MMiddle: Um Ambiente Multi-Middleware para Desenvolvimento de Aplicações Distribuídas

André Gustavo Duarte de Almeida, Thais Vasconcelos Batista, Flávia C. Delicato

Departamento de Informática e Matemática Aplicada – Universidade Federal do Rio Grande do Norte (UFRN)
Campus Universitário – 59.056-300 – Natal – RN – Brasil

andre@consiste.dimap.ufrn.br, thais@dimap.ufrn.br, flavia.delicato@dimap.ufrn.br

Abstract. *This paper presents a proposal of a multi-middleware environment to the development of distributed applications that abstracts away different underlying middleware platforms. The paper describes: (i) the specification of a reference architecture to the environment, (ii) an implementation that validates such an architecture integrating CORBA and EJB, (iii) a case study that illustrates the use of the environment and (iv) a performance evaluation. The proposed environment, promotes reuse of components from different middleware platforms in a transparent way to the developer and with no expressive performance loss.*

Resumo. *Este artigo apresenta a proposta de um ambiente multi-middleware para desenvolvimento de aplicações distribuídas, o qual abstrai diferentes plataformas de middleware subjacentes. O artigo descreve: (i) a arquitetura de referência especificada para o ambiente, (ii) uma implementação que valida tal arquitetura integrando CORBA e EJB, (iii) um estudo de caso ilustrando o uso do ambiente, (iv) a análise de desempenho. O ambiente proposto promove o reuso de componentes de diferentes plataformas de middleware de forma transparente para o desenvolvedor de aplicações e sem perdas expressivas em termos de desempenho.*

1. Introdução

Plataformas de *middleware* [Bernstein 96] oferecem uma infra-estrutura para facilitar o desenvolvimento baseado em componentes definindo formas padrão para declaração de interfaces de componentes e para comunicação entre eles. A plataforma CORBA (*Common Object Request Broker Architecture*) [OMG 04] tem se destacado entre as demais por ser uma especificação aberta, independente de fabricante e linguagem. EJB (*Enterprise Java Beans*) [Sun 03] é uma especificação que define uma arquitetura para o desenvolvimento de componentes utilizando a linguagem *Java*. A grande popularidade de Java, e o fato de ser fornecida uma implementação de referência, tornaram essa especificação uma das mais utilizadas no mundo.

Atualmente uma das principais metas no desenvolvimento de aplicações distribuídas é a interoperabilidade, principalmente devido a grande diversidade de aplicações legadas, base de dados, linguagens de programação e sistemas operacionais que precisam operar em conjunto. O uso de plataformas de middleware como as supracitadas garante um nível de interoperabilidade no qual aplicações distribuídas implementadas em diferentes linguagens e executando em diferentes plataformas de hardware e sistemas operacionais podem interagir, desde que haja o suporte subjacente de uma mesma plataforma de middleware.

Dessa forma, o uso de middleware provê *transparência* de linguagem e sistema operacional.

Interoperabilidade a nível de diferentes plataformas de middleware é importante pois cada plataforma usa diferentes modelos de objetos, cada qual com características e benefícios específicos, e podendo interoperar com determinados sistemas legados. Há muitas situações em que componentes que seguem diferentes modelos precisam ser combinados para prover uma solução completa para uma aplicação. Além disso, o amplo reuso de componentes apenas é possível se houver suporte a coexistência de componentes de diferentes plataformas de *middleware* em uma mesma aplicação e de forma transparente para o programador.

A especificação CORBA promove o reuso através do suporte a interoperabilidade, especificando uma interface comum para componentes, escrita na linguagem IDL (*Interface Definition Language*). Tal interface é independente de linguagem de programação e pode ser acessada por qualquer cliente escrito em qualquer linguagem que tenha o *binding* para CORBA. Componentes de outras plataformas de middleware não são tão reusáveis. Por exemplo, o EJB trabalha apenas com a linguagem Java, e no .Net os componentes são especificados para possibilitar a comunicação com outros componentes da própria plataforma. A sistemática de funcionamento do CORBA poderia ser aplicada nesses outros middlewares para dar suporte a interoperabilidade. Porém, mesmo com o suporte de CORBA, tal interoperabilidade é dificultada: (i) pelo fato de outras plataformas de middleware não descreverem a interface IDL dos componentes; (ii) por haver incompatibilidade das implementações CORBA não escritas na linguagem específica da plataforma (Java no caso do EJB); (iii) pela ausência de uma implementação de referência para CORBA; (iv) pelo pouco suporte para transporte de objetos complexos entre chamadas remotas de linguagens de programação diferentes.

Outra estratégia amplamente utilizada para se obter interoperabilidade são os *Web Services* [Booth et al 04], onde servidores disponibilizam serviços descritos através de uma interface especificada em WSDL (*Web Service Description Language).* A interoperabilidade é provida pelo uso de padrões em termos de linguagens e protocolos de comunicação. Entretanto, *Web Services* não suportam o paradigma de conversação entre objetos [Coelho 07], implementando o paradigma de troca de mensagens, que não permite o cliente referenciar objetos remotos. Soluções como WSRF (*Web Services Resource Framework*) [Granham et al 06] fornecem aos *Web Services* a capacidade de manter estados transacionais, porém continuam sem permitir o uso de objetos remotos diretamente pelo cliente, cabendo a esse implementar mecanismos para reconstruir tais objetos.

Pelo exposto, observa-se a necessidade de prover um ambiente que suporte o uso conjunto de diversas plataformas de *middleware,* de forma transparente para as aplicações e que, ao mesmo tempo, não onere o desenvolvedor nem gere um grande impacto no desempenho das aplicações. Para definir a especificação desse ambiente faz-se necessário prover uma arquitetura de referência, que deve: (i) especificar como os componentes das diversas plataformas de *middleware* são acessadas pela camada multimiddleware, (ii) prover mecanismos de seleção dinâmica de componentes, que abstraiam do desenvolvedor as complexidades inerentes a busca de componentes em cada plataforma, (iii) possibilitar a utilização dos componentes de forma transparente sem a necessidade de modificar ou adaptar componentes das diversas plataformas de middleware para uso dentro do ambiente proposto, (iv) realizar o mapeamento de dados para a linguagem alvo do ambiente.

O propósito desse trabalho é prover uma arquitetura de referência para desenvolvimento de aplicações distribuídas com suporte a diversidade de plataformas de middleware, chamada arquitetura *multi-middleware (MMiddle)*. Essa arquitetura deve oferecer suporte a construção de aplicações que precisem acessar componentes existentes nas diversas plataformas de middleware para realizar seus objetivos, sem que seja necessário adaptar tais componentes. Também é finalidade desse trabalho fornecer uma implementação que valide a especificação definida. Tal implementação utilizará, como exemplo, EJB e CORBA e será desenvolvida no contexto do LuaSpace [Almeida 06], um ambiente que utiliza uma linguagem de configuração para definição de estrutura de aplicações baseadas em componentes e que dá suporte a reconfiguração dinâmica de aplicações, promovendo o reuso de componentes CORBA. Portanto, o ambiente será expandido, incorporando os elementos da arquitetura de referência e suporte a interoperabilidade CORBA-EJB.

Este artigo está estruturado da seguinte forma. A Seção 2 apresenta os conceitos básicos das plataformas usadas nesse trabalho: EJB e CORBA. A Seção 3 apresenta a arquitetura de referência e a implementação que valida a arquitetura definida. A Seção 4 apresenta o estudo de caso realizado com a implementação, juntamente com uma análise de desempenho. A Seção 5 compara a proposta apresentada com trabalhos relacionados. A Seção 6 contém as conclusões.

2. Conceitos Básicos

2.1 CORBA

CORBA (*Common Object Request Broker Architecture*) [OMG 04] é um padrão proposto pela *Object Management Group* (OMG) cujo propósito é permitir interoperabilidade entre aplicações em ambientes distribuídos e heterogêneos. Este padrão estabelece a separação entre a interface de um objeto e sua implementação. Para descrição da interface do objeto, CORBA oferece a linguagem para definição de interfaces (IDL). Para implementação do objeto CORBA, pode ser utilizada qualquer linguagem de programação que tenha o mapeamento (*binding*) para CORBA. A arquitetura CORBA é composta por um conjunto de blocos funcionais que usam o suporte de comunicação do ORB (*Object Request Broker*) - o elemento responsável por coordenar as interações entre os objetos, interceptando as chamadas dos clientes e direcionando-as para o servidor apropriado.

Todo objeto CORBA possui uma identificação, chamada *referência do objeto*, que é atribuída pelo ORB na criação do objeto. Para usar um objeto, o cliente deve obter a sua referência, pois em uma invocação de um método sobre o objeto o ORB o identifica através da mesma. O *Repositório de Interfaces* definido no padrão CORBA disponibiliza informações necessárias para a construção de chamadas dinâmicas. Este repositório armazena todas as definições IDL dos objetos CORBA disponíveis para uso. A utilização do repositório de interfaces para a localização de objetos apresenta a restrição de ser necessário conhecer a referência do objeto para adquirir mais informações sobre ele. Todas as implementações CORBA devem prover suporte ao protocolo IIOP (*Internet Inter-Orb Protocol*) que deve ser usado em redes TCP/IP. Esse protocolo é usado para envio/recebimento de mensagens entre o cliente e o servidor possibilitando a transmissão de diversos tipos de dados suportados por CORBA.

2.2 EJB

Os elementos que compõem a especificação EJB são: servidor EJB, *container*, componente (*bean* ou *enterprise bean*), descritor de implantação, interface *home* e interface *remote*. O servidor *EJB* gerencia um ou mais *containers* e provê serviços comuns como transações, segurança, persistência, entre outros. Há uma variedade de servidores EJB disponíveis no mercado: BEA's WebLogic [BEA 07], IBM WebSphere [Jain 07] e JBoss [Matsumura 05].

O *container EJB* tem como finalidade oferecer ao programador do componente os serviços disponibilizados pelo servidor. Os serviços são definidos de maneira ortogonal ao componente, ou seja, a especificação dos serviços utilizados na aplicação é separada dos arquivos Java que implementam a lógica da aplicação. Usando uma *semântica declarativa* o programador especifica, em um arquivo XML chamado *descritor de implantação,* as instruções para implantação do componente, a lista de recursos necessários para o mesmo, os papéis de segurança para a aplicação, a informação de autenticação e a lista de controle de acesso para os vários métodos. Isto é possível porque o *container* é o intermediário entre o cliente e o componente, interceptando todas as chamadas de métodos direcionadas ao componente. Da mesma forma que um *stub* RMI está entre o cliente e o objeto remoto, o *container* EJB está entre o cliente e o componente. O cliente nunca acessa diretamente um método do componente, o acesso é realizado via *container*, através de suas interfaces *home* e *remote*.

A interface *home* é responsável pelo controle das operações de ciclo de vida de um componente: criação, remoção e localização. A interface *remote* expõe a *interface* do componente que define os métodos que o mesmo oferece para os clientes. A interface *remote* é um *proxy* para a instância do componente, e é ele que atua como *interceptador*. Quando o cliente invoca um método do componente, esse objeto recebe a invocação e direciona para a instância do componente. Antes de fazer o direcionamento, há a interceptação dos serviços requisitados pelo componente.

2.3 Aspectos de Interoperabilidade CORBA-EJB

A interoperabilidade entre CORBA e EJB reside na capacidade de clientes escritos em qualquer linguagem poderem acessar componentes EJB como se estivessem acessando objetos CORBA. Um das metas da arquitetura EJB é fazer com que clientes EJB e servidores CORBA e vice-versa interajam entre si, fazendo com que componentes EJBs sejam utilizados em uma ampla gama de sistemas heterogêneos, evitando que restrições desnecessárias sejam feitas aos arquitetos de software no momento de projetar o sistema ou a integração do mesmo. O principal problema é que essa interação não ocorre de forma trivial, precisando adaptar o modelo EJB ao modelo CORBA, muitas vezes de forma manual e com esforço extra do programador para construir entidade que tornem essa interação a mais homogênea possível.

A maioria das implementações EJB, tais como JBoss e a implementação de referência da Sun, foram construídas com base no protocolo CORBA-IIOP (*Internet Inter-Orb Protocol*), permitindo que a comunicação CORBA-EJB ocorra seguindo os padrões CORBA para desenvolvimento de aplicações. Como visto anteriormente, para desenvolver aplicações CORBA é necessário escrever um contrato, em IDL, do componente que oferece/solicita serviços. Usando tal contrato juntamente com o *binding* específico para cada linguagem, a interoperabilidade é alcançada. Componentes CORBA podem ser escritos em qualquer linguagem desde que tenham associadas a si uma especificação IDL, enquanto componentes EJB são escritos unicamente em Java. Ou seja, para que um cliente CORBA acesse um componente EJB o mesmo deve ter uma interface IDL equivalente para

que seja realizada a comunicação. Na especificação [OMG 03] são definidos os procedimentos para mapeamento Java para IDL. Na versão 2.3 da especificação CORBA o sistema de tipos de dados da IDL foi ampliado para que fosse possível definir *CORBA ValueTypes,* que equivale a objetos locais que são objetos definidos pelo usuário e podem ser passados como parâmetros e retornados por métodos. A instância de uma classe Java é equivalente a um *CORBA ValueType*. Sem a adição desse tipo de objeto seria impossível existir interoperabilidade utilizando componentes escritos em Java.

Existem vários problemas relacionados ao mapeamento EJB para CORBA, que são discutidos em [IONA 01]. Um dos principais problemas citados é a necessidade do cliente não escrito em Java precisar familiarizar-se com detalhes específicos da linguagem. No desenvolvimento de aplicações EJB, com base em especificações anteriores a 3.0, para cada componente EJB instalado no servidor existem duas interfaces: *Home* e *Remote*. Para que clientes CORBA acessem esse EJB é necessário definir as mesmas interfaces em IDL e gerar os *stubs* requeridos para que as mesmas sejam utilizadas. Há ferramentas que produzem IDL com base em classes Java. Uma delas é o compilador *rmic* disponível em qualquer versão do JSDK (*Java System Development Kit*) da *SUN*. Essas ferramentas seguem a especificação [OMG 03] para construir as IDLs. O problema associado a esse tipo de ferramenta é a construção de IDL com grande quantidade de informações desnecessárias e não amigáveis, podendo gerar uma infinidade de arquivos que muitas vezes são inúteis para o funcionamento da aplicação. De posse dos *stubs*, o cliente CORBA pode realizar chamadas para o EJB da mesma forma como realizaria chamadas para componentes CORBA. Clientes CORBA utilizando estratégias de geração automática de *stubs* podem recorrer, com mais sucesso, ao mapeamento automático de componentes EJB em interfaces IDL A Figura 1 mostra a sistemática de chamada de clientes CORBA a componentes EJB.

Para que o componente EJB seja acessado via um cliente CORBA, o EJB precisa ser resolvido pelo serviço de nomes do CORBA. Normalmente a implementação do serviço de nomes CORBA é feita pelo *container* EJB. O cliente CORBA instancia o serviço de nomes, em seguida realiza a operação de *lookup* (que retorna a referência da interface *home)* e, na seqüência, o procedimento equivale ao mesmo usado em clientes Java acessando EJB: o método *create* é invocado, obtendo referência à interface *remote* e, assim, acessando os métodos de negocio implementados pelo EJB.

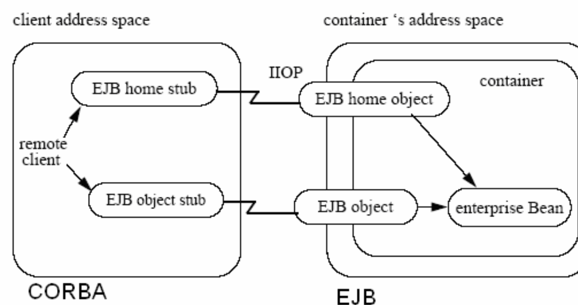


Figura 1 – Invocação de EJB através de clientes CORBA

A implementação da especificação EJB utilizada neste trabalho é o JBOSS 4.0.4 [Fleury 03], uma das implementações mais utilizadas, disponível sob a licença GPL. O JBoss pode receber requisições de clientes escritos em CORBA através do módulo IIOP, utilizando o JacORB [Brose 97], implementação CORBA em Java integrada ao servidor de aplicações. No JBoss, para que um componente EJB seja acessível através de um cliente CORBA é necessário que, no descritor de implantação, seja especificado que o mesmo aceita requisições via IIOP e RMI (invocação tradicional).

Outra abordagem alternativa para acesso a componentes EJB a partir de clientes CORBA é a utilização de *Wrappers,* os quais são objetos CORBA implementados em Java que têm como função permitir o envio/recebimento de tipos complexos Java. Supondo que um método Java receba como parâmetro um *java.util.Collection,* se usarmos o mapeamento direto é necessário que o cliente (seja C++, Python, Lua) forneça uma implementação para essa classe, uma vez que a mesma é passada por valor. Tal abordagem torna-se extremamente complexa e tediosa. Os *Wrappers* servem para mapear os tipos complexos Java em tipos mais próximos dos tipos que podem ser definidos em especificações IDL. Semelhante a abordagem citada anteriormente, onde todos os tipos complexos devem ter uma implementação na linguagem em que foi escrito o cliente, essa tática obriga os desenvolvedores EJB a implementarem *Wrappers* para todos os EJB existentes.

3. Ambiente multi-middleware

Nessa seção serão descritos o ambiente *MMidle*, a especificação de sua arquitetura de referência e a implementação desenvolvida para validar a mesma.

3.1 *Arquitetura*

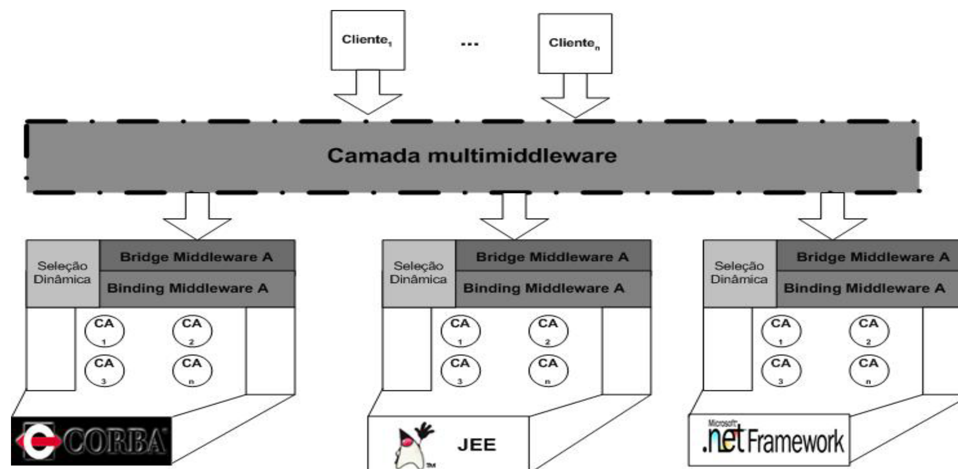


Figura 2 - Arquitetura do ambiente de desenvolvimento multi-middleware

A Figura 2 ilustra a arquitetura de referência do ambiente *multi-middleware MMiddle* para desenvolvimento de aplicações distribuídas que possam utilizar componentes de diferentes plataformas de middleware. Encontram-se em destaque os elementos chaves da interoperabilidade: (i) a *camada multi-middleware* que é responsável por abstrair a complexidade de acesso do cliente/desenvolvedor; (ii) a *bridge*, que tem como objetivo detectar o registro de componentes na plataforma de middleware específica (p.ex. em CORBA a *bridge* deve monitorar os registros de componentes no repositório de interfaces, e no JEE monitorar o *deploy* do componente) e notificar o serviço de seleção dinâmica para que o mesmo mantenha também o registro de tal componente*; (iii)* o *binding middleware*, responsável por tratar do mapeamento de tipos de dados e delegar a execução das chamadas aos componentes da plataforma; (iv) o *mecanismo de seleção dinâmica*, que tem como objetivo localizar componentes em cada plataforma de middleware.

A camada *multi-middleware* é responsável por acessar as diversas plataformas de middleware subjacentes de modo transparente para o desenvolvedor ou cliente que esteja requisitando algum serviço. Ela atua como um *Facade* [Gamma 05], delegando as requisições para cada plataforma de middleware que se registra com a camada. Na camada *multi-middleware* existe um componente que implementa a interface *IManager*, e que atua

como um gerenciador, recebendo o registro da referência dos *bridges* de cada plataforma especifica, cabendo a ele gerenciar como as solicitações devem ser feitas a esses *bridges.*

O mecanismo de seleção dinâmica é implementado para cada plataforma, de forma a lidar com as peculiaridades relativas a busca de objetos em cada uma delas. Tal mecanismo não lida com mapeamentos ou invocações de métodos. Essas tarefas cabem ao componente *bridge*. O objetivo do mecanismo de seleção é localizar objetos, utilizando critérios de busca que se baseiam em assinatura de métodos e em propriedades definidas pelos serviços. Cada mecanismo de seleção tem liberdade para definir como devem ser armazenadas as informações sobre os serviços oferecidos. Porém, deve utilizar uma forma padrão, para especificar critérios de busca, a fim de ser possível garantir a combinação de critérios de busca e a uniformidade na seleção dos componentes.

Para cada plataforma existe um componente *bridge*, cujo objetivo é abstrair a complexidade inerente de acessar os diversos serviços das plataformas de *middleware* seguindo as definições especificadas pela camada *multi-middleware*. Portanto, o componente *bridge* provê mecanismos para invocação de métodos (que, no final, são delegados ao *binding)*, passagem e recebimento de valores enviados pelos diversos componentes da plataforma. Esse componente também tem a função de detectar os componentes existentes na sua plataforma de *middleware* subjacente e notificar o mecanismo de seleção dinâmica da camada *multi-middleware* da disponibilidade do mesmo, mantendo, assim, um registro do componente, da sua localização e da especificação da sua interface. Essa interface é descrita de acordo com a plataforma, por exemplo, em CORBA é a interface IDL e na nossa implementação de referência, para a plataforma JEE utilizamos um mapeamento XML para representar as interfaces dos EJB de forma acessível ao mecanismo de seleção dinâmica. Todo processo de notificação e registro dos serviços é realizado de forma transparente para os usuários da camada *multi-middleware*.

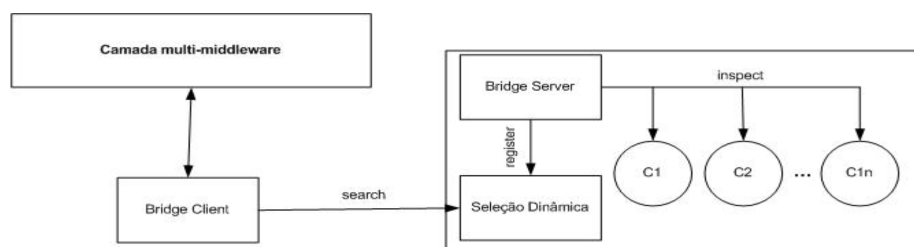


Figura 3 – Caráter dual do bridge

A implementação da camada *multi-middleware* definirá como deve ser implementado o componente *bridge* que, obrigatoriamente, deve ser implementado em um caráter dual, ou seja, uma parte deve ser implementada na linguagem referente à plataforma de *middleware* (Java para EJB, C# por exemplo, para .Net) e a outra parte, para lidar diretamente com a camada multi-middleware, implementada na linguagem de programação usada na camada *multi-middleware*. A Figura 3 ilustra a arquitetura dual do *bridge*, onde existem: (i) um módulo servidor que é responsável por detectar e inspecionar os componentes da plataforma de *middleware* especifica e, por isso, é implementando na linguagem alvo da plataforma e (ii) um módulo cliente que é responsável por invocar o mecanismo de seleção dinâmica e realizar todos os procedimentos de inicialização e configuração inerentes a cada plataforma. Na Figura 3 omitimos o *binding*, que é ilustrado no diagrama de seqüência da Figura 4.

O *Binding Middleware* é responsável por mapear os tipos de dados especificados em cada plataforma de *middleware* para os tipos de dados utilizados pela implementação da camada *multi-middleware*, além de ser responsável por receber/enviar as requisições

de/para a camada superior. Em cada plataforma há ainda os diversos componentes representados na Figura 2 pelos círculos na parte inferior da representação do *binding*. O *MMidle* foi projetado para suportar um número qualquer de plataformas de *middleware* desde que os elementos citados estejam presentes, uma vez que os detalhes de mapeamento de tipos, invocação de métodos e demais aspectos relacionados às diversas plataformas de *middleware* existentes ficam a cargo dos componentes *bridge* e *binding*.

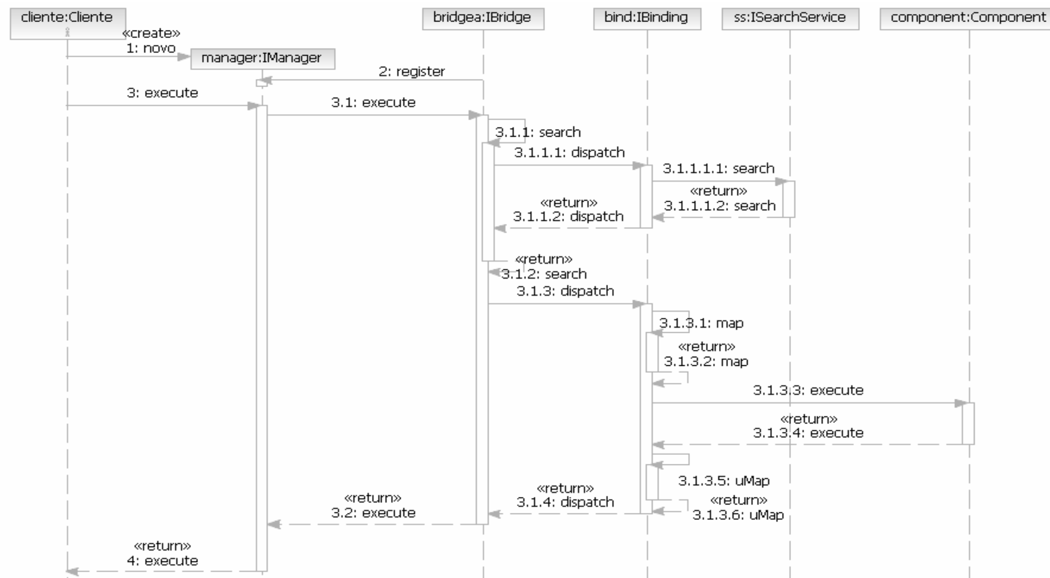


Figura 4 – Diagrama de seqüência de utilização da camada multi-middleware

A Figura 4 ilustra o diagrama de seqüência correspondente a utilização da camada *multi-middleware*. O cliente instancia um objeto que implementa a interface *IManager* (tal objeto foi implementado usando o padrão *Singleton* e consiste no gerenciador, anteriormente explicado). Em seguida os *bridges* das plataformas de middleware específicas devem registrar-se no gerenciador, para que seja possível a busca/execução dos métodos implementados pelo diversos componentes existentes nas plataformas de middleware representados pelo *bridge*. Na sequência o cliente invoca o método *execute* que recebe como parâmetros a assinatura do método e os parâmetros que devem ser passados para realizar a execução. Em seguida, para cada *bridge* registrado, o método *search* é invocado passando como parâmetro o critério de busca (a assinatura do método). O mecanismo de seleção dinâmica localiza as referências dos componentes que atendem o critério de busca. Com as referências retornadas, o *bridge* solicita ao mecanismo de *binding* a execução dos métodos dos componentes que foram retornados, devolvendo para o cliente o conjunto de resultados provenientes da execução desses métodos.

Um dos diferenciais da proposta é a possibilidade do cliente conseguir manipular objetos que foram retornados das chamadas dos métodos de negócio de forma idêntica a como funcionaria se estivesse programando na plataforma de middleware especifica. Para ilustrar essa capacidade, suponha que um método de negócio JEE retorne um objeto complexo, como um *ArrayList* ou um tipo definido pelo usuário. Através dos componentes *bridge* e *binding* é possível invocar os serviços desses objetos sem ter que reimplementar os métodos dessas classes. Tal capacidade difere da solução provida pela abordagem de *Web Services*, a qual, por não ser Orientada a Objetos, permite que apenas os dados dos objetos complexos sejam enviados após uma requisição. No caso da arquitetura multi-middleware proposta, a requisição é feita pelo *binding*, e quando o retorno é enviado, o binding tem a função de mapear esse objeto para um correspondente na linguagem alvo do ambiente.

É importante ressaltar que o ambiente multi-middleware não foca na integração entre as plataformas de middleware de forma a permitir que, por exemplo, haja uma comunicação direta entre componentes CORBA e EJB. O ambiente permite que componentes de diversas plataformas de middleware sejam utilizados de forma transparente para uma aplicação. Isso poderia ser necessário para integrar sistemas legados, tendo o ambiente *MMiddle* como uma camada adicional onde os componentes seriam acessados de forma transparente e o resultado dessa execução seria retornado para a aplicação.

3.2 Implementação

A implementação de referência da arquitetura descrita na Seção 3.1 utiliza Lua como linguagem para a camada *multi-middleware*. Tal linguagem foi escolhida por oferecer mecanismos de reflexão através de *tag methods* e uma diversidade de *bindings* para várias linguagens, que permitem que código escrito nessas linguagens sejam acessíveis pelo código Lua e vice-versa. Os *bindings* utilizados na implementação foram desenvolvidos por terceiros: o *binding* de Lua para CORBA é o *LuaOrb* [Cerqueira 99] e para a plataforma JEE da linguagem Java utilizamos *LuaJava* [Cassino 99] uma ferramenta de scripting que permite que objetos Lua sejam referenciados por objetos Java e vice-versa. *LuaJava* executa através da JVM, que deve ser inicializada para que a ferramenta possa ser utilizada, enquanto *LuaOrb* é uma biblioteca escrita em C++. Para fazer com que as duas ferramentas coexistam, permitindo que objetos CORBA e Java trabalhem dentro do mesmo espaço de execução, modificamos *LuaJava* para incluir uma chamada JNI (*Java Native Interface*), que permite que aplicações Java executem aplicações nativas. Os mecanismos de *binding* não são conversores de objetos, ou seja, um objeto Java não é convertido em um objeto Lua. O *binding* cria *proxys* para os objetos de forma que a invocação do método de um objeto fica transparente para o utilizador do ambiente. No final o objeto Java é que é invocado (no caso do LuaJava). O mecanismo de seleção dinâmica utilizado para a plataforma CORBA é o *Discovery Service* [Cacho et al. 04] que suporta buscas assíncronas, balanceamento de carga e combinação de critérios de busca. Para a plataforma JEE implementamos uma versão simplificada do *Discovery Service*, que permite a composição de critérios de busca, porém sem balanceamento de carga e busca assíncrona. Na Tabela 1 apresentamos o conjunto de propriedades que podem ser utilizadas como critério de busca no *MMiddle*.

Tabela 1 – Nome de propriedades do Mecanismo de Seleção

Propriedade	**Descrição**
Servicename	Nome do serviço
OperationName	Nome da operação
ParamName	Nome do Parâmetro
OperationExceptionName	Nome de exceção gerada por uma operação
AttributteName	Nome de atributo

Como mencionado, o componente *bridge* é responsável por detectar os componentes da plataforma de middleware específica e notificar o serviço de seleção dinâmica. No caso de CORBA, quando a interface de um componente é publicada no repositório de interface, o *bridge* para CORBA registra essa interface com o *Discovery Service*. Para JEE, quando um EJB é instalado no servidor de aplicação o *bridge* inspeciona todas as interfaces e classes que compõem o componente e constrói uma representação XML das interfaces de acesso, independente da especificação. Isso é feito através de reflexão computacional, onde para a especificação 2.1 do EJB verificamos se a interface é descendente de *home* e *remote*, e para a especificação 3.0, se a interface está

apropriadamente anotada. Uma vez feita a notificação ao serviço de seleção dinâmica, o componente está disponível para localização.

```
1 local c=ManageService.getInstance()
2 local results=c:search("operationname=='print'")
3 for i=1,table.getn(results) do
4      results[i]:print("hello")
5 end
```

Figura 5 – Seleção dinâmica de componentes

Figura 5 mostra o trecho de código para seleção de todos os componentes que possuem a operação com nome *print.* Na linha 1 recuperamos a referência ao façade (gerenciador) da camada *multi-middleware* que contém a referência aos mecanismos de seleção. Na linha 2 o método *search* é invocado recebendo como parâmetro o critério de busca *operationname=='print'*. Esse critério é delegado para todos os serviços de seleção registrados, cabendo a cada um realizar a busca e retornar a referência ao objeto que possui tal método. Nas linhas 3 e 4 há um laço para a execução do método *print* de todos os objetos retornados. Deve-se notar que pelo critério de busca poderíamos retornar qualquer método cuja assinatura fosse *print*, independente do número de parâmetros. Na linha 4 determinamos a execução do método passando apenas um parâmetro do tipo string. Cabe a *bridge* delegar a execução de forma apropriada aos componentes retornados pelo método *search.*

4. Estudo de Caso e Avaliação de Desempenho

4.1 Estudo de caso

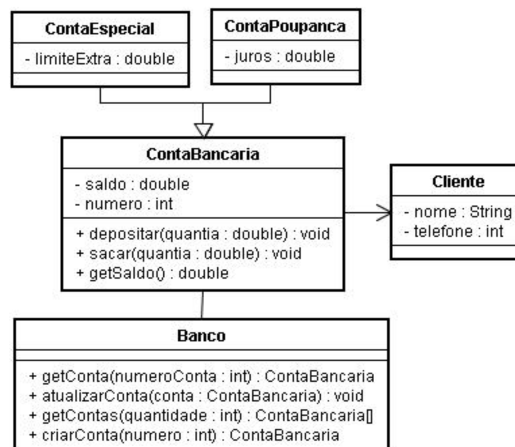


Figura 6 – Diagrama de Classes da Aplicação Bancária

Nessa seção discutiremos o estudo de caso que mostra as potencialidades do ambiente *multi-middleware* e consiste de uma aplicação bancária para o *Banco Money*. O banco foi formado a partir da aquisição de vários bancos menores. Os sistemas de informação desses bancos compartilham a mesma base de dados, porém foram implementados utilizando plataformas de *middleware* diferentes, existindo uma implementação em CORBA e outra em EJB 3.0. A Figura 6 mostra o diagrama de classe da aplicação bancária considerada.

Na aplicação desenvolvida com uso de CORBA as classes *Cliente*, *ContaBancaria*, *ContaEspecial* e *ContaPoupança* são especificadas em IDL através de *valuetypes* e a classe *Banco* como uma interface. A implementação da interface *Banco* é feita utilizando a linguagem Lua e é registrada no serviço de seleção dinâmica *Discovery Service*.

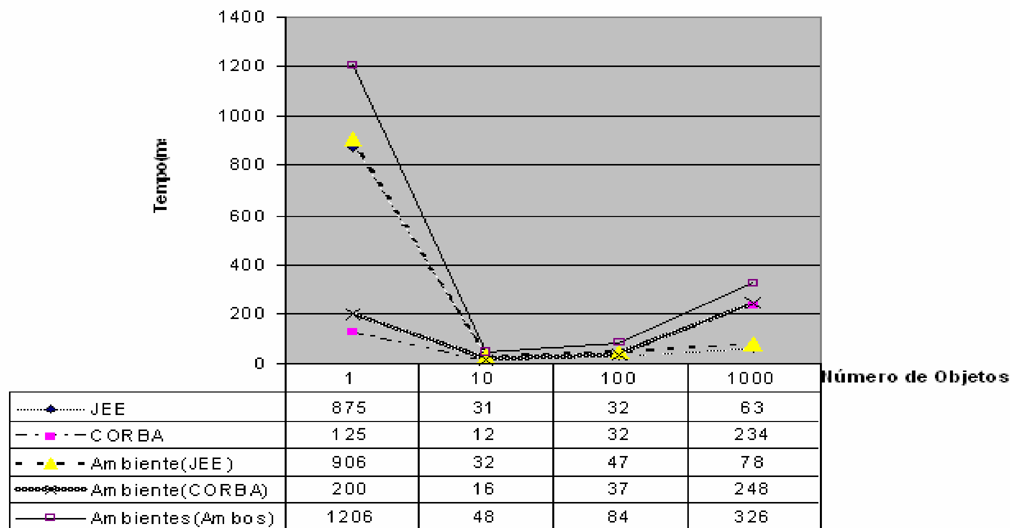
Na aplicação desenvolvida utilizando JEE a classe Banco é implementada através de um *Stateless* EJB com interface Remota e as demais classes são consideradas classes de domínio. Para efeito de simplificação não utilizamos *Entity Beans* nesse exemplo. Uma vez instalado no servidor, o componente *bridge* para a plataforma JEE deve inspecionar as classes e interfaces, mapeando a definição das mesmas para o formato XML e notificando o serviço de seleção dinâmica. Para efeito de testes algumas instâncias da classe *ContaBancaria* e *ContaEspecial* foram criadas em ambas as implementações (CORBA e JEE) para simularem a base de dados. A Figura 7 mostra o trecho de código da aplicação principal. Na linha 1 recuperamos a instância do *ManageService*, chamando em seguida o método *search* passando como parâmetro o critério de busca "*operationname=='criarConta' || operationname=='getContas'*", esse critério de busca é repassado aos mecanismos de seleção dinâmica registrados no *ManageService*. Em seguida, para cada componente encontrado executamos o método *criarConta*, passando número da conta. No nosso caso o conjunto de resultados tem tamanho 2, referindo-se a implementação CORBA e EJB. Se a assinatura do código *criarConta* fosse diferente em uma das implementações seria necessário verificar se *resulados[i]:criarConta* retornaria valor *nil*, o que indicaria a não existência do método. Na linha 4 executamos o método do objeto, que representa uma *ContaBancaria*, que em CORBA é um *valuetype,* e em JEE equivale a uma classe Java pura (POJO). Para conseguir fazer o mapeamento apropriadamente o *binding* precisa acessar a definição da classe, ou seja, do mesmo jeito que no desenvolvimento de aplicações JEE é necessário gerar um arquivo *jar* contendo arquivos para o cliente Java.

```
1 local mng=ManageService.getInstance()
2 local resultados=mng:search("operationname=='criarConta' ||
                operationname='getContas'")
3 for i=1,table.getn(resultados) do
4      obj=resultados[i]:criarConta(1)
5      obj:depositar(100)
6      print(obj:getSaldo())
7 end
```

Figura 7 – Localização e Execução da Aplicação Bancária

4.2 Avaliação de Desempenho

A Figura 8 apresenta o gráfico da análise de desempenho do *MMiddle* A análise foi feita utilizando o estudo de caso apresentado, acrescentando na classe *Banco* o método *getContas*, o qual recebe um inteiro com a quantidade de contas a serem criadas e, em seguida, retornadas para o cliente. Foi feita uma simulação com chamadas consecutivas a esse método, criando-se uma variável de simulação representando o número de objetos a serem retornados por ele. Tal variável foi configurada com os valores 1,10,100 e 1000. Na primeira vez que a chamada ao método foi realizada, a variável foi configurada para o valor 1 (indicando que apenas 1 objeto deveria ser retornado por *getContas*), e nessa chamada foi contado o tempo de inicialização (carregar *bean*, executar busca). Em seguida foi medido o tempo transcorrido (em milisegundos) antes da chamada do método *getContas* e imediatamente após a chamada. Isso foi feito respectivamente para: (i) um cliente em Java do *Bean*, (ii) um cliente CORBA da implementação CORBA do estudo de caso, (iii) utilizando o ambiente *multi-middleware* somente com o *bean* instalado no servidor, (iv) novamente utilizando o ambiente com apenas a implementação CORBA executando e (v) com as duas implementações (EJB e CORBA) em execução no ambiente *MMiddle*.



	1	10	100	1000
JEE	875	31	32	63
CORBA	125	12	32	234
Ambiente(JEE)	906	32	47	78
Ambiente(CORBA)	200	16	37	248
Ambientes(Ambos)	1206	48	84	326

Figura 8 – Gráfico de desempenho (Número de Objetos X Tempo de execução)

O gráfico mostra que, inicialmente, o tempo de execução em todos os cinco casos é alto devido a carga inicial dos serviços necessários para o funcionamento da plataforma de middleware. Por exemplo, no início da execução de CORBA são carregados os serviços de repositórios de interface e de nomes. Para as simulações com uso do ambiente multi-middleware é levado em conta o tempo de inicialização mencionado anteriormente mais o tempo que o serviço de seleção dinâmica específico de cada plataforma leva para fazer busca de componentes. A avaliação seguinte refere-se a execução do método *getContas* para chamadas com os valores de 10,100 e 1000 objetos a serem retornados. Podemos verificar que o tempo varia muito pouco entre as chamadas feitas a partir de uma plataforma específica (J2EE, por exemplo), e com a mesma plataforma no ambiente multi-middleware. A diferença acontece somente no momento do mapeamento dos dados pelo *binding* especifico e no momento inicial onde a busca é realizada pelo mecanismo de seleção dinâmica. Concluímos a partir do gráfico que o tempo total incluindo a busca dos componentes e o uso do *binding* não se distancia do tempo de execução levado pelo uso das plataformas isoladas. A diferença de tempo de execução que ocorre entre as chamadas com 100 e 1000 objetos é explicada pelo aumento na quantidade de objetos a serem retornados e relaciona-se principalmente ao retorno do método na plataforma CORBA. A complexidade relacionada ao método, o qual retorna uma lista de tipos complexos, torna a sua execução mais lenta quando utilizando a plataforma CORBA, uma vez que o suporte de tipos complexos é mais custoso em termos de desempenho, devido aos procedimentos de empacotamento e desempacotamento (*marshalling* e *unmarshalling*) utilizados pelo protocolo IIOP do CORBA. Para medir esses tempos utilizamos, antes e depois da invocação dos métodos, a chamada *System.currentMilis()*, que retorna o número de millisegundos transcorridos de 1 de Janeiro de 1970 até o momento atual.

5. Trabalhos Relacionados

A tecnologia de Serviços Web (*Web Services*) provê uma solução para a integração de sistemas e a comunicação de diferentes aplicações, permitindo que dados sejam enviados/recebidos através de mensagens em formato XML. Os *Web Services* através do WSRF (*Web Services Resource Framework*) [Granham 06] podem realizar conversações que não são orientadas a objeto [Hopkins 05], nas quais o estado de um objeto é expresso através de um documento XML, porém o seu comportamento (métodos) não pode ser transmitido, fazendo com que o cliente precise implementar tais métodos se desejar fazer

com que o comportamento siga o do objeto original. Em nossa proposta, o ambiente multi-middleware é orientado a objetos e pode receber objetos provenientes do retorno dos métodos, sendo tal capacidade possibilitada pelos componentes que integram o ambiente.

Em [Chiang 07] é discutido um gerador automático de *Wrappers* CORBA, os quais escondem a complexidade do código do servidor para as aplicações cliente. Aplicado inicialmente para prover transparência de acesso a aplicações *Mainframe*, o gerador pode ser utilizado para integrar diferentes plataformas de middleware, porém nenhuma implementação especifica é discutida e não fica claro o nível de complexidade relacionado à geração desses *wrappers*, bem como desempenho dos mesmos. Em contraste, nosso trabalho apresenta a arquitetura e implementação da solução proposta e mostra que não há perdas expressivas de desempenho. O diferencial da nossa proposta é a não necessidade do programador desenvolver esses *wrappers*. Além disso, como dito na introdução, o objetivo do ambiente é possibilitar o acesso de componentes de software provenientes de diversas plataformas de middleware e não criar uma camada para interação entre essas plataformas de middleware.

6. Conclusões

Esse trabalho apresentou o *MMiddle*, um ambiente *multi-middleware* para desenvolvimento de aplicações distribuídas que permite o uso de diversas plataformas de middleware, sem perder as características intrínsecas de cada uma delas. O ambiente fornece uma forma simples de programadores reusarem componentes de diferentes plataformas sem precisar saber em qual plataforma o componente encontra-se, nem em qual linguagem é implementado. O ambiente tem o diferencial de prover suporte a conversão orientada a objetos, resolvendo os problemas de incompatibilidade entre os dados das plataformas de *middleware*. O mecanismo de seleção dinâmica permite a localização de componentes combinada com a subseqüente invocação da execução dos métodos dos componentes localizados, sem a necessidade de conhecer em detalhes as interfaces dos serviços implementados nas plataformas especificas ou mesmo os seus nomes. É importante frisar que o ambiente não propõe uma nova forma de interoperabilidade entre plataformas de middleware nem visa permitir que os componentes das diversas plataformas comuniquem-se diretamente.

Os *WebServices*, como mencionado na seção de trabalhos relacionados, têm sido a solução mais utilizada para alcançar interoperabilidade entre diversas plataformas de middleware, bem como com aplicações legadas. Nosso objetivo não foi produzir uma solução melhor ou pior que *WebServices,* mas apontar outra solução para se alcançar interoperabilidade. Apesar da solução que propomos integrar diversos elementos e diferentes tecnologias dentro de um ambiente, os desenvolvedores apenas precisam registrar seus componentes no serviço de seleção dinâmica da sua plataforma de middleware, através do mecanismo de *bridge*. As demais funções que permitem a interoperabilidade são realizadas de forma transparente pela plataforma multimiddleware.

Referências

Almeida, A. D., Batista and Cacho, N., Batista T.(2006) "LuaSpace EPlus: Um Ambiente para Desenvolvimento de Aplicações CORBA no Eclipse" In: Anais do XIV Simpósio Brasileiro de Redes de Computadores (SBRC'2006), SBC, pp. 1315-1330, Curitiba - PR, Maio 2006

BEA Systems(2007), Bea Weblogic Server 10: The Rock Solid Foudantion for SOA, http://www.bea.com/content/news_events/white_papers/BEA_WL_Server10_wp.pdf

Bernstein, P.(1996) Middleware. Communications of the ACM, 39(2), February 1996.

Brose, G. (1997) “JacORB: Implementation and Design of a Java ORB”. In Proceedings of Dais 97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Cottbus, Germany, Chapman & Hall.

Booth, D. et al. (2004) Web Services Architecture Specification. Disponivel em http://www.w3.org/TR/ws-arch/

Cacho, N., Batista, T. and Elias, G. (2004) Um Serviço CORBA para Descoberta de Componentes, In: Anais do XVIII Simpósio Brasileiro de Engenharia de Software (SBES'2004), SBC, pages 273-288, ISBN85-7669-002-0, Brasilia, DF, Outubro.

Cassino, C. and Ierusalimschy, R. (1999) “LuaJava – Uma Ferramenta de Scripting para Java” In Simpósio Brasileiro de Linguagens de Programação (SBLP´99), pp. 125-137, Porto Alegre.

Cerqueira, R., Cassino, C. and Ierusalimschy, R. (1999) “Dynamic Component Gluing Across Different Componentware Systems”. In International Symposium on Distributed Objects and Applications (DOA´99), 362-371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

Chiang, C-C. (2007) ACM “Automatic software wrapping” In Proceedings of the 45th annual southeast regional conference, Winston-Salem, North Carolina USA, p. 59-64, March.

Coelho, O.(2007) Escolhendo entre Web Services, Enterprise Services e Remoting. Disponível em http://msdn.com/brasil/msdn/Tecnologias/arquitetura/Escolhendo.mspx

Fleury, M. and Reverbel F. (2003) “The JBoss Extensible Server”, Lecture Notes in Computer Science, Springer Berlin / Heildelberg, pp. 344-373.

Gamma, E., Helm R., Johnson R. e Vlissides (2005) Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos. Editora Bookman. pp. 179-186.

Granham, S. et all (2006). Web Services Resource Framework Specification. Disponivel em http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf

Hopkins, R. (2005) Web Services Resource Framework, Disponível em http://www.nesc.ac.uk/action/esi/download.cfm?index=2836

Ierusalimsky, R., Figueiredo, L. H., and Celes, W. (1996) “Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652.

Jain,A.(2007) What´s new in WebSphere Application Server Community Edition. Disponível em http://www.ibm.com/developerworks/websphere/library/techarticles/0709_jain/0709_jain.html

IONA Technologies (2001) CORBA-EJB Interoperability White Paper. Disponível em http://whitepapers.silicon.com/0,39024759,60041720p-39000396q,00.htm

Matsumura, M.(2005) JBoss Application Server: Standard Based Infrastructure for the Enterprise. http://www.jboss.com/pdf/JBossAS-EnterpriseInfrastructure.pdf

OMG (Object Management Group) (2003) Java to IDL Mapping. Disponível em ftp://ftp.omg.org/pub/docs/ptc/99-03-09.pdf

OMG (2004) Common Object Request Broker Architecture: Core Specification Technical Report Revision 3.0.3.

SUN(2003) Enterprise Java Beans Specification. Disponível em http://java.sun.com/products/ejb/docs.html