapsulation* de um componente, os principais são: *inspect()* retorna descrição do componente; *addIF(name, impl)* adiciona ou substitui a interface com nome *name* pela implementação *impl*; *delIF(name)* remove a interface com nome *name* do componente e *changeObject(obj)* substitui o objeto conteiner do componente por *obj*.

Além desses métodos, a abordagem LOpenORB fornece ainda: *addFactory-Method(m,fname, f), delFactoryMethod(m,fname)* e *getFactoryMethod(m,fname)* que respectivamente adiciona um *Factory* com nome *fname* para o método *m* e que será executado pelo método *f*; remove um *Factory* com nome *fname* do método *m* e obtém um *Factory* com nome *fname* do método *m*. O *Binding* de um método também pode ser definido através da função *setBindingMethod(m,b)* ou pode ser obtido através *getBinding-Method(m)*. O primeiro recebe o nome do método *m* e uma instância da classe *BindCtrl b* e o segundo apenas o nome do método.

Os mesmos métodos disponibilizados pelo meta-objeto *composition* do OOPP são fornecidos pelo LOpenORB, entre eles: *inspect()* retorna o grafo de componentes; *insert(comp, bind)* insere um componente *comp* na grafo de componentes definido por *bind*; *replace(name,comp)* substitui o componente com nome *name* pelo componente *comp*, refazendo todos os *bindings*.

Como foi mostrado na seção 3.1, o LOpenORB possui uma abordagem diferente da utilizada no OOPP para a utilização dos *bindings* locais. Por isso, o método *composition* pode ser aplicado a uma interface, componente ou componente composto. Isto difere da abordagem empregada pelo OOPP, onde esse método só pode ser aplicado a um componente composto. Essa diferença amplia em muito o poder de representação dos grafos de componentes, que deixam de estar restritos aos componentes compostos e passam a representar quase a totalidade dos vínculos (*bindings*) existentes numa aplicação. Isso pode ser visto na própria implementação da infra-estrutura de comunicação(figura 3) do LOpenORB, que será discutida na próxima seção, onde existem componentes compostos ligados a outros componentes através de *bindings* locais constituindo um caminho que pode ser percorrido pelo meta-objeto *composition*. Dessa forma, partindo do objeto *Capsule Local* é possivel alcançar os componentes *Capsule*, *NodeMngr*, *Accept Factory*, *Accept*, *Thread Mngr*, *Transport Wrapper*, *Dispatcher Factory*, *Dipatcher*, *EndPoint*, *Protocol* e por fim a interface que exporta o método *vender*.

4. Utilizando abstrações de alto nível na construção da infra-estrutura de comunicação

Nas seções anteriores tratamos dos elementos que constituem a arquitetura do LOpenORB sem no entanto citar algo relacionado a infra-estrutura de comunicação. Parece estranho, portanto, que um ORB não possua, em sua arquitetura, elementos de comunicação, uma vez que o seu principal objetivo é prover uma camada de comunicação entre componentes distribuídos. Esse foi portanto o nosso maior objetivo: construir um ORB a partir de um limitado conjunto de elementos básicos que oferecam suporte a reflexão para que

o produto gerado, no caso o ORB, também fornecesse suporte a reflexão e uma alta capacidade de adaptação. Esta capacidade de adaptação é obtida uma vez que os próprios elementos da infra-estrutura de comunicação são componentes e componentes compostos ligados através de suas interfaces e *bindings* locais onde os meta-objetos (*encapsulation* e *composition*) podem atuar na reflexão e adaptação de forma transparente e flexível.

Os primeiros componentes criados, seguindo essa idéia, foram *Capsule* e *NodeMngr*. Estes componentes são responsáveis pelas funcionalidades básicas da infra-estrutura de comunicação. O componente *Capsule* é responsável por gerenciar o ambiente local provendo para isso a interface *provide_capsule* que disponibiliza funções como: *registerComponent*, *getIdComp* e *getIRef*, etc. Todos os componentes que podem ser acessados remotamente devem ser registrados no componente *Capsule*. Esse registro é feito através do método *registerComponent(compname, CompRef)* que recebe o nome e a referência do componente, retornando seu *id* de registro. Para obter o *id* de um componente pode-se usar o método *getIdComp(name)*. Para obter a interface de um componente registrado utiliza-se *getIRef(idcomp, ifacename)*, que recebe o *id* do componente e o nome da interface externa e retorna a referência da interface correspondente. Como um *Capsule* pode ser acessado remotamente, ele próprio se registra com *id* igual a 1. Isso permite que para acessar uma funcionalidade do componente *Capsule* é necessário apenas invocar *getIRef(1,"provide_capsule")*.

O componente *NodeMngr* gerencia as portas do ORB através da interface *provide_nodemngr* com os métodos *newPort*, *serve*, *servePort*, etc. O método *newPort(complist)* recebe como parâmetro uma lista de componentes que poderão receber invocações em uma porta. Em seguida, são utilizadas as funções *inspect* e *getFactoryMethod*, do meta-objeto *encapsulation*, para separar os componentes que possuem um *Factory* com nome *RemoteListen*. Esse *Factory* é padronizado para utilização do componente *NodeMngr* e a sua presença determina os componentes e *bindings* que serão criados e combinados para tratar e processar as mensagens recebidas (figura 3). Na ausência desse *Factory* utiliza-se o *Accept Factory* vinculado ao *NodeMngr*. O resultado da função *newPort* é uma lista que relaciona os componentes fornecidos com as portas criadas. O método *getPort(portNum)* pode ser usado para obter uma ou mais portas de um *NodeMngr*. Para isso, este método recebe o número da porta e retorna o componente *Accept* correspondente, ou recebe "*"e retorna uma lista de componentes *Accept*.

Nas próximas duas sub-seções, descreveremos como invocações podem ser tratadas e enviadas pela infra-estrutura de comunicação utilizando-se apenas das abstrações de alto nível definidas na seção 3.1.

4.1. Tratando invocações

O processo de tratamento de invoções é iniciado pela chamada do método *serve* da interface *provide_capsule*. Este método usa a interface importadora *use_nodemngr*, do componente *Capsule*, para invocar os métodos *newPort* e *serve*, que criam e verificam a lista de portas criadas com sua respectiva lista de *Factories*. Para as portas que possuem *Factory* definido, esse *Factory* é invocado. Para as que não possuem, o método *create* do *Accept Factory* é invocado.

O método *create* cria um componente composto formado pelos componentes *Accept*, *Transport Wrapper*, *Dispatcher Factory* e *Thread Mngr*. Na seqüência, esse método cria um *binding* local entre o componente criado e a respectiva porta do *NodeMngr*. O último passo é invocar o método *listen* do componente *Accept*.

O método *listen* invoca o método *createListen*, do componente *Transport Wrapper*, que fica bloqueado esperando uma nova conexão. Quando uma conexão é estabe-

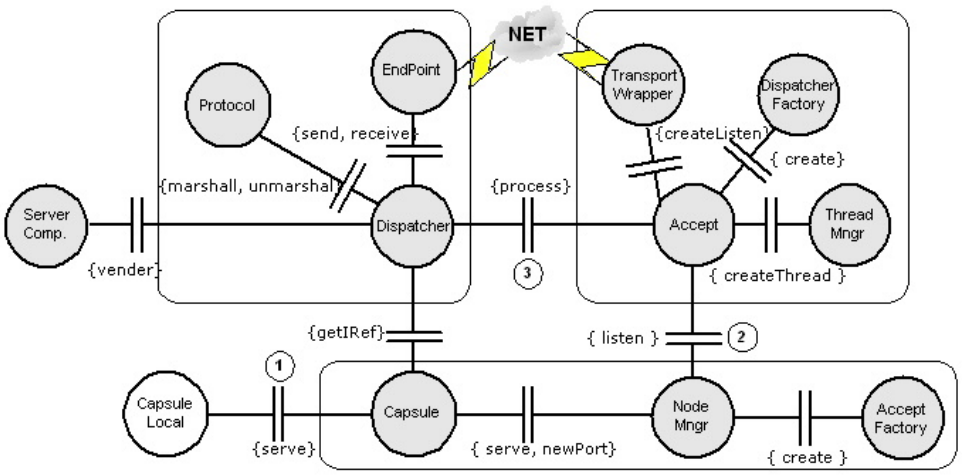

Figura 3: Caminho de execução do servidor

lecida, o método *create* do componente *Dispatcher Factory* é invocado, retornando um componente composto pelos componentes *Dispatcher* e *Protocol*. O próximo passo é estabelecer dois *bindings* locais: o primeiro entre este componente e o componente *EndPoint*, retornado pelo método *createListen*, e o segundo entre os componentes *Dispatcher* e *Capsule*. O último passo é invocar o método *createThread* do componente *ThreadMngr* para processar em uma *Thread* separada a invocação do método *process* do componente *Dispatcher*.

Para tratar cada conexão, *process* invoca o método *receive* do componente *EndPoint* que obtém a mensagem. Esta é tratada através do método *unmarshal*, do componente *Protocol* e através do método *getIRef* do componente *Capsule*, a interface que irá executar o método é localizada. Para invocar o método desejado, é necessário criar um *binding* local entre essa interface e o componente *Dispatcher*. Através desse *binding* pode-se invocar o método desejado, receber seus parâmetros de retorno, fazer o *marshall* da mensagem e enviar a resposta através do método *send*, finalizando o caminho de execução do servidor.

Na abordagem do OOPP não existe a separação dos papéis entre componentes e interfaces como é feito no LOpenORB. Dessa forma, as funcionalidades se misturam, produzindo um ambiente de difícil adaptação. Por exemplo, os componentes *Accept*, *Transport Wrapper* e *EndPoint* da abordagem LOpenORB são implementados pela classe *Msg* e *RecvMsg* no OOPP. Isso é feito através dos métodos *sendreq*, *recvreq*, *sendrep* e *recvrep*, sendo os dois primeiros para enviar e receber requisições e os dois últimas para enviar e receber respostas. As funcionalidades do componente *Protocol* são implementadas no OOPP pelas bibliotecas do Python *marshal*, no envio de *Streams*, e *cPickle* no envio dos demais tipos de invocações (Op. e Signal). As funcionalidades do componente *dispatcher* são implementadas no OOPP por cinco classes: *NodeMngr, NameServe*, *Capsule*, *Stub* e *SinkStub*. Cada uma dessas classes implementa um *loop* para tratar suas próprias operações remotas. Dessa forma, *NameServe* possui um *loop*, no método *__init__* que trata apenas das operações remotas disponibilizadas por esta classe, como *exportIRef, lookupIRef*, etc. Da mesma forma, as classes *Stub* e *SinkStub* possuem, no método *__serveloop__*, um loop para tratar as operações disponibilizadas pelas *Operational* interfaces e *Streams* interfaces.

A abordagem utilizada por OOPP possui sérias limitações de adaptação uma vez que a troca de um dos componentes da infra-estrutura de comunicação é algo bastante difícil. O mesmo não ocorre com o LOpenORB que para, por exemplo, substituir o

componente *Protocol* de uma determinada conexão é necessário apenas a invocação do método *replace("dispatcherProtocol",NewProtocol)*, do meta-objeto *composition*.

4.2. Realizando invocações

Descrevemos anteriormente como o LOpenORB implementa o mecanismo de recebimento e tratamento de invocações remotas. Neste momento detalharemos como as invocações são enviadas pelos clientes, utilizando para isso a invocação do método *vender*, pertencente a interface *DepVendas* do componente com *id* igual a 5.

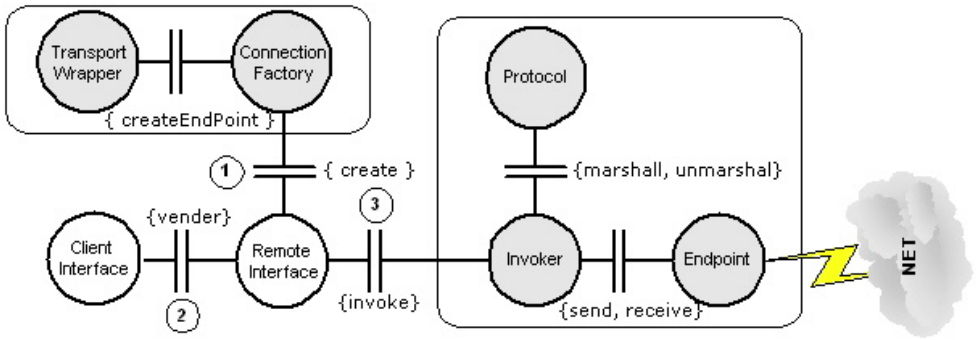

Figura 4: Caminho de invocação do cliente

A figura 4 ilustra os componentes envolvidos na construção do caminho de execução do lado cliente. O primeiro passo é obter uma interface local que representa a interface remota. Para isso a interface *provide_capsule* fornece o método *getRemoteInterface("localhost", 8080, "CompVendas", "DepVendas")*. Este método recebe o nome do host, a porta, o nome do componente e o nome da interface. Em seguida, retorna uma interface local que exporta os métodos da interface remota. Internamente o que esse método faz é criar uma interface que atua como *proxy* genérico para os métodos exportados por *DepVendas* e criar um componente composto formado pelos componentes *Transport Wrapper* e *Connection Factory*. Na seqüência, o método cria um *binding* local entre o componente composto e a interface criada(*Remote Interface*). O *BindCtrl*, do binding local criado, é associado através da função *addFactoryMethod("vender", "FactoryInvocation",{BindCtrl,"create"})* do meta-objeto *encapsulation*.

Para invocar o método remoto a interface cliente cria um *binding* local entre sua interface e a interface remota. Quando o cliente invoca o método *vender*, o *BindCtrl* verifica que o método tem um *Factory* com nome *FactoryInvocation*. Neste momento, o método *create* do componente composto formado por *Connection Factory* é invocado, criando um componente composto formado pelos componentes *EndPoint*, *Protocol* e *Invoker*. O passo seguinte envolve a criação de um *binding* local entre esse componente e a interface remota, modificando através da função *setBindingMethod("vender",BindCtrl)*, do meta-objeto *encapsulation*, o *binding* do método *vender*. Por fim, este método invoca a função *delFactoryMethod("vender","FactoryInvocation")* para evitar que o caminho seja recriado para cada invocação. Dessa forma, quando a interface cliente invoca o método *vender*, esse método é repassado para a interface remota, que por sua vez repassa para o componente *Invoker*. Este componente realiza o *marshall* da mensagem, através do componente *Protocol*, e a envia através do componente *EndPoint*. Dependendo do componente *Invoker* utilizado, o cliente pode ficar ou não bloqueado no método *receive* esperando a resposta do servidor. No caso de uma resposta, o componente *Invoker* invoca o método *unmarshal* para tratar a mensagem e retorna para a interface cliente o resultado.

Na abordagem do OOPP, para invocar um método aleatório, como foi feito acima, é necessário criar uma instância da classe *CapsuleProxy(host,port)* e, em seguida, invocar o método *callMethod(componente, interface, methodname, args)*. O mesmo ocorre

quando se deseja invocar métodos das classes *NodeMngr* e *NameServe*, onde deve-se criar instâncias das classes *NodeMngrProxy* e *NameServerProxy*. Para invocar um método, estes *proxies* utilizam-se diretamente do método *message* da classe *Msg* que, por sua vez, usa os métodos *sendreq* e *recvreq*, também dessa mesma classe. Esta abordagem além de ter as mesmas limitações de adaptação presentes no processo de recebimento e tratamento de invocações, também é bastante inflexível, uma vez que depende da construção de *proxies* específicos.

4.3. Utilizando a infra-estrutura de comunicação do LOpenORB

A discussão anterior tratava da definição dos mecanismos de tratamento e invocação de requisições remotas. Neste momento, discutiremos a utilização desses mecanismo para a construção de um *Operational binding*.

Um *Operational binding (Op. binding)* é um componente composto e distribuído formado por quatro interfaces: *iface1*,*iface2*, *ctrl1* e *ctrl2*. As interfaces *iface1* e *ctrl1* disponibilizam a interface do componente remoto e sua respectiva interface de controle. O mesmo ocorre para *iface2* e *ctrl2* que disponibilizam, respectivamente, a interface local e a interface de controle local. Dessa forma, se for necessário invocar métodos da interface remota, deve-se criar um *binding* local com a interface *iface1*. Mas, se for preciso parar a interface local, deve-se criar um *binding* local com *ctrl2*. No LOpenORB, o *Op. binding*, como os demais *bindings* explícitos (*Signal* e *Stream*), são implementados através de um componente. Dessa forma, um *Op. binding* é criado por um componente com apenas uma interface *provide_opbind* que é registrada no *Capsule* para receber invocações remotas. Esta interface possui dois métodos: *create* e *createRemote*. O método *create* é invocado pelo método *remoteBind* para criar um *Operational binding*. O método *remoteBind(iref1, iref2)*, recebe duas interfaces: *iref1* que pode ser uma interface local como *IRef(obja, {"vender","cancelarPedido"},{"entregar"})* e *iref2* que pode ser uma interface remota como *IRef({}, {"entregar"}, {"cancelarPedido"})*. O processo de construção de um *Op. binding* resume-se a criação de dois componentes, um local e outro remoto, tendo cada um quatro interfaces. O método *create* da interface *provide_opbind*, inicialmente cria o componente *compLocal* com apenas a interface local informada, *iref1*. Esse componente é registrado no *Capsule* e através do Id desse componente é criado um outro componente para controlar a interface local. De posse das duas interfaces locais, *iface2* e *ctrl2* o método *create* invoca o *createRemote*, ao host que disponibiliza a interface remota *iref2*. São passados para esse método a interface *iface2*, *ctrl2* e a interface remota *iref2*. No host remoto, o método *createRemote* cria um componente *compFunc* com as interfaces *iface1*, *iface2* e *ctrl1* recebidas como parâmetro e registra o novo componente no *Capsule*. A partir do id do novo componente, um componente *compCtrl*, com interface *ctrl1* é criado e composto, através do meta-objeto *composition*, com o componente *compFunc*. Por fim, a função retorna uma lista com as interfaces *iface1* e *ctrl1*. De volta ao *host* que iniciou o processo de criação do *Op. binding*, o método *create* recebe os parâmetros de retorno do método *createRemote* e também realiza a composição do *compLocal* com as interfaces fornecidas, finalizando dessa forma o processo de criação de um *Operational binding*.

No OOPP não existe a idéia de um componente remoto com a finalidade de criar um *Operational binding*. Todo processo é feito através do *Capsule* local e de um *CapsuleProxy*. Dessa forma são criadas as portas, locais e remotas, e, em seguida, através da classe Stub e das portas criadas, são criados os *stubs* local e remoto. Por fim é criado o componente que representa o *Operational* binding. Esta abordagem apresenta problemas de desempenho que serão comentadas na próxima seção.

5. Comparação e avaliação de desempenho

Esta seção visa analisar o desempenho do LOpenORB comparado ao OOPP. Para isso, os teste são divididos em duas partes. A primeira parte compara apenas os elementos básicos(*bindings* locais, componentes, etc) e a segunda compara a infra-estrutura de comunicação. Todos os experimentos foram conduzidos em um PC Duron 1.6MHz com 256MB de RAM, executando Linux-Mandrake 10.0. O LOpenORB foi testado utilizando Lua 5.0, e o OOPP utilizando Python 2.3. Todos os testes foram realizados com o mesmo conjunto de elementos, ou seja, tudo que foi implementado em Lua (objetos, métodos exportados, componentes, etc) também foi igualmente implementado em Python.

O primeiro teste, dos elementos básicos, compara o tempo de criação de uma Interface através da invocação do método IRef. A abordagem OOPP obteve um tempo de 56.98μs, enquanto a abordagem LOpenORB obteve 71.04μs. Essa diferença de 14μs deve-se ao fato de que a abordagem LOpenORB cria uma instância da classe *Methods* para todos os métodos importados e exportados, enquanto que o OOPP cria uma instância da classe *IMethod* apenas para os métodos exportados.

O segundo teste, da primeira parte, trata do tempo para criação de um componente. Nele, OOPP obteve 13.11μs, enquanto no LOpenORB, o tempo foi de 10.96μs. A diferença é ainda menor para a criação de um componente composto, onde são obtidos os tempos de 76.15μs e 77.96μs, respectivamente, para o OOPP e o LOpenORB. Essa pequena diferença entre os tempos, mostra a semelhança no processo de criação dos componentes e componentes compostos nas duas abordagens.

No teste relacionado ao tempo de criação de um *binding* local, a abordagem LOpenORB, mesmo tendo que criar um maior número de objetos, sendo eles 1 instância de *BindCtrl*, 1 instância de *BindingGraphElement* para cada método importado e duas instâncias de *GraphNode* para cada método importado, ainda conseguiu um tempo de 25.03μs, ficando abaixo de 30.94μs da abordagem do OOPP, que cria apenas uma instância de *LBindCtrl* e uma instância de *IMethod* para cada método importado. A diferença está no fato de que para cada método importado é associado uma instância diferente de *IMethod*, enquanto que no LOpenORB, todos os métodos importados recebem sempre a mesma instância de *BindCtrl*.

No último teste das funcionalidades básicas foi verificado o tempo de execução, de um mesmo método, através de um *binding* local. Para ter uma noção do tempo associado a cada linguagem, o mesmo método, implementado de forma igual nas duas linguagens, foi executado. Em Python, o método obteve um tempo de 47.92μs, enquanto que em Lua o tempo foi de 52.88μs. Quando invocado por uma interface importadora vinculada a uma interface exportadora por um *binding* local, os tempos subiram para 195.75μs no OOPP e 90.12μs no LOpenORB. Ou seja, a presença de um *binding* local reduziu a velocidade de invocação no LOpenORB em 37.24μs, enquanto que no OOPP a perda foi de 147.93μs.

O resultado da segunda parte dos teste pode ser visto no gráfico da figura 5. Neste gráfico são comparados os tempos de criação de *Operational bindings* nas abordagens OOPP e LOpenORB. Para um *Op. binding* a abordagem OOPP consumiu 40.78ms, enquanto que o LOpenORB consumiu 24.48ms. Comparando os valores obtidos quando são atingidos 20 *Op. bindings*, onde OOPP obteve um tempo de 790.21ms e o LOpenORB um tempo de 281.57, temos que, dividindo esses valores por 20, o tempo de criação de um *Op. binding* no LOpenORB, caiu de 24.48ms para 14.07ms, enquanto que o OOPP, com 39.51ms, manteve-se com praticamente o mesmo tempo de criação.

Esta grande diferença de tempo deve-se, principalmente, ao número de invocações remotas, executadas através da classe *CapsuleProxy*, necessárias para criar um *Op. bin-*

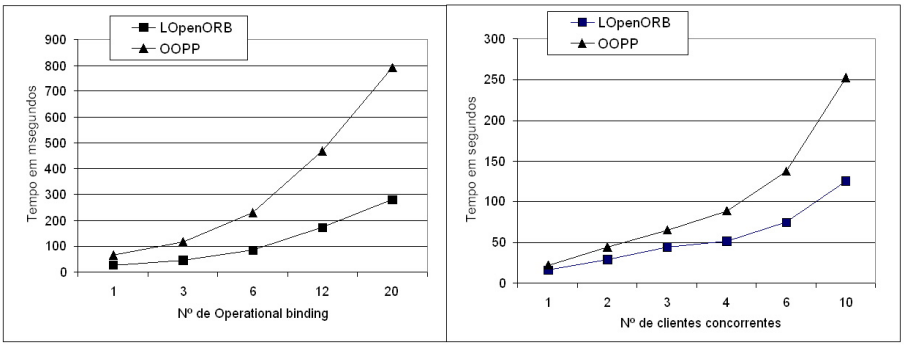

Figura 5: Gráfico de criação e execução de Op. *binding*

ding no OOPP. Nesta abordagem, são necessárias pelo menos quatro invocações remotas, enquanto que no LOpenORB, é necessária apenas uma. O tempo de uma invocação remota também foi alvo de análise. No OOPP, uma invocação remota consome 2.52ms enquanto que no LOpenORB, a primeira invocação consome 3.69ms e da segunda em diante, esse tempo cai e permanece constante em 1.31ms. Esta diferença entre a primeira e as subseqüentes invocações do LOpenORB é gerado pela tempo inicial necessário para criar o componente composto responsável por tratar uma conexão. O gráfico direito da figura 5 ilustra o tempo consumido em segundos pelas abordagens para realizar 10.000 invocações de um mesmo método remoto, aumentando-se progressivamente o número de clientes concorrentes. Esse gráfico comprova que mesmo tendo um tempo maior para a primeira invocação, o LOpenORB, consegue reduzir esse tempo e mantê-lo estável pelas subseqüentes invocações.

6. Conclusões

Nesse artigo apresentamos aspectos de implementação de middlewares reflexivos, em particular do Open-ORB, um dos precursores em termos de *middleware de próxima geração*. As decisões relacionadas a implementação do middleware reflexivo são fundamentais para que a arquitetura possa ser amplamente aproveitada e fornecer uma alta capacidade de adaptação dinâmica para as aplicações. A descrição do LOpenORB mostrou que, além de fornecer abstrações da alto nível com suporte a reflexão, como interfaces, componentes e *bindings* locais, é possível e necessário também aplicar essas abstrações na construção dos mecanismos internos do ORB, como na infra-estrutura de comunicação e na construção dos *bindings* explícitos.

Na comparação com OOPP, a abordagem de implementação dos *bindings* locais fornecida pelo LOpenORB mostrou-se mais eficiente tanto em termos de desempenho, como no suporte a reflexão, uma vez que toda a composição interna do ORB e da aplicação podem ser inspecionadas e modificadas. A utilização de *Factories*, na pré-invocação de métodos, mostrou-se capaz de criar os mecanismos de envio e tratamento de invocações permitindo assim uma escolha dinâmica do tipo de API de comunicação (Sockets, TLI, etc), do protocolo (GIOP, XML, etc), dos *dispatchers* e *invokers*. Esse aspecto é interessante para diversas classes de aplicações, por exemplo, aplicações multimídia onde protocolos diferentes precisam ser usados de acordo com o tipo de mídia sendo transmitida.

A avaliação de desempenho mostrou que a combinação de interfaces, componentes e *bindings* locais, seguida por uma clara definição do papel de cada componente que realiza essas abstrações, pode gerar uma solução eficiente e adaptável. Isto fica claro na comparação das abordagens de criação de e execução de um Op. *binding*, que mesmo

utilizando uma abordagem mais genérica, foi possível reduzir tanto o tempo de criação, como o tempo de invocação de um Op. *binding*, na comparação com a abordagem particularizada e restritiva do OOPP.

Apesar de, nesse trabalho, as discussões serem conduzidas e ilustradas em termos de uma linguagem de implementação específica, a arquitetura de implementação e estratégias usadas são independentes de linguagem e podem ser empregadas usando-se outras linguagens tanto interpretadas quanto compiladas.

Em termos de trabalhos relacionados, não foram encontrados na literatura trabalhos que discutam em profundidade aspectos de implementação, que efetivamente proveêm o suporte para adaptação dinâmica. Os trabalhos restringem-se em discutir arquiteturas do middleware e elementos de tal arquitetura que dão suporte a adaptação dinâmica. Os aspectos de implementação são superficialmente tratados. Consideramos, então, que há uma lacuna importante em termos de mapeamento das abstrações arquiteturais para os aspectos de implementação dessa arquitetura. É justo essa lacuna que esse trabalho procurou preencher.

Referências

Agha, G. A. (2002). Adaptive middleware. *Commun. ACM*, 45(6):31–32.

Andersen, A., Blair, G. S., and Eliassen, F. (2000). A reflective component-based middleware with quality of service management. In *PROMS 2000, Protocols for Multimedia Systems*, Cracow, Poland.

Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98.

Blair, G. S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. B. (2001). The design and implementation of Open ORB v2. *IEEE Distributed Systems Online*, 2(6).

Blair, G. S., Coulson, G., Robin, P., and Papathomas, M. (1998). An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London. Springer-Verlag.

Fitzpatrick, T., Blair, G., Coulson, G., Davies, N., and Robin, P. (1998). Supporting adaptive multimedia applications through open bindings. In *Proceedings of the International Conference on Configurable Distributed Systems*, page 128. IEEE Computer Society.

Kon, F., Costa, F., Blair, G., and Campbell, R. H. (2002). The case for reflective middleware. *Commun. ACM*, 45(6):33–38.

Okamura, H., Ishikawa, Y., and Tokoro, M. (1992). AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the Workshop on New Models for Software Architecture*.

Smith, B. C. (1982). *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc.

Tripathi, A. (2002). Challenges designing next-generation middleware systems. *Commun. ACM*, 45(6):39–42.