Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

AO-OIL: Um Middleware Orientado a Aspectos Baseado em uma Arquitetura de Referência

Dissertação de Mestrado

José Diego Saraiva da Silva

Natal, RN, Fevereiro de 2009

José Diego Saraiva da Silva

AO-OIL: Um Middleware Orientado a Aspectos Baseado em uma Arquitetura de Referência

Dissertação de mestrado submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Orientador:Profª. Drª. Thaís Vasconcelos Batista (UFRN)

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Silva, José Diego Saraiva da.

AO-OIL: um middleware orientado a aspectos baseado em uma arquitetura de referência / José Diego Saraiva da Silva. – Natal , RN, 2009.

120 f.

Orientador: Thaís Vasconcelos Batista.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em
Sistemas e Computação.

1. Middleware – Aspectos – Dissertação. 2. Arquitetura de referência para middlewares orientados a aspectos – Dissertação. 3. RE-AspectLua – Dissertação. I. Batista, Thaís Vasconcelos. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 004.75(043.3)

AO-OIL: Um Middleware Orientado a Aspectos Baseado em uma Arquitetura de Referência

José Diego Saraiva da Silva

Esta dissertação de mestrado foi avaliada e considerada aprovada pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

_____

Profª. Drª. Thaís Vasconcelos Batista (UFRN)
Orientador

_____

Prof. Dr. Marcilio Carlos Pereira de Souto (UFRN)
Vice-Coordenador do Programa

Banca Examinadora:

_____

Profª. Drª. Thaís Vasconcelos Batista (UFRN)
Presidente

_____

Profª. Drª. Flávia Coimbra Delicato (UFRN)

_____

Prof. Dr. Paulo Pires (UFRN)

_____

Prof. Dr. Renato Cerqueira (PUC-Rio)

AGRADECIMENTOS

Dedico meus sinceros agradecimentos:

A Deus pelo dom da minha vida.

À Professora Thaís, pela orientação e o incentivo

Aos que sempre estiveram ao meu lado: Minha mãe Fátima, meu pai Martiniano, meu irmão Diógenes e meus queridos amigos.

Resumo

As plataformas de middlewares têm sido amplamente utilizadas como infra-estrutura subjacente para o desenvolvimento de sistemas distribuídos. Elas fornecem transparência de localização e de heterogeneidade e um conjunto de serviços que facilitam a construção de aplicações distribuídas. Atualmente, os middlewares acomodam uma variedade crescente de requisitos a fim de atender aos mais variados domínios de aplicação. Essa ampla variedade de requisitos procovou um aumento na complexidade dos middleware, devido a introdução de vários conceitos transversais na arquitetura. Tais conceitos não são modularizados adequadamente pelas técnicas de programação tradicionais, resultando no espalhamento e entrelaçamento desses conceitos no código do middleware. A presença de conceitos transversais limitam a escalabilidade do middleware. O paradigma orientado a aspectos tem sido utilizado com sucesso para melhorar a extensibilidade, a modularidade e a capacidade de personalização das plataformas de middleware através da separação dos conceitos transversais do código base da middleware.

Este trabalho apresenta o AO-OiL, uma arquitetura de um middleware orientado a aspectos (OA) dinamicamente adaptável, baseado na arquitetura de referência para middlewares OA. O AO-OiL consiste em uma refatoração orientada a aspectos do middleware OiL (Orb in Lua) para separação de conceitos básicos e conceitos transversais e segue a filosofia de que as funcionalidades do middleware devem ser ditadas pelos requisitos da aplicação. A arquitetura proposta foi implementada em Lua e RE-AspectLua. Para avaliar o impacto da refatoração na arquitetura, esse trabalho apresenta uma análise comparativa de desempenho entre o AO-OiL e o OiL.

Área de Concentração: Sistemas Distribuídos
Palavras-chave: Aspectos, Middleware, Arquitetura de Referência para Middlewares Orientados a Aspectos, RE-AspectLua.

Abstract

Middleware platforms have been widely used as an underlying infrastructure to the development of distributed applications. They provide distribution and heterogeneity transparency and a set of services that ease the construction of distributed applications. Nowadays, the middlewares accommodate an increasing variety of requirements to satisfy distinct application domains. This broad range of application requirements increases the complexity of the middleware, due to the introduction of many cross-cutting concerns in the architecture, which are not properly modularized by traditional programming techniques, resulting in a tangling and spread of theses concerns in the middleware code. The presence of these cross-cutting concerns limits the middleware scalability and aspect-oriented paradigm has been used successfully to improve the modularity, extensibility and customization capabilities of middleware.

This work presents AO-OiL, an aspect-oriented (AO) middleware architecture, based on the AO middleware reference architecture. This middleware follows the philosophy that the middleware functionalities must be driven by the application requirements. AO-OiL consists in an AO refactoring of the OiL (Orb in Lua) middleware in order to separate basic and crosscutting concerns. The proposed architecture was implemented in Lua and RE-AspectLua. To evaluate the refactoring impact in the middleware architecture, this paper presents a comparative analysis of performance between AO-OiL and OiL.

Area of Concentration: Distributed Systems
Key words: Aspect, Middleware, Aspect-Oriented Middleware Reference Architecture, RE-AspectLua.

Sumário

Lista de Figuras

Lista de Listagens

Lista de Tabelas

Capítulo 1

Introdução

O avanço das tecnologias de rede e sua maior difusão entre empresas e pessoas fez com que a demanda por softwares distribuídos aumentasse de maneira avassaladora em diversas áreas e com diversos propósitos. O desenvolvimento de sistemas distribuídos envolve um expressivo nível de complexidade, uma vez que tais sistemas têm que lidar com uma grande variedade de plataformas de hardware e de software. Aliado à isso, a contínua evolução tecnológica tem demandado por um aumento de produtividade e influenciado no aumento da complexidade dos requisitos dos sistemas. Este cenário impulsionou o surgimento dos sistemas de middleware [Bernstein, 1996] para dar suporte ao desenvolvimento de sistemas distribuídos.

Um middleware pode ser definido como uma camada de abstração situada entre as aplicações e o sistema operacional que provê facilidades de alto nível para o desenvolvimento de aplicações distribuídas [Bernstein, 1996]. Os sistemas de middleware oferecem transparência de heterogeneidade e distribuição, visando a interoperabilidade entre os sistemas.

Em geral, os sistemas de middleware são implementados utilizando o paradigma de orientação a objetos, e oferecem aos desenvolvedores um modelo de programação que combina a orientação a objetos com a computação distribuída. Além disso, os sistemas de middleware geralmente embutem em sua arquitetura um conjunto de serviços que encapsulam soluções para problemas recorrentes das aplicações distribuídas, por exemplo, serviço de nomes, de segurança, de transações, entre outros. O objetivo primordial é facilitar o desenvolvimento de sistemas distribuídos, deixando o programador da aplicação preocupado apenas com a lógica de negócio.

A primeira geração de sistemas de middleware enfatizava o conceito de transparência. Em computação, transparência significa que o desenvolvedor não precisa saber detalhes sobre o objeto que implementa determinado serviço. Essa abordagem ficou conhecida como *arquitetura caixa-preta*. Um sistema caixa-preta é um sistema no qual o usuário apenas pode utilizá-lo, não

sendo permitido alterar o comportamento ou a visão dos serviços oferecidos pelo mesmo.

Os sistemas de middleware passaram a acomodar uma variedade crescente de requisitos a fim de atender aos mais variados domínios de aplicação, como sistemas de tempo real, sistemas multimídias, sistema ubíquos, e de arquiteturas de hardware subjacentes, como PDA's, sistemas embarcados, estações de trabalho [Coulson et al., 2002b]. Entretanto, a estrutura rígida, monolítica e inflexível dessa geração provocou um aumento tanto na complexidade interna quanto na complexidade externa dos mesmos, comprometendo um dos seus principais objetivos - simplificar o desenvolvimento de sistemas distribuídos. A complexidade interna está relacionada com o desenvolvimento do middleware em si, enquanto que a complexidade externa está relacionada com a dificuldade de utilização do modelo de programação e dos serviços do middleware.

Para atender a essa crescente variedade de requisitos, os sistemas de middleware passaram a ter como desafios a necessidade de adaptar-se dinamicamente a mudanças do ambiente e de se adequar as mais variadas demandas. Nesse contexto surgiram os sistemas de *middleware de nova geração* [Coulson et al., 2002b] [Blair et al., 1998] [Kon et al., 2002] trazendo soluções para os problemas encontrados pelos sistemas de middleware de primeira geração. Essa nova geração baseia-se no *conceito de reflexão* computacional para suportar eficientemente a adaptação dinâmica, permitindo a construção de sistemas distribuídos flexíveis e reconfiguráveis dinamicamente. Chama-se de sistema computacional reflexivo, quando este é capaz de prover informações de si próprio e é capaz de atuar sobre ele mesmo, redefinindo o seu comportamento [Stankovic and Ramamritham, 1994]. Os sistemas reflexivos fornecem estruturas de dados que formam a auto-representação do sistema e refletem as entidades do domínio da aplicação e suas relações. A auto-representação e a implementação do sistema são causalmente conectadas, assim uma alteração na auto-representação é refletida na implementação e vice-versa.

Um middleware reflexivo é organizado como um conjunto de componentes que colaboram entre si, expondo a implementação de seus mecanismos subjacentes, ao contrário dos sistemas de middleware da primeira geração. Essa abertura dos detalhes internos permite a adição e a remoção de funcionalidades no middleware de forma mais direta e fácil. Os componentes de um middleware reflexivo pode ser classificado em dois níveis: nível base e nível meta. O primeiro contém os componentes que implementam os serviços do middleware em si, enquanto que o segundo contém os componentes que representam e modificam os componentes internos do middleware.

Em termos de middleware, o nível meta deve fornecer mecanismos para dar suporte a inspeção do comportamento interno e da estrutura da plataforma (introspecção), e para alterar o comportamento interno dos mecanismos subjacentes do middleware em tempo de execução (adapta-

ção). Como por exemplo, a troca de protocolos de transportes, QoS dentre outros serviços.

Em outras palavras, o nível base tem a responsabilidade de realizar computação com o objetivo de resolver problemas relacionados ao domínio do sistema, no caso o middleware. Enquanto que o objetivo do nível meta é realizar computação para resolver problemas relacionados a computação do nível base [Maes, 1987]. O meta nível pode ser implementado de duas maneiras complementares: reflexão estrutural e a reflexão comportamental. A reflexão estrutural expõe a estrutura interna de objetos e componentes com o intuito de permitir inspeções e alterações nessa estrutura. Por outro lado, a reflexão comportamental está relacionada à atividade dos mecanismos subjacentes. Ela permite o monitoramento e o controle das funções básicas do middleware.

Apesar da reflexão computacional juntamente com a orientação a objetos serem utilizadas como solução para adaptação dinâmica, não solucionam o problema da complexidade do middleware, pois os vários conceitos envolvidos na construção do middleware como persistência, transação, qualidade de serviço, segurança, comunicação, sincronismo, dentre outros, não são adequadamente modularizados através dos paradigmas convencionais de desenvolvimento. Esses conceitos, denominados de *conceitos transversais*, em geral, estão entrelaçados e espalhados pelo código do middleware em si, aumentando a complexidade, diminuindo a reusabilidade e dificultando o entendimento do código do middleware como um todo.

O desenvolvimento de software orientado a aspectos – DSOA - [Kiczales et al., 1997] é um paradigma de desenvolvimento que visa resolver o problema de códigos entrelaçados e espalhados em sistemas de software. A programação orientada a aspectos (POA) [Kiczales et al., 1997] permite a separação dos conceitos transversais através da introdução de uma nova abstração que os encapsula: o aspecto. Vários trabalhos [Putrycz and Bernard, 2002] [Cunha et al., 2006] [Rashid and Chitchyan, 2003] [Kiczales et al., 1997] indicam que os aspectos aumentam a modularidade, o reuso e a manutenibilidade do sistema. Outro benefício dessa técnica de programação é a melhora na evolução dos sistemas de software, uma vez que ela localiza mudanças em pontos determinados e diminui as dependências entre os vários aspectos que compõem o sistema como todo [Greenwood et al., 2008].

A POA tem sido utilizada para modularizar os vários interesses transversais presentes na construção dos sistemas de middleware [Jacobsen, 2001] [Cacho. et al., 2006] [Truyen et al., 2001], possibilitando que funcionalidades sejam inseridas de forma incremental e dinâmica e permitindo que os sistemas de middleware sejam personalizáveis pelo código da aplicação. A utilização dessa abordagem permite que o núcleo do middleware tenha apenas os serviços essenciais. As outras funcionalidades do middleware são representadas como aspectos. Esta abordagem torna o middleware mais simples de ser desenvolvido, mais flexível, mais leve e mais adaptável.

No entanto, as arquiteturas de middleware orientado a aspectos, em geral, têm sido especificadas de maneira *ad-hoc*. Cada plataforma contém um conjunto de soluções para tratar um mesmo problema: simplificar a complexidade dos sistemas de middleware. Entretanto, elas não seguem um vocabulário conceitual comum que permita comparações entre as várias tecnologias de middleware, pois não havia uma arquitetura de referência que servisse como modelo base para o desenvolvimento de sistemas de middleware orientados a aspectos (OA). Para resolver tal problema, em [Loughran et al., 2005] é apresentada uma proposta de arquitetura de referência para a construção de sistemas de middleware (OA). A utilização de um modelo de referência facilita o entendimento, a comparação entre os sistemas de middleware e permite agrupar problemas e soluções recorrentes na produção desse tipo de sistema.

A arquitetura de referência estabelece três níveis arquiteturais. O nível 1 define conceitos gerais, princípios e padrões, definindo abstrações que facilitem o desenvolvimento de sistemas de middleware OA. O nível 2 mapeia os conceitos transversais do middleware para as abstrações do nível 1. Ou seja, nesse nível os conceitos transversais são descritos através dos elementos definidos no nível 1. O nível 3 mapeia o nível 2 para uma plataforma de middleware específica.

1.1 Motivação

O principal propósito de um middleware é simplificar a construção de sistemas distribuídos em ambientes heterogêneos. Além da diversidade de ambiente, os sistemas de middleware possuem uma variedade de requisitos. Alguns requisitos estão ligados diretamente as funcionalidades dos middleware, enquanto que outros requisitos estão relacionados a propriedades dos sistemas em si que são muito importantes para qualidades do produto, como confiabilidade, facilidade de uso, desempenho dentre outros.

Um middleware pode ser utilizado por dois tipos de atores: os usuários e os desenvolvedores. Os usuários atuam apenas utilizando os serviços da plataforma para atender aos requisitos particulares de uma aplicação. Por sua vez, os desenvolvedores atuam personalizando a plataforma, adicionando novas funcionalidades ou configurando as existentes.

Entretanto, a variedade de ambientes e de requisitos aonde os sistemas de middleware devem funcionar provocou um aumento de ordem de magnitude considerável na complexidade interna e externa desses sistemas. Isso ocorreu devido a estrutura monolítica e inflexível da primeira geração dos sistemas de middleware. A necessidade de plataformas personalizáveis que sejam capazes de se adequar aos requisitos atuais das aplicações e aos requisitos que irão emergir no futuro próximo, tornou-se crucial para evitar o aumento em larga escala da complexidade do

middleware.

Esse aumento da complexidade limita a evolução, a personalização e a utilização dos sistemas de middleware. Portanto, a principal motivação desse trabalho é endereçar o problema do constante aumento da complexidade dos sistemas de middleware.

Nesse contexto, o desenvolvimento orientado a aspecto tem papel crucial. Os aspectos permitem a inserção de funcionalidades de maneira incremental ao mesmo tempo em que mantém essas funcionalidades tão independentes quanto possível, aumentando a modularidade do sistema. Cada uma das funcionalidades que são transversais pode ser implementada como um aspecto que irá compor com o código base do middleware em si, código relacionado as funcionalidades não transversais, formando uma instância do middleware que atenda aos requisitos do usuário.

Para dar suporte ao desenvolvimento de sistemas de middleware orientados a aspectos, foi proposto por Loughran et al [Loughran et al., 2005], uma arquitetura de referência para middleware orientado a aspectos. A formulação de um modelo de referência que reúna as melhores práticas de desenvolvimento é fundamental para a identificação e especificação dos principais conceitos envolvidos na construção desse tipo de sistema. Por esse motivo, os conceitos expostos na arquitetura de referência são genéricos. Entretanto, ainda não existe uma validação dessa arquitetura de referência, ou seja, não existem relatos na literatura de sistemas de middleware construídos com base nela. Sendo assim, a validade dos conceitos propostos precisam ser avaliados. Logo, é necessário verificar se os conceitos definidos são corretos e se eles são suficientes para expressar adequadamente um middleware.

O OiL (Orb in Lua) [Maia et al., 2005] é um middleware baseado em CORBA e, como tal, apresenta os problemas de entrelaçamento e espalhamento de código devido a presença de conceitos transversais como comunicação e *threads*. O OiL é um middleware que endereça adaptação dinâmica através de um modelo de classes e componentes dinamicamente adaptáveis. Tais estruturas suportam mudanças em suas definições em tempo de execução, ou seja, quando a definição de uma classe ou de um componente sofre uma alteração, essa alteração é propagada para todas as instâncias do componente ou da classe em questão. Esse middleware foi desenvolvido em Lua [Ierusalimschy et al., 2007] e tem o objetivo de ser simples, flexível, leve e portável, para que ele possa atuar em uma ampla gama de plataformas, de servidores a celulares. Portanto, a refatoração do OiL de acordo com a arquitetura de referência tem por objetivo transforma o OiL em um middleware orientado a aspectos, tornando sua arquitetura mais simples com um núcleo básico e com aspectos representando os elementos adicionais que não fazem parte das funcionalidades básicas. Além disso, o AO-OiL coloca em prática os conceitos definidos na arquitetura

de referência.

1.2 Objetivos

Esse trabalho visa refatorar a arquitetura do OiL para gerar uma nova arquitetura orientada a aspectos, o AO-OiL, seguindo as recomendações definidas na arquitetura de referência para middleware orientados a aspectos. Ou seja, um dos objetivos desse trabalho é validar a arquitetura de referência através da aplicação dos seus conceitos no AO-OiL.

Além do uso da arquitetura de referência, esse trabalho visa implementar o AO-OiL usando RE-AspectLua [Batista and Vieira, 2007], uma linguagem orientada a aspectos baseada em Lua cujo foco está no dinamismo, na reutilização de aspectos e na heterogeneidade das composições. Sendo assim, o RE-AspectLua encaixa-se adequadamente nos propósitos dessa pesquisa. Ele permite que a composição dos aspectos com o código da aplicação seja coordenada por arquivos de configuração escritos em Lua.

O AO-OiL tem como objetivo combinar um middleware dinâmico, OiL, e um processo de *weaving* dinâmico fornecido pelo RE-AspectLua, com o intuito de permitir que aspectos sejam inseridos dinamicamente conforme as necessidades da aplicação. Como resultado do trabalho pretende-se prover uma plataforma para o desenvolvimento de aplicações distribuídas de baixa complexidade e tamanho, com possibilidade de adaptação dinâmica.

Os objetivos específicos desse trabalho são:

- Uso da orientação a aspecto na refatoração do OiL e especificação da sua nova arquitetura. em conformidade com a arquitetura de referência para middleware OA.
- Instanciação em RE-AspectLua da linguagem de ponto de junção abstrata, APL, definida na arquitetura de referência.
- Implementação do AO-OiL em Re-AspectLua.
- Validação da arquitetura de referência.
- Implementação de um estudo de caso em OiL e AO-OiL com o intuito de avaliar, aproximadamente, o desempenho do middleware proposto.
- Avaliação do nível de separação de conceitos e do tamanho do middleware proposto através de um conjunto de métricas específicas para sistemas de middleware orientados a aspectos.

1.3 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma: O capítulo 2 apresenta os conceitos fundamentais para a compreensão desta pesquisa: programação orientada a aspectos, RE-AspectLua e middleware OiL. O capítulo 3 descreve em detalhes a arquitetura de referência para middleware orientado a aspectos. O capítulo 4 apresenta o mapeamento da arquitetura de referência para o OiL. O capítulo 5 apresenta a arquitetura e a implementação do *middleware* proposto nesse trabalho: o AO-OiL. O capítulo 6 apresenta uma avaliação do desempenho do AO-OiL bem como do nível de separação de conceitos e do tamanho do middleware proposto. Os trabalhos relacionados são apresentados no capítulo 7. Por fim, o capítulo 8 expõe as sugestões de trabalhos futuros e as considerações finais.

Capítulo 2

Conceitos Básicos

Esse capítulo tem como objetivo introduzir conceitos e princípios básicos necessários para a compreensão dos conceitos envolvidos no desenvolvimento do presente trabalho. Este capítulo está organizado da seguinte forma: a seção 2.1 apresenta o paradigma da programação orientada a aspectos e a linguagem RE-AspectLua, utilizada nesse trabalho. Como RE-AspectLua é uma extensão de AspectLua, também é exposto uma breve descrição da linguagem AspectLua e a motivação para a criação de RE-AspectLua. A seção 2.2 descreve em detalhes o OiL, middleware usado como base dessa pesquisa.

2.1 Programação Orientada a Aspectos

A Programação orientada a aspectos (POA) [Kiczales et al., 1997] é um paradigma de desenvolvimento de software que visa melhorar a modularização dos programas através da separação dos vários conceitos envolvidos na construção dos sistemas de software em dois tipos de unidades de código diferentes: os componentes e os aspectos. Os componentes são elementos responsáveis por encapsular conceitos que podem ser modularizados por procedimentos generalizados (objeto, método, procedimento ou API). Por sua vez, os aspectos são elementos que encapsulam conceitos transversais.

A motivação desse paradigma surge do fato de que as técnicas de programação atuais conseguem expressar muito bem o comportamento das entidades do sistema, mas falham em capturar interesses transversais. Além disso, alguns requisitos de software não são facilmente modularizados por objetos ou funções. Esses requisitos, denominados de *conceitos transversais*, tendem a ficar espalhados e entrelaçados com outros requisitos pelo código do sistema, dificultando a programação, a compreensão, a legibilidade e a manutenibilidade dos softwares.

Os *aspectos* são abstrações que modularizam os conceitos transversais melhorando a separação de conceitos no nível de implementação. Eles podem descrever cada conceito do sistema isoladamente. A POA define o processo de composição dos aspectos e dos componentes em uma aplicação, que é denominado *weaving*. Tal processo pode ocorrer em tempo de compilação ou tempo de execução, dependendo da plataforma de orientação a aspectos utilizada, e é realizado por uma ferramenta denominada de *Aspect Weaver*. O *Aspect Weaver* é um compilador ou um interpretador que agrega o código dos aspectos e o código dos componentes.

Existem várias implementações dos conceitos da orientação a aspectos e junto com essas implementações várias terminologias e elementos são encontrados. Esse trabalho utiliza a terminologia empregada pelo AspectJ [Kiczales et al., 1997], por ser uma linguagem precursora da POA e amplamente utilizada. Nessa terminologia, os aspectos são unidades que encapsulam conceitos transversais e descrevem pontos de junção (*join points*) e *advices*. Os pontos de junção são pontos do fluxo de execução de um programa onde os aspectos podem atuar. Por exemplo, chamadas de funções, leitura e escrita de variáveis, tratamento de exceções dentre outros. Um *advice* é uma ação que é aplicada quando um ponto de junção é atingido. O processo de composição, *weaving*, pode então ser definido como o processo de associação dos pontos de junção aos *advices* correspondentes.

Dentre as várias linguagens de orientação a aspectos existente, detalharemos RE-AspectLua [Batista and Vieira, 2007] por ser a linguagem utilizada nesse trabalho.

2.1.1 AspectLua

AspectLua [Cacho et al., 2005] é uma extensão da linguagem Lua que oferece suporte à programação orientada a aspecto. AspectLua é baseada nos conceitos de orientação a aspectos estabelecidos pelo AspectJ.

Lua é uma linguagem interpretada, tipada dinamicamente e com suporte ao gerenciamento automático de memória. Lua é caracterizada por ser centrada em tabelas, que são vetores associativos, e por possuir uma semântica extensível através de reflexão computacional.

AspectLua implementa as abstrações da POA através dos mecanismos reflexivos de Lua. A vantagem dessa abordagem é o fato de que o interpretador Lua padrão não precisa ser alterado. Porém, a implementação da POA através da reflexão exige um gerenciamento organizado e unificado dos mecanismos reflexivos e introspectivos, para que a coerência entre o nível base e o meta-nível seja mantida. O nível base é formado pelos elementos que representam os conceitos da aplicação, e o meta-nível é formado pelos elementos responsáveis pela reflexão. Entretanto, apesar de oferecer facilidades reflexivas, elas estão espalhadas em muitos mecanismos pela lin-

guagem, e um modo unificado para acessar essas facilidades é fundamental para o gerenciamento da consistência entre o nível base e o nível meta. Por isso, AspectLua utiliza o protocolo de meta-objetos denominado LuaMOP [Fernandes et al., 2004]. O LuaMOP é um protocolo de meta-objetos que provê uma camada de abstração sobre as funcionalidades reflexivas da linguagem Lua. Ele permite a criação de meta-elementos para cada elemento da linguagem Lua. Os meta-elementos oferecem um meio disciplinado para acessar e modificar a estrutura interna dos elementos da linguagem. Através desse protocolo, os aspectos podem gerenciar o nível base para que o comportamento desejado seja refletido no sistema. A Figura 2.1 exibe a arquitetura em camadas do AspectLua. A primeira camada é composta pelo interpretador Lua padrão e suas características reflexivas. A segunda camada é formada pelas abstrações do protocolo de meta-objetos LuaMOP. LuaMOP define um conjunto de meta-classes que permite a introdução dinâmica de aspectos definidos pela camada superior. No topo da arquitetura está o AspectLua que fornece mecanismos para a criação de aspectos de modo que os mecanismos reflexivos subjacentes ficam transparentes para o programador. Dessa maneira, os aspectos são definidos no meta-nível via os meta-objetos, e os componentes da aplicação são definidos via as abstrações tradicionais de Lua. Portanto, o processo de *weaving* consiste em combinar os dois níveis e é da competência do LuaMOP.

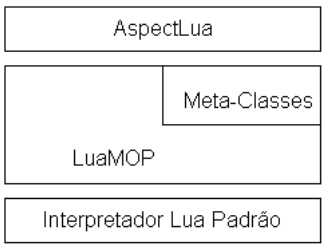


Figura 2.1: Arquitetura de AspectLua

Através do LuaMOP, AspectLua provê o seguinte conjunto de funcionalidades: (i) a inserção e remoção de aspectos em tempo de execução, ou seja, suporta a programação orientada a aspectos dinâmica, (ii) a definição da ordem de precedência entre os aspectos, (iii) *wildcards* e (iv) *anticipated join points*. O *anticipated join points* é um mecanismo que permite a interceptação de elementos ainda não declarados no programa. Dessa maneira, o *anticipated join points* possibilitam a carga dinâmica de bibliotecas e métodos não definidos pela aplicação, evitando a alocação de recursos desnecessários.

Assim como em AspectJ, os aspectos em AspectLua são definidos em termos de pontos de junção (*join points*) e *advices*. O conjunto de pontos de junção do AspectLua é constituído por

chamadas a métodos, *introductions* e acesso a campos e a tabelas (leitura e escrita). A Listagem 2.1 exibe a sintaxe de AspectLua para definir um aspecto genérico. Na linha 4, tem-se a criação de um aspecto em AspectLua, que ocorre por meio de invocações ao método *new()* da classe *Aspect*. Cada aspecto possui o método *aspect* que é responsável por definir o aspecto em questão. A linha 5 exibe tal método, que recebe como parâmetros as seguintes tabelas: nome do aspecto, o ponto de junção e o *advice*.

```
1  function fadvice()
2      --codigo aspectual
3  end
4  a = Aspect:new()
5  a:aspect( {name = "nome aspecto"},
6      {pointcutname = "nome do ponto de corte",
7      designator = "tipo do designador",
8      list = {"some-class.method1()","another-class.*"},
9      {type ="advice-type", action = fadvice} }
10 )
```

Listagem 2.1: Definição abstrata de um aspecto em AspectLua

O ponto de junção, exibido nas linhas de 6 a 8, é formado por um nome, um designador e pela lista de elementos que devem ser interceptados. O designador, expresso pelo campo *designator*, seleciona o tipo de ponto de junção. AspectLua suporta os seguintes tipos: *call* para invocações de métodos, *callone* para aspectos que devem executar apenas uma única vez, *introductions* para introduzir funções em objetos Lua e *get/set* para capturar leituras e escritas nas variáveis. A lista de elementos a serem interceptados é definida pelo campo *list*. Esse campo contém expressões de pontos de junção, ou seja, ele define um conjunto de variáveis e funções a serem interceptados.

Finalmente, na linha 9, o *advice* é definido pelo seu momento de atuação e pela ação a ser tomada quando o ponto de junção for atingido. O campo *type* define o momento no qual o aspecto atua. Um aspecto pode atuar antes, *before*, depois, *after*, ou durante, *around*, um ponto de junção. O campo *action* aponta para a ação a ser tomada. No caso da Listagem 2.1, a ação a ser tomada é o método *fadvice*, definido nas três primeiras linhas.

A Listagem 2.2 ilustra a criação de um aspecto utilizando AspectLua. As sete primeiras linhas definem a classe *Bank*, que é formada pelo elemento *balance* e pelo método *Bank.deposit* e *Bank.withdraw*. Em seguida, na linha 11, é possível observar a criação de um objeto aspecto. Na linha 12, o nome *log* é dado ao aspecto criado na linha anterior. Nas linha 13 a 15 tem-se a definição do ponto de junção *log_deposit*. Tal ponto de junção define que o aspecto *log* irá executar o método *log_advice* antes (*before*) de todas as chamadas do método *Bank.deposit*.

```
1  Bank = {balance = 0}
2  function Bank:deposit(amount)
3      self.balance = self.balance + amount
4  end
5  function Bank:withdraw(amount)
6      self.balance = self.balance - amount
7  end
8  function log_advice(a)
9    print('It was deposited: ' .. a)
10 end
11 asp = Aspect:new()
12 id = asp:aspect( {name = 'log'},
13     {pointcutname = 'log_deposit', designator = 'call',
14     list = {'Bank.deposit'}},{type = 'before',
15     action = log_advice} )
```

Listagem 2.2: Definição de um aspecto em AspectLua

O controle de precedência dos aspectos é efetuado pelas funções *getOrder* e *setOrder*. A primeira retorna a lista de aspectos associados a uma determinada função ou variável, enquanto que a segunda função permite alterar a ordem de execução dos aspectos. Ela recebe como parâmetros o nome da variável ou função e a nova ordem de execução.

Entretanto, apesar da programação orientada a aspectos endereçar a modularização dos interesses transversais, os aspectos definidos pela abordagem tradicional não são reutilizáveis. Isso acontece porque (i) o código do aspecto é fortemente associado com o elemento transversal sobre o qual ele atua, (ii) os aspectos estão acoplados a detalhes do código base que o programador é livre para alterar e (iii) o processo de composição dos aspectos com o código base é definido no código do aspecto, impedindo que o aspecto seja composto com diferentes elementos ou de diferentes maneiras.

AspectLua por ser baseado na POA tradicional, sofre dos problemas supra-citados. O código da Listagem 2.2 mostra que o ponto de junção *log_deposit*, definido nas linha 13 a 15, referencia diretamente a função *Bank.deposit* do elemento *Bank* pertencente ao código base da aplicação. Logo, se quiséssemos aplicar o método *log_advice* a outros elementos do código, teríamos que ou alterar o aspecto definido na Listagem 2.2 ou criar um novo aspecto.

2.1.2 RE-AspectLua

Para promover a reusabilidade dos aspectos e suportar a composição heterogênea, interfaces aspectuais, *pontos de junção abstratos* e uma linguagem de conexão para instanciação de *pontos de junção abstratos* foram adicionadas a AspectLua gerando uma nova linguagem denominada RE-Aspectlua.

As *interfaces aspectuais* [Chavez et al., 2006] são contratos de funcionalidades que descrevem como o aspecto interage com os outros elementos do sistema. Elas são constituídas por um conjunto de refinamentos. Os refinamentos especificam um conjunto de *pontos de junção abstratos*, elementos que serão afetados pelos aspectos, e um conjunto de ações a serem tomadas quando tais pontos de junção são alcançados. Por exemplo, a Listagem 2.3 mostra a definição de uma interface aspectual genérica em RE-AspectLua. Tal listagem mostra que para criar uma interface em RE-AspectLua, basta invocar o método *new* passando como parâmetros o nome da interface e a lista de refinamentos da mesma.

```
ai = AspectInterface:new({name = 'nome da interfame'} , refinements = { '
   lista de refinamentos' })
```

Listagem 2.3: Sintaxe da criação de uma interface

Os *pontos de junção abstratos* são declarações de pontos de junção independentes do código base da aplicação. Eles provêem apenas um *label* ou *tag* que no futuro irá apontar para pontos de junção reais, logo o ponto de junção especificado nesse momento é genérico e independente de contexto. A Listagem 2.4 ilustra como um *ponto de junção abstrato* é definido em RE-AspectLua. Um *ponto de junção abstrato* é uma tabela que contém um campo denominado *refine* e um campo denominado *action*. O primeiro é uma *string* que serve como identificador do *ponto de junção abstrato* na interface. Essa *string* será transformada em um campo dos aspectos que implementam tal interface pois, no futuro, esse campo irá apontar para um ponto de junção real, em outras palavras, esse campo irá apontar para um ponto no fluxo de execução de um programa. O segundo campo é um ponteiro para a ação a ser tomada quando o *ponto de junção abstrato* for alcançado.

```
1 function fadvice()
2     --codigo aspectual
3 end
4 { refine = "ponto de corte abstrato", action = fadvice }
```

Listagem 2.4: Sintaxe para a definição de um ponto de junção abstrato

A linguagem de conexão é uma linguagem de *script* utilizada para expressar as relações entre aspectos e elementos bases. Nesse contexto, a linguagem de conexão é responsável por instanciar os pontos de junção abstratos, respeitando as interfaces aspectuais definidas, e definir a ordem de precedência entre os aspectos. Instanciar um ponto de junção abstrato significa conectar um aspecto a um ou mais componentes regulares.

A opção por utilizar uma linguagem de conexão para expressar tais relações em detrimento ao uso de conectores, é devido ao fato de que as linguagens de conexão oferecem maior flexibilidade e expressividade. As linguagens de conexão permitem expressar relações complexas tais como (i) remoção condicional de um aspecto na presença de outro aspecto, (ii) definição de protocolos aspectuais e (iii) definição de código de cola, *glue code*, para tratar erros durante a composição. Além disso, as linguagens de conexão não impõem um modelo pré-definido de programação como ocorrem com os conectores.

Em RE-AspectLua, um aspecto é um componente formado por um conjunto de interfaces. A Figura 2.2 ilustra o modelo de aspecto de RE-AspectLua, no qual um aspecto é formado por um conjunto de interfaces que por sua vez são compostas por um conjunto de refinamentos. Vale a pena ressaltar que os aspectos não estão associados ao código base da aplicação, o que favorece a reutilização dos aspectos.

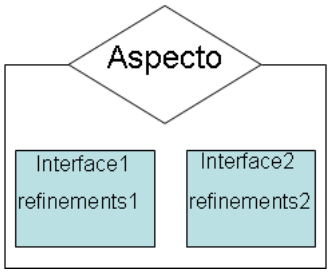


Figura 2.2: Modelo aspectual

Em RE-AspectLua, a criação de um aspecto é divida em duas fases: (i) especificação e (ii) instanciação. No primeiro momento, a etapa de especificação, os aspectos são definidos por um conjunto de interfaces que não fazem referências a nenhum ponto concreto do programa. Ao final da etapa de especificação, o aspecto ainda é abstrato, pois ele não faz referência a nenhum elemento concreto do código da aplicação. Portanto, ele não está habilitado a atuar.

No segundo momento, a instanciação, os pontos de junção abstratos definidos na primeira etapa são instanciados em pontos de junção concretos através da linguagem de conexão. A partir desse momento, o aspecto passa a atuar em um conjunto de pontos no fluxo de execução do programa.

A Figura 2.3 apresenta uma conexão aspectual. Os refinamentos da Figura 2.3 são conectados a pontos concretos da aplicação. O *refinements1* da interface *interface1* é associado ao *metodoA* da classe *ClasseA*. Similarmente, *refinements2* da interface aspectual *interface2* é associado aos métodos *metodoB* e *metodoC* da classe *ClasseB*.

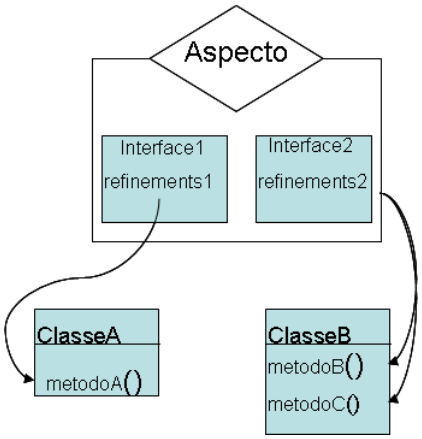


Figura 2.3: Composição de um aspecto em RE-AspectLua

A Listagem 2.5 mostra a definição de um aspecto abstrato em RE-AspectLua. O código dessa listagem corresponde a primeira etapa do processo de criação de um aspecto citado anteriormente: a especificação. Essa etapa ocorre na linha 6 através da criação da interface *ilog*. Ela determina que o método *log_advice*, definido na linhas 1 a 3, irá executar quando o conjunto de pontos de junção abstrato *log* for alcançado. Portanto, o objetivo desse trecho de código é especificar o comportamento da interface aspectual independentemente do código base da aplicação. Por fim, a linha 7 associa a interface aspectual *ilog* ao aspecto *aspect_log1* e a linha 8 associa a mesma interface ao aspecto *aspect_log2*.

```
1 function log_advice(a)
2   print('It was deposited: ' .. a)
3 end
4 aspect_log1 = Aspect:new()
5 aspect_log2 = Aspect:new()
6 ai_log = AspectInterface:new({name = 'ilog'},{ {refine = 'log',action =
    log_advice}})
7 aspect_log1:interface(ai_log)
8 aspect_log2:interface(ai_log)
```

Listagem 2.5: Definição de um aspecto em RE-AspectLua

Em RE-AspectLua, os aspectos podem compartilhar definições de interfaces. Dessa maneira, dois aspectos que tenham as mesmas interfaces aspectuais podem interagir de diferentes maneiras com a aplicação. Essa capacidade de RE-AspectLua é denominada composição heterogênea.

A Listagem 2.6, correspondente a segunda fase do processo de criação de um aspecto, exibe a instanciação de um aspecto que se utiliza da referida capacidade. O código dessa listagem define que o ponto de junção abstrato *log* da interface *ilog* irá atuar antes de todas as chamadas ao método *Bank.deposit* para o *aspect_log1*, e depois de todas as chamadas ao método *Bank.withdraw* para o *aspect_log2*. Dessa maneira, o código da Listagem 2.6 mostra como a interação heterogênea é expressa em RE-AspectLua.

```
1 aspect_log1.ilog["log"]= {designator = 'call',list = {'Bank.deposit'}, type=
    'before'}
2 aspect_log2.ilog["log"]= {designator = 'call',list = {'Bank.withdraw'}, type
    ='after'}
```

Listagem 2.6: Instanciação de um aspecto em RE-AspectLua

2.2 Orb in Lua - OiL

O OiL [Maia et al., 2005] (*Orb in Lua*) é uma implementação leve, portável e flexível de um ORB CORBA (*Common Request Broker Architecture*) de acordo com a especificação 3.0 da OMG [CORBA, 1999] desenvolvido nas linguagens Lua e C. A portabilidade do OiL advém do fato de que o interpretador Lua é inteiramente feito em ANSI C, logo é possível gerar interpretadores Lua que atuam desde microcontroladores a *mainframes*. A flexibilidade do ORB em questão também é diretamente suportada por Lua, pois ela é uma linguagem dinâmica que fornece mecanismos de extensão e reflexão sobre os quais o OiL constrói as suas abstrações. A infra-estrutura de distribuição do OiL é feita sobre a biblioteca LuaSocket [Nehab, 2004]. O LuaSocket é uma biblioteca programada em C e em Lua que fornece os protocolos TCP e UDP, e um conjunto de módulos com funcionalidades comuns das aplicações que lidam com a Internet, como DNS, FTP, HTTP, LTN12, MINE, SMTP, URL, etc.

A comunicação com o LuaSocket é feita através de objetos de entrada/saída que representam diferentes domínios, como o TCP e o UDP já implementados. Esses objetos de entrada/saída provêem uma interface padrão para operações de entrada e saída entre diferentes ambientes, diferentes sistemas operacionais e diferentes domínios. Todas as operações de entrada e saída são bloqueantes e *buferizadas* por padrão. Além disso, todas as aplicações podem ler *streams* de dados dos domínios linha por linha, bloco por bloco ou até que a conexão seja encerrada.

O OiL não faz uso de *stubs* ou *skeletons* estáticos, como fazem a maioria das tecnologias de middleware existentes. Todo o suporte à comunicação é criado em tempo de execução, incluindo as invocações e os despachos dos métodos [Maia et al., 2005]. Dessa forma, não é necessário utilizar um segundo compilador para gerar estaticamente os *stubs* e os *skeletons*. Ao invés dos *stubs*, o OiL utiliza *proxies* dinâmicos. Esses *proxies* são responsáveis por realizar as invocações de métodos de acordo com uma dada definição de interface. Dessa maneira, cada *proxy* está associado a uma particular interface de forma causal, e, portanto, sempre que tal interface muda, o *proxy* associado muda também.

A comunicação ocorre, essencialmente, através de mensagens escritas e lidas em canais de comunicação, descritos através de *sockets*, de acordo com o protocolo GIOP (*General Inter-ORB Protocol*). Para a produção das mensagens GIOP, é necessário conhecer as operações e os atributos de cada objeto. Com o objetivo de descrever essas informações de maneira uniforme e conhecida, o OiL provê suporte a IDL (*Interface Description Language*) padrão da OMG. Tal linguagem é baseada na linguagem de programação C++, entretanto ela é puramente declarativa e pode ser mapeada para qualquer linguagem de programação.

Além disso, o OiL oferece suporte à gerência automática de *servants*, que retira do programador a responsabilidade pela gerência do ciclo de vida da aplicação servidora, e à programação concorrente através do *multithread* cooperativo.

O *multithread* cooperativo é um modelo de programação concorrente baseado em co-rotinas. As co-rotinas são primitivas nativas oferecidas pela linguagem Lua para a implementação de concorrência. Elas são utilizadas para criar novos fluxos de execução independentes, comumente chamados de *threads*, com pontos de troca de contexto explícitos. Ou seja, o modelo de *multithread* cooperativo é um modelo de programação não preemptivo. Portanto, o programador é responsável por programar a política de escalonamento dos *threads*. As vantagens desse modelo são: todas as operações são atômicas por padrão, a execução é determinística, uma vez que o programador define onde e quando ocorrem as trocas de contexto, e os mecanismos de sincronização são mais simples. Por outro lado, a principal desvantagem das co-rotinas é que elas não tiram proveito das máquinas multiprocessadas. Particularmente no caso de Lua, as co-rotinas não interagem com o sistema operacional, pois elas são implementadas em nível de usuário, sendo assim existe uma *thread* de *kernel* para n co-rotinas.

O OiL é um middleware simples de ser utilizado. As Listagens 2.7 e 2.8 exibem um *Hello World* utilizando o OiL. Na Listagem 2.7 tem-se o código do servidor, que primeiramente cria um *servant*, linhas de 3 a 6, denominado *Hello* que contém a função *hello_world*. Em seguida, na linha 7, é invocada a função *oil.init* responsável pela inicialização do OiL. Nas linhas 8 a 13,

função *loadidl* é invocada para carregar a definição da interface no OiL.

```
1  require "oil"
2  oil.main(function()
3      local Hello = {}
4      function Hello:hello_world(name)
5          print("Hello World "..name.."!")
6      end
7      local orb = oil.init()
8      orb:loadidl[[
9          interface Hello {
10             void hello_world(in string n);
11         };
12     ]]
13     local servant = orb:newservant(Hello,nil,"Hello")
14     oil.writeto("hello.ref",orb:tostring(servant))
15     orb:run()
16 end)
```

Listagem 2.7: Código do servidor

Na linha 13, um *servant* é criado através da função *newservant*. O primeiro parâmetro dessa função é um objeto com a implementação do *servant*, por sua vez o segundo parâmetro é opcional, trata-se de uma chave identificadora do objeto definida pelo usuário. Por fim, o terceiro parâmetro possui o nome da interface do *servant*. Na linha 13, função *oil.writeto* é utilizada para armazenar a IOR do *servant* em um arquivo. A criação da IOR é feita invocando a função *oil.tostring*, passando a referência do *servant* como primeiro parâmetro e o nome do arquivo como segundo parâmetro da função. Finalmente, para iniciar o processamento das requisições pelo OiL, a função *oil.run* é invocada na linha 14.

Na Listagem 2.8, o código da aplicação cliente é apresentado. É possível notar nesse código que as invocações remotas são realizadas através do *proxy hello* definido na linha 4. A função *oil.newproxy* é utilizada para criar *proxies* a partir de referências textuais. Ainda na mesma linha está presente a função *oil.readfrom*, tal função é utilizada para ler de um arquivo. Após a criação do *proxy hello*, uma invocação remota a função *hello_world* é feita na linha 5.

O OiL visa ter poucas dependências estáticas, baixo acoplamento, ser reflexivo e configurável dinamicamente. Por esses motivos, uma estrutura monolítica não é adequada. Para prover um melhor suporte a personalização e a adaptação do ORB, o OiL foi projetado utilizando componentes e orientação a objetos. Entretanto, apesar da linguagem Lua não ser orientada a objetos, ela pode ser estendida para oferecer suporte a orientação a objetos sem a necessidade de alterar o

interpretador Lua padrão. O suporte ao modelo orientado a objeto do OiL é implementado pelo LOOP (*Lua Object-Oriented Programming*) [Maia, 2004], um modelo de orientação a objetos dinâmico.

```
1 require "oil"
2 oil.main(function()
3     local orb = oil.init()
4     local hello =orb:newproxy(oil.readfrom("hello.ref"))
5     hello:hello_world("OiL")
6 end)
```

Listagem 2.8: Código do cliente

2.2.1 LOOP- Lua Object-Oriented Programming

O LOOP é uma biblioteca escrita em Lua que fornece diferentes modelos de programação orientada a objetos para Lua. Ele disponibiliza modelos de classes, com suporte a herança simples e a herança múltipla, componentes, fábricas e *templates*.

O foco do LOOP está no dinamismo. Portanto, as classes e os componentes foram projetados para oferece mudanças em suas definições em tempo de execução. Quando a definição de uma classe ou componente sofre uma alteração, essa alteração é propagada para todas as instâncias do componente ou da classe em questão mantendo a consistência do sistema.

As classes do LOOP são implementadas através de meta-tabelas compostas apenas pelas operações disponíveis para os objetos de uma dada classe. Dessa maneira, as classes não contém nenhuma informação sobre seus atributos, como ocorre na programação orientada a objetos tradicional. As informações sobre os atributos residem nas instâncias das classes. Essa separação entre comportamento e estado permite criar mecanismos de adaptação que mudam a implementação dos sistemas sem que haja substituição de objetos ou transferência de estado entre objetos [Maia et al., 2005]. Por esse motivo, as classes do LOOP são chamadas de classes dinâmicas. O código na Listagem 2.9 mostra como é simples criar uma classe utilizando o LOOP.

A primeira linha do código da Listagem 2.9 cria uma classe denominada *Bank*. Tal classe define o campo *balance* e o método *deposit* respectivamente nas linhas 3 e nas linhas de 5 a 7. O campo *__init*, definido na linha 2, é uma rotina utilizada para inicializar apropriadamente uma instância da classe *Bank*. O *__init* atua como um meta-método utilizado para a criação de diferentes construtores para uma classe. Ele recebe a classe e os valores apropriados para a construção da mesma.

```
1 Bank = oo.class {}
2 function Bank:__init()
3     return oo.rawnew(self,{ balance = 0})
4 end
5 function Bank:deposit(n)
6     self.balance = self.balance + n
7 end
```

Listagem 2.9: Classe *Bank* utilizando o LOOP

O LOOP oferece os seguintes modelos de classes: *base*, *simple*, *multiple*, *cached* e *scoped*. O modelo *base* é o modelo base para todos os modelos de classe do LOOP, por isso ele é um modelo simples e define apenas classes sem suporte a herança. O modelo *simple* permite ao programador definir classes com suporte a herança simples. Já o modelo *multiple* permite a criação de classes com suporte a herança múltipla. Esses dois últimos modelos implementam a herança através de uma hierarquia de meta-tabelas. Entretanto, buscar na hierarquia de classes sempre que um campo é indexado pode provocar um *overhead* significativo em algumas aplicações. Felizmente, o LOOP oferece o modelo *cached* que evita tais buscas. As classes desse modelo copiam os campos definidos nas superclasses dentro delas mesmas. Porém, as classes pertencentes a esse modelo são manipuladas através de *proxies* que interceptam todas as mudanças nas cópias dos campos herdados. Esses *proxies* tornam os acessos as operações das classes mais custosos, mas são essenciais na atualização dos campos herdados por esta classe, quando alguma de suas superclasse sofre uma alteração. O último modelo, denominado de *scoped*, implementa a noção de escopo privado e protegido para os métodos de uma classe.

Por sua vez, o modelo de componentes do LOOP é composto por dois tipos de pacotes: *os pacotes de componentes* e os *pacotes de portas*. Os primeiros fornecem uma API para a criação de componentes. Os pacotes de portas fornecem diferentes implementações de portas que são utilizadas para conectar os componentes. O modelo de componentes definido pelo LOOP estabelece dois tipos de portas, as facetas (*facets*) e os receptáculos (*receptacles*). As facetas publicam os serviços de um componente através de suas interfaces, enquanto que os receptáculos descrevem funcionalidades requisitadas por um componente. A Figura 2.4 ilustra como é a estrutura de um componente do LOOP.

As facetas são implementadas através de tabelas, denominadas objetos facetas, que fornecem os valores e os métodos definidos por elas. Por outro lado, os receptáculos são implementados como campos que contêm referências para as tabelas que implementam as funcionalidades requisitadas. A Listagem 2.10 mostra a especificação de dois componentes: um denominado

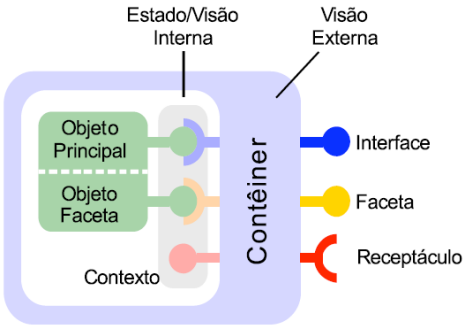


Figura 2.4: Componente do LOOP

BrokerTemplate e outro denominado de *ProxyTemplate*. O primeiro componente é composto por uma faceta, *broker*, e dois receptáculos, *proxies* e *references*. E o segundo componente é composto por uma única faceta denominada de *proxies*.

Os receptáculos podem ser de dois tipos: simples ou múltiplos. Um receptáculo é dito simples se este permite apenas uma faceta ligada nele. Os receptáculos múltiplos são aqueles que se ligam a mais de uma faceta, portanto eles se conectam a mais de um componente. Tal modelo de receptáculo oferece três especializações: *ListReceptacle*, *HashReceptacle* e *SetReceptacle*. No *ListReceptacle*, os componentes ligados não possuem um identificador explícito. Por outro lado, o *HashReceptacle* fornece identificadores explícitos para os componentes conectados aos receptáculos. Finalmente, o *SetReceptacle* é um receptáculo que se comporta do mesmo jeito do *ListReceptacle*, entretanto não permite componentes repetidos.

Ainda na Figura 2.4, é possível ver um elemento ainda não comentado, o elemento interface. O elemento interface provê uma acesso direto ao objeto que implementa as funcionalidades do componente.

Os pacotes de componentes do LOOP oferecem vários tipos de componentes com diferentes características. E como foi dito anteriormente, o foco do OiL está no dinamismo. Por esse motivo, o LOOP define um tipo especial de componente: *Dynamic*. Eles são componentes que suportam adição e remoção de portas em tempo de execução. Quando uma nova faceta é adicionada, é possível definir que a implementação dessa faceta será instanciada para cada componente sobre demanda. Ou seja, a criação dessa porta será realizada no momento do primeiro acesso a tal porta pela instância do componente.

Além dos componentes dinâmicos, o LOOP provê o seguinte conjunto de pacotes de componentes: *base*, *contained* e *wrapped*. O pacote *base* é a implementação mais leve do modelo de componentes. Ele utiliza o próprio objeto de implementação para armazenar as interfaces

do componente. O pacote *contained* contém o modelo padrão de componentes do LOOP. Os componentes do tipo *contained* criam objetos adicionais para armazenar as interfaces externas do componente, facetas e receptáculos, e o objeto de contexto, que contém as interfaces internas dos componentes. E por último, o pacote *wrapped* cria apenas um objeto adicional para representar as interfaces externas, mas usa o próprio objeto de implementação do componente como o objeto de contexto. O objeto de contexto é uma tabela que mapeia cada nome de porta para a sua implementação.

Finalizando as principais abstrações do modelo de componente do LOOP, tem-se os *templates* e as fábricas. Os *templates* são objetos que mapeiam os nomes das portas para os seus tipos específicos. O código na Listagem 2.10 ilustra a definição de um *template* para um componente. Ele cria dois *templates*, um *template* denominado *BrokerTemplate* com uma faceta batizada de *broker* e dois receptáculos nomeados de *proxies* e *references* e um *template* denominado *ProxyTemplate* com um receptáculo chamado *proxies*. Nas linhas 11 e 12 instâncias desses *templates* são criados através de fábricas. Por fim, na linha 14, as instâncias criadas são ligadas, ou seja, a interface de um componente é associada ao receptáculo do outro componente.

```
1  local BrokerTemplate = component.Template {
2      broker = port.Facet ,
3      proxies = port.Receptacle ,
4      references = port.Receptacle ,
5  }
6  local ProxyTemplate = component.Template{
7      proxies = port.Facet,
8  }
9  BrokerTemplateFactory = BrokerTemplate{broker = ServerBroker}
10 ProxyTemplateFactory = ProxyTemplate{proxies = ObjectProxies}
11 local Broker = BrokerTemplateFactory()
12 local Proxy = ProxyTemplateFactory()
13 --processo de binding
14 Broker.proxies = Proxy.proxies
```

Listagem 2.10: Criando um *template* para um componente

Uma vez que um *template* é definido pode-se utilizá-los para criar fábricas de componentes que seguem aquele dado *template*. Para criar uma fábrica, usa-se um componente *template* como construtor. Tal construtor recebe como parâmetro uma tabela que contém o construtor de cada faceta do componente a ser gerado. O código da Listagem 2.11 mostra a criação de um componente através de uma fábrica implementada a partir de um *template* definido na Listagem 2.10.

A primeira linha do código da Listagem 2.11 instancia um classe nomeada de *BrokerClass*. Essa classe define o método *object*, exibido nas linhas 2 a 17, que é responsável por criar novos *servants*. Na linha 18 é criado uma fábrica de componentes através do construtor do *template BrokerTemplate*. A listagem em questão mostra que o construtor de *BrokerFactory* recebe a tabela *BrokerClass* e a atribui a faceta *broker* do componente. Por fim, a linha 19 demonstra como é simples utilizar uma fábrica para instanciar componentes.

Nos sistemas baseados em componentes, a funcionalidade do sistema é atingida através da utilização de componentes que cooperam entre si. Logo, além de instanciá-los, é necessário conectá-los. O processo de criação, como mostrado anteriormente, é feito através das fábricas e dos *templates*. O processo de conexão dos componentes, denominado de *binding*, é feito através de uma simples atribuição. A linha 14 da Listagem 2.10 mostra o processo de *binding* do OiL.

```
1  BrokerClass = oo.class{ context = false }
2  function BrokerClass:object(object, key)
3      local context = self.context
4      key = key or "\0"..self:hashof(object)
5      local result, except = context.objects:register(object, key)
6      if result then
7          object = result
8          result, except = context.references:referenceto(key,self.config)
9          if result then
10             rawset(object, "__reference", result)
11             result = object
12         else
13             context.objects:unregister(key)
14         end
15     end
16     return result, except
17 end
18
19 BrokerFactory = BrokerTemplate{ broker = BrokerClass }
20 local broker = BrokerFactory()
```

Listagem 2.11: Definição de uma fábrica de componentes

2.2.2 A Arquitetura do OiL

A arquitetura do OiL é mostrada na Figura 2.5. Essa arquitetura é composta por duas sub-arquiteturas: a arquitetura do servidor e a arquitetura do cliente. A primeira gerencia os ciclos de

vida dos *servants*, as conexões, interfaces expostas pelos *servants* e os objetos que implementam as interfaces expostas. Por outro lado, a arquitetura do cliente tem como finalidade fornecer meios para a recuperação de referências para os *servants*, bem como permitir que as invocações de operações nos *servants* ocorram de maneira que a localização deles seja transparente para quem os utiliza.

A Figura 2.6 mostra como é a estrutura do OiL do lado do servidor. Nessa figura podemos destacar os seguintes elementos: *Server Broker*, *Request Dispacher*, *Request Receiver*, *Object Referrer* e o *Request Listener*. O *Request Receiver* controla a aceitação das requisições, o acesso aos canais de comunicação e notifica o *Request Dispacher* que existem requisições pendentes. O *Request Receiver* possui o receptáculo *listener*, que é utilizado para a criação de canais de comunicação e para obtenção de requisições, e o receptáculo *dispatcher* permite requisitar o despacho das requisições para seus respectivos *servants*. Além disso, esse componente possui a faceta *acceptor* que disponibiliza funções para a criação de canais de comunicação e para gerência da execução do próprio *Request Receiver*. Em geral, esse componente atua de maneira serial, ou seja, *Request Receiver* obtém uma requisição do canal e a envia para o despacho. O atendimento de requisições de forma concorrente pode ser feito utilizando o módulo *cooperative* da arquitetura, apresentado na seção 2.2.4. Tal módulo adiciona um escalonador ao OiL. O *Request Dispacher* mantém o repositório de referências para os objetos exportados pelo OiL, além de realizar os despachos dos métodos. Esse componente possui duas facetas: *objects* e *dispatcher*. A faceta *objects* registra cada objeto, associando um identificador único a cada um deles. O objeto registrado deve ser uma construção Lua que contém todas as funções exportadas por esse objeto na sua interface, caso ela exista. Por sua vez, a faceta *dispatcher* tem a finalidade de recuperar a referência de um objeto a partir de seu identificador. Isso é feito por meio da função *dispatcher* da faceta em questão. Por fim, o *Server Broker* é responsável por registrar novos *servants* e controlar o processamento dos mesmos. Ele exporta a faceta *broker* que é a interface de controle do servidor, oferecendo as operações de inicialização, execução, desligamento, suspensão dentre outras. O *Server Broker* tem dois receptáculos: *references* e *acceptor*.

Além desses componentes, o servidor conta ainda com dois outros componentes: o *Object Referrer* e o *Request Listener*. Respectivamente, um codifica as referências para objetos em referências na forma textual, enquanto que o outro tem por finalidade escrever e escutar nos canais de comunicação.

O ciclo de vida do servidor resume-se a um conjunto de passos para registrar um objeto no middleware e ficar aguardando a chegada de chamadas remotas para tais objetos. Os passos necessários para realizar o registro de um objeto são enumerados a seguir:

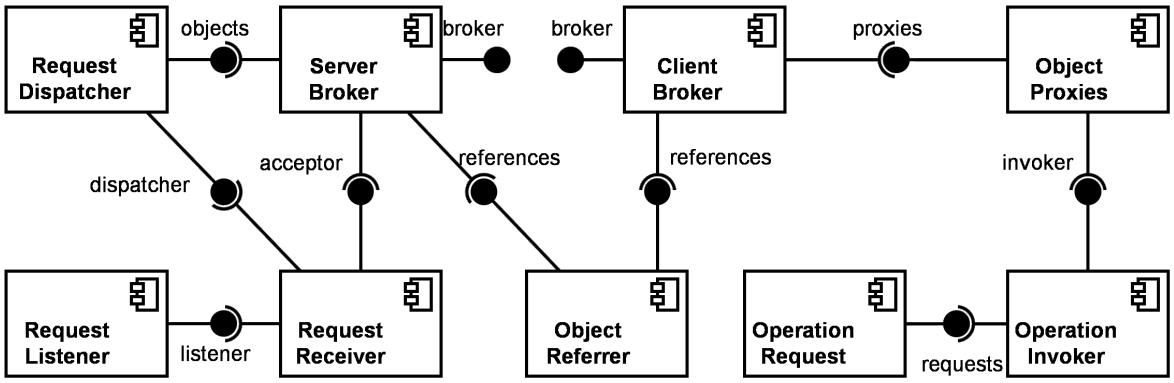


Figura 2.5: Arquitetura geral

1. Inicialmente o *Server Broker* invoca a função *init* para inicializar a instância do ORB em questão. Essa função recebe uma tabela com os valores de configuração de cada protocolo. Em geral, essa tabela contém o *host* e porta na qual o servidor deve escutar.

2. Para registrar a implementação de um objeto no OiL, utiliza-se a faceta *broker* do componente *Server Broker*, passando a implementação do objeto.

3. O *Server Broker* invoca a função *register* do *Request Dispatcher* para registrar o objeto e obter o *servant*. O *servant* é então disponibilizado para o desenvolvedor da aplicação.

4. O OiL assume que o protocolo de comunicação utilizado suporte referências codificadas que contêm a informação necessária para que um cliente possa contactar um objeto remoto. Essas referências são criadas pelo *Server Broker* através da função *encode* do receptáculo *references*. O *object referrer* codifica a referência e a devolve ao desenvolvedor.

5. Por fim, para iniciar a execução do OiL, o desenvolvedor chama a função *run* da faceta *broker*. A função *run* chama *acceptall* do *Request Receiver* para ficar esperando passivamente por uma requisição.

A partir desse momento, o servidor fica passivamente aguardando uma requisição chegar. Quando uma requisição chega ao servidor, a seqüência de passos a seguir é executada:

1. Quando uma requisição chega, o *Resquet Receiver* notifica o *Request Listener* ligado à porta correspondente ao canal de comunicação, passando o canal de comunicação.

2. O *Request Listener* lê a requisição do canal de transporte e a decodifica. Em seguida, ele repassa as informações de volta ao *Resquet Receiver*.

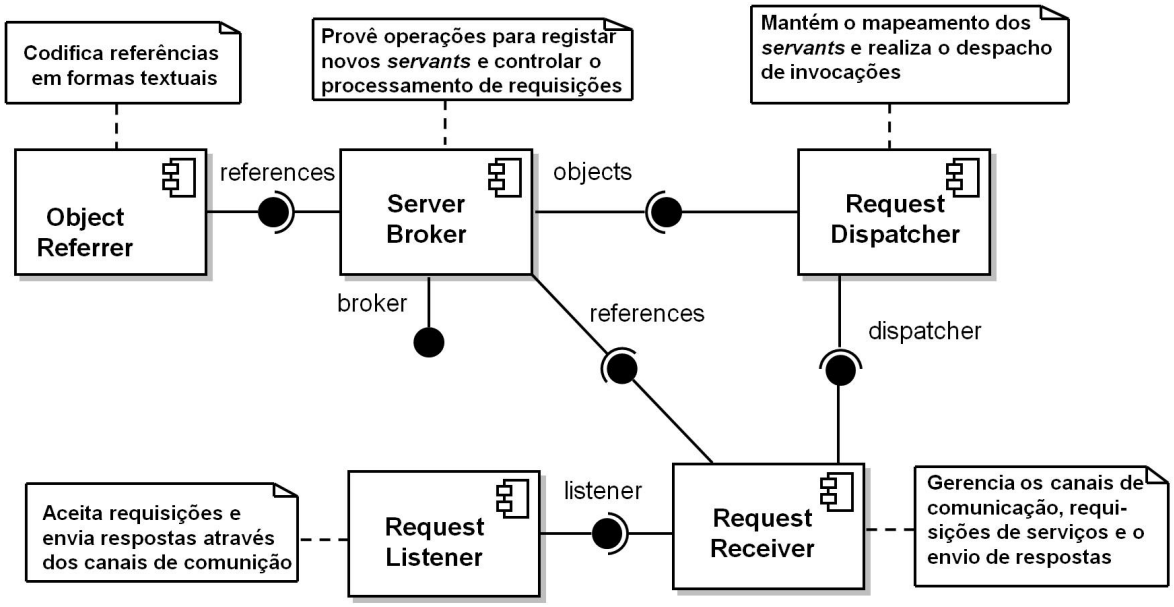


Figura 2.6: Arquitetura servidora

3. O *Request Receiver* de posse do canal de comunicação, do identificador do objeto *servant* e dos parâmetros, passa-os ao *Request Dispatcher* através da invocação da função *dispacher* do receptáculo *Dispatcher*.

4. O *Request Dispatcher* tem um mapa dos *servants* que associa o identificador do objeto à sua referência. Assim, ele pode invocar uma operação diretamente ou criar uma co-rotina para tal.

5. Após a execução da operação requisitada, o resultado é passado para o *Request Receiver* que invoca a função *reply* do *Request Listener*.

6. De posse do resultado da requisição, o OiL codifica e envia uma mensagem pelo canal de comunicação, através do *Request Listener*, finalizando a chamada remota.

Finalmente, a arquitetura do cliente, ilustrada na Figura 2.7, basicamente fornece meios para que uma aplicação cliente acesse e utilize os serviços disponibilizados pelos *servants*. Os principais elementos dessa arquitetura são: *Client Broker*, *Proxies*, *Operation Invoker*, *Object Referrer* e o *Operation Request*.

O *Client Broker* é o componente responsável por criar os objetos *proxies*, através do receptáculo *proxies*, a partir de informações extraídas das formas textuais pelo *Object Referrer* por

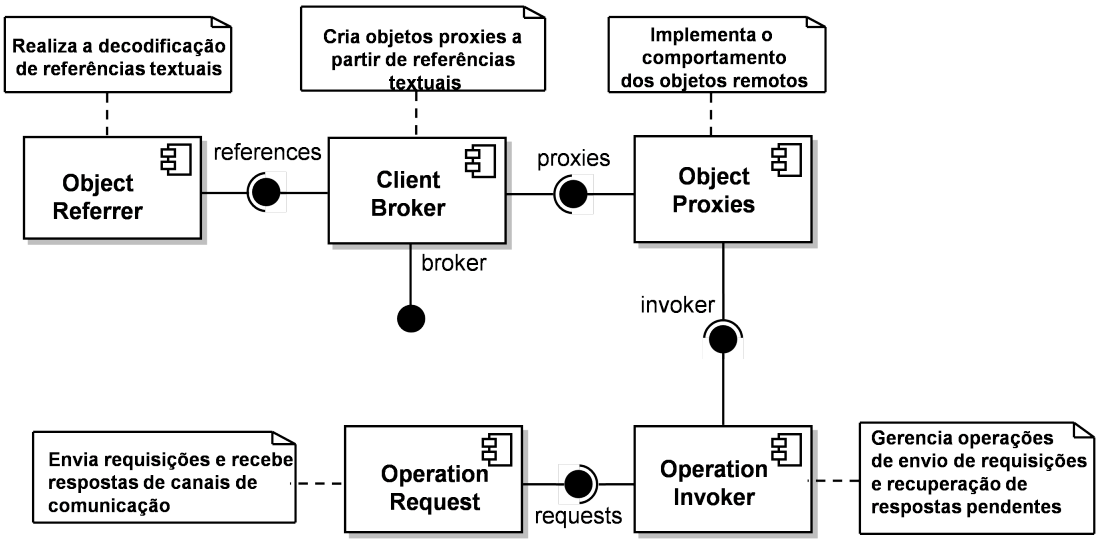


Figura 2.7: Arquitetura cliente

meio do receptáculo *references*. O componente *Object Proxies* é uma fábrica de objetos *proxy*. A função *proxyto* exportada pela faceta *proxies* do componente em questão retorna um objeto *proxy* mediante a passagem de uma referência decodificada. O acesso de um cliente a um *servant* é feito através de requisições de operações executadas sobre o *proxy* associado ao *servant* em questão. Assim, quando uma invocação é feita sobre tal *proxy*, ele repassa a operação e a referência dele mesmo para o *Operation Invoker*. Esse irá se encarregar de estabelecer um canal de comunicação com o objeto *servant* que implementa o serviço solicitado pela aplicação cliente. É da competência do *Operation Invoker* gerenciar a acesso a rede, enviando requisições e recebendo respostas. Esse objeto possui uma faceta, *invoker*, que exporta o método *invoker* para realizar a chamada remota. Essa função recebe como parâmetro a referência para o objeto remoto a ser conectado, bem como os parâmetros da chamada em si. Por fim, o *Operation Request* é responsável por escrever e escutar nos canais de comunicação do lado do cliente. Ele exporta a faceta *requests* que disponibiliza os métodos para gerência dos canais de comunicação como *getchannel*, *request* e *getreplay*.

Os passos para a criação de um *proxy* e a realização de uma chamada remota do lado do cliente podem ser resumidos nos seguintes:

1. O cliente solicita ao *Client Broker* um *proxy* para o objeto remoto a partir de uma referência codificada para tal objeto.

2. O *Client Broker* solicita ao *Object Referrer* a decodificação da referência. Isso é feito por

meio do receptáculo *references*.

3. Dado que o *Client Broker* possui uma referência remota válida, ele solicita ao *Object Proxies* a criação de um *proxy*.

4. A partir de agora, o cliente possui um objeto *proxy* sobre o qual poderá realizar invocações remotas. Ao realizar uma inovação sobre o *proxy*, essa chamada é transformada em uma chamada à função *invoker* do *Operation Invoker*.

5. *Operation Invoker* codifica a mensagem a ser enviada para o objeto remoto. Então, ele utiliza o *Operation Requester* para obter um canal de comunicação através do qual possa enviar uma mensagem de requisição codificada.

6. Uma vez que a requisição é efetuada, o *proxy* pode verificar se os resultados da requisição estão disponíveis. Para isso, ele invoca o método *getreply* exposto pelo componente *Operation Requester*. Esse método pode atuar de maneira bloqueante ou não dependendo de um parâmetro adicional.

2.2.3 Construção e Modelagem da Arquitetura

O OiL oferece três tipos de arquiteturas por padrão: *base*, *typed* e *cooperative*. As duas últimas estendem a arquitetura básica, chamada *base*, para adicionar verificação de tipos e concorrência, respectivamente. Todas as arquiteturas do OiL são definidas em módulos. Os módulos são descritos através de um conjunto de *templates* e de uma função denominada *assemble* que recebe uma tabela contendo os componentes e estabelece conexões entre eles. A Listagem 2.12 exibe o módulo *base* que define a arquitetura apresentada na Figura 2.6.

As linhas 1 a 15 de tal listagem definem o conjunto de *templates* do módulo. Esse conjunto de *templates* define quantos componentes fazem parte da montagem. Já as linhas 16 a 23 implementam a função *assemble* que estabelece as conexões entres os componentes da arquitetura.

É possível notar que os módulos apenas definem um conjunto de esqueletos da arquitetura, sendo assim, é necessário que os componentes sejam preenchidos com objetos que implementem as suas infra-estruturas. Para tornar isso possível, o OiL utiliza o conceito de construtores de componentes para cada módulo instalado em sua arquitetura. Tais construtores são fábricas de componentes utilizadas pelo OiL para criar e inicializar subconjuntos da arquitetura do middleware. Os construtores definem uma função *create* que recebe uma tabela de instâncias de componentes e adiciona a essa tabela as instâncias dos componentes criados por ela. É impor-

tante ressaltar que um construtor de componente não pode criar um componente se já existe uma instância dele na tabela passada para a função *create*.

```
1  ServerBroker = component.Template{
2      broker = port.Facet,
3      objects = port.Receptacle,
4      acceptor = port.Receptacle,
5      references = port.Receptacle
6  }
7  RequestDispatcher = component.Template{
8      objects = port.Facet,
9      dispatcher = port.Facet
10 }
11 RequestReceiver = component.Template{
12     acceptor = port.Facet,
13     dispatcher = port.Receptacle,
14     listener = port.Receptacle
15 }
16 function assemble(components)
17     setfenv(1, components)
18     RequestReceiver.listener = RequestListener.listener
19     RequestReceiver.dispatcher = RequestDispatcher.dispatcher
20     ServerBroker.objects = RequestDispatcher.objects
21     ServerBroker.acceptor = RequestReceiver.acceptor
22     ServerBroker.references = ObjectReferrer.references
23 end
```

Listagem 2.12: Definição do módulo arquitetural *Base*

É evidente que, como os construtores são responsáveis apenas por subconjuntos da arquitetura, pode ser necessário executar uma seqüência deles para que o ORB seja construído.

Além disso, o projeto do middleware em questão exige que os construtores que redefinem uma implementação definida por outros construtores devem executar primeiro. Então, por exemplo, em uma instância do OiL concorrente e com verificação de tipos, os construtores dos módulos arquiteturais *typed* e *cooperative* devem executar antes do construtor do módulo base.

Com o objetivo de tornar a construção e a montagem do middleware uma tarefa simples e flexível, o OiL fornece a função *oil.builder.build* cuja a finalidade é de criar e montar os componentes para formar uma instância do OiL. Essa função recebe como parâmetros uma *string* e uma tabela. A *string* define a lista de módulos arquiteturais que estendem os módulos seguintes, e a tabela contém todos os componentes criados pelos construtores anteriores e que deve ser usada

para armazenar os componentes criados pelo atual construtor. Em outra palavras, essa função chama os construtores de componentes na ordem especificada pela *string* passada como parâmetro. Após chamar a seqüência de construtores, *build* invoca a função *assemble* de cada módulo arquitetural para realizar as associações entre os módulos. O diagrama de seqüência da Figura 2.8 resume esses passos. A Listagem 2.13 apresenta o código da função *oil.builder.build*.

```
1  function build(customization, built)
2      for name in customization:gmatch(NamePat) do
3          --[[VERBOSE]] verbose:built(true, "creating ",name," components")
4          local success, module = pcall(require, FactoryFrm:format(name))
5          if success then
6              built = module.create(built)
7          elseif not module:match("module '.-' not found:") then
8              --[[VERBOSE]] verbose:built(false)
9              error(module, 2)
10             --[[VERBOSE]] else verbose:built("unable to load builder for
                   architecture ",name)
11         end
12         --[[VERBOSE]] verbose:built(false)
13     end
14     for name in customization:gmatch(NamePat) do
15         --[[VERBOSE]] verbose:built(true, "assembling ",name," components")
16         local success, module = pcall(require, ArchFrm:format(name))
17         if success then
18             module.assemble(built)
19         elseif not module:match("module '.-' not found:") then
20             --[[VERBOSE]] verbose:built(false)
21             error(module, 2)
22             --[[VERBOSE]] else verbose:built("unable to load architecture
                   definition for ",name)
23         end
24         --[[VERBOSE]] verbose:built(false)
25     end
26     return built
27 end
```

Listagem 2.13: Código da função *build*

A função *build* apresentada na Listagem 2.13 pode ser divida em duas partes. A primeira parte é delimitada pelas linhas de 2 a 13. Nesse trecho, o OiL realiza o processamento (*parser*) da *string customization*, que contém a lista de módulos a serem instanciados pelo mesmo, e em

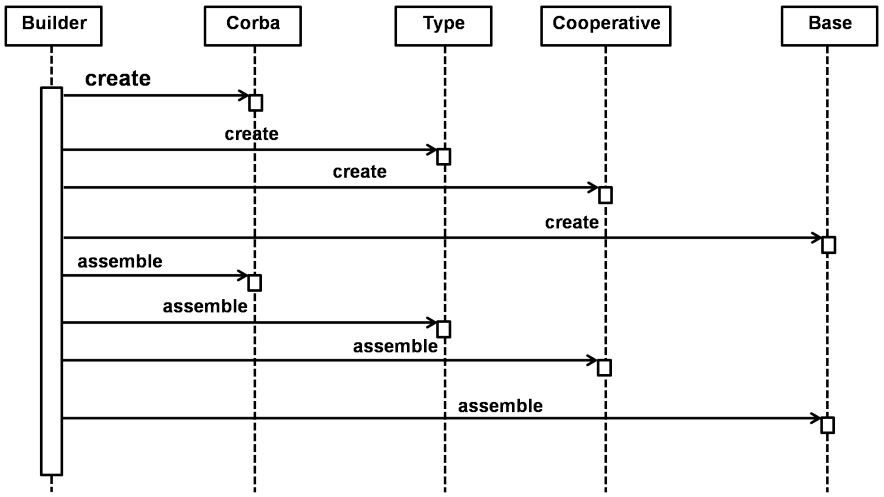


Figura 2.8: Diagrama de seqüência do módulo *builder*

seguida chama a função *create* de cada construtor passando a tabela de componentes *build*. Ao término dessa fase, todos os componentes da arquitetura estão criados dentro da tabela *build*, entretanto as ligações entre eles ainda não estão definidas. A fase 2 está programada nas linhas 14 a 25. Durante a segunda fase, as ligações entre os módulos são estabelecidas por meio de uma seqüência de invocações a função *assemble* de cada módulo arquitetural na ordem estabelecida pela *string customization*. Finalmente, uma instância do OiL é retorna na linha 26.

2.2.4 A Camada de Concorrência do OiL

Os mecanismos de suporte a programação concorrente em Lua apesar de fáceis, são de baixo-nível e exigem que políticas de escalonamento sejam implementadas pelas as aplicações que desejam utilizar a concorrência. Em ambientes distribuídos, isso obrigaria que toda aplicação servidora implementasse a sua política de escalonamento e gerenciasse a exclusão mútua nos canais de comunicação do servidor. Outra grande desvantagem é que as aplicações poderiam provocar *starvation*.

Para facilitar o desenvolvimento de aplicações *multithreads*, o OiL oferece suporte a programação concorrente através de uma camada de concorrência implementada pelo módulo *cooperative*. Essa camada de concorrência provê um escalonador, que seleciona qual requisição será atendida, e mecanismos de sincronização para garantir a exclusão mútua nos canais de comunicação. O escalonador evita que o programador seja obrigado a implementar uma política de escalonamento para toda aplicação residente no OiL.

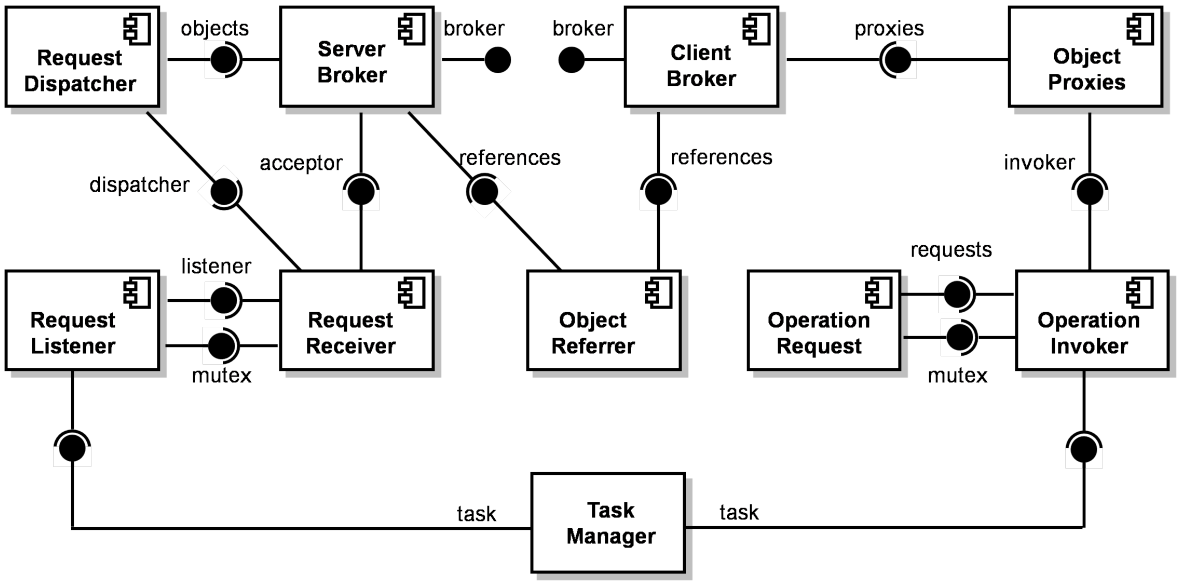


Figura 2.9: Arquitetura concorrente do OiL

A Figura 2.9 mostra como fica a arquitetura do OiL em um ambiente *multithread*. Observando a Figura 2.9 é possível notar a presença de dois novos elementos, o *Task Manager* e o *Mutex*. O *Task Manager* é o escalonador da arquitetura. Ele é responsável por gerenciar a execução das co-rotinas. Quando a arquitetura *cooperative* é utilizada, todo código executado pelo OiL deve estar em uma co-rotina registrada no escalonador. Por sua vez, os *mutexes* são utilizados para garantir a exclusão mútua nos canais de comunicação. A Listagem 2.14 mostra um exemplo de uma aplicação com suporte a concorrência.

Na linha 4, um objeto *proxy* é criado. Em seguida, nas linhas 4 a 10, a função *showprogress* é definida. A função *oil.newthread* na linha 13 cria e inicia a execução de uma nova co-rotina para executar a função *showprogress* passada como parâmetro. A *oil.newthread* inicia imediatamente a execução da nova co-rotina e a original co-rotina é suspensa e retornada quando o escalonador do OiL decidir por fazê-lo. Os outros argumentos passados para *oil.newthread* são parâmetros adicionais passados para a função que executará em uma nova *thread*.

O *Task Manager* exporta três facetas. A primeira, *tasks*, permite criar novas co-rotinas, passando para a elas a função a ser executada, e gerenciar suas atividades, através de uma API. A segunda faceta, *sockets*, exporta operações da biblioteca de LuaSocket [Nehab, 2004]. Por fim, a faceta *control* implementa as políticas de escalonamento, registro de co-rotinas, e os despacho para a execução e preempção das mesmas.

```
require "oil"
oil.main(function()
    local orb = oil.init()
    local proxy = orb:newproxy(assert(oil.readfrom("proxy.ior")))
    local function showprogress(id, time)
        print(id, "about to request work for "..time.." seconds")
        if proxy:request_work_for(time)
            then print(id, "result received successfully")
            else print(id, "got an unexpected result")
        end
    end
    for id, time in ipairs(arg) do
        oil.newthread(showprogress, id, tonumber(time))
    end
end)
```

Listagem 2.14: Aplicação OiL com suporte a concorrência

2.2.5 Verificação de Tipos no OiL

Em seu módulo básico, o OiL confia que o desenvolvedor sabe o que está fazendo, e , portanto, não realiza nenhum tipo de verificação dinâmica de tipos. Entretanto, como Lua não possui verificação estática de tipos, a verificação de tipos em tempo de execução ganha uma importância maior para o desenvolvedor. Por esse motivo, o OiL oferece um módulo de verificação de tipos.

Os protocolos que utilizam interfaces explícitas e bem definidas para expor as operações de um objeto remoto permitem que a verificação de tipo seja efetuada com base nessas interfaces. Isto é, a verificação baseia-se no número e tipo dos de parâmetros e de valores de retorno. No caso específico do protocolo CORBA, a codificação e decodificação dos dados trafegados pela rede é inteiramente ligada às interfaces, IDL's, das chamadas remotas. Por esse motivo, CORBA define um repositório de interfaces que é utilizado para verificar se a operação requisitada está de acordo com a interface publicada. Esse repositório permite o acesso às definições das interfaces em tempo de execução, através de um identificador único gerado por ele para cada tipo de IDL. Então, o compilador IDL atribui um identificador de repositório a todo tipo identificado na especificação da interface [Henning and Vinoski, 1999].

A arquitetura com suporte a verificação de tipos é apresentada na Figura 2.10. Ela possui três novos elementos: *Type Repository*, *Proxy Indexer* e o *Servant Indexer*. O componente *Type Repository* implementa o repositório de interfaces do OiL. Esse componente possui uma faceta

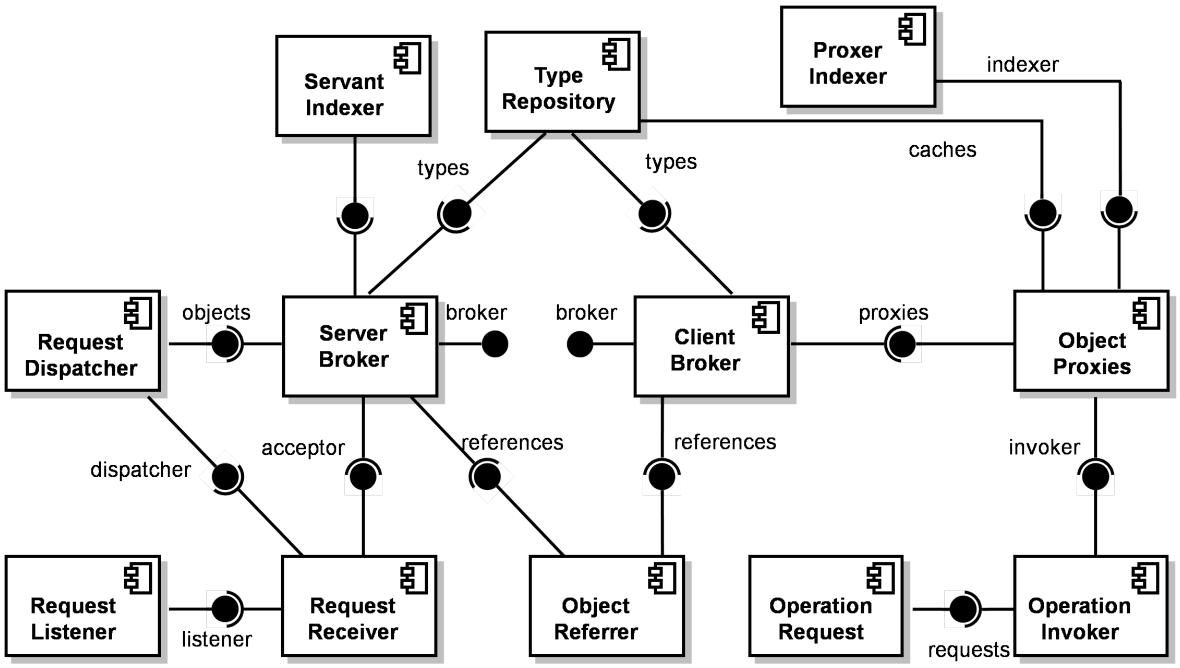


Figura 2.10: Arquitetura do OiL com suporte a verificação de tipos

e um receptáculo. A faceta, *type*, registra novas definições de interface no repositório e localiza interfaces através do identificador da interface ou de uma *string* contendo a especificação da interface. Quando um *servant* é criado, o identificador e a definição da sua interface devem ser passados para *Type Repository* para que a definição fique disponível no repositório. Nesse momento, a interface desse objeto é associada ao seu *servant*. Os componentes *Proxy Indexer* e *Servant Indexer* permitem que os *proxies* e os *servants*, respectivamente, possuam operações não definidas nas suas interfaces IDL, tais como as operações básicas de referências CORBA como *"_is_a"*, *"is_equivalent"*, *"_narrow"*,*"_interface"* e outras que estão na pseudo-interface *Object*. Essa interface é a superclasse de todas as interfaces [Henning and Vinoski, 1999].

Além disso, é possível notar na Figura 2.10 que o *Request Dispatcher* e o *Object Proxies* ganharam um novo receptáculo, *indexer*. Esse receptáculo tem por finalidade procurar pelas interfaces ligadas aos *servants* e aos *proxies* em questão durante as chamada remotas feitas a eles para que a verificação de tipos possa ser realizada. No caso dos *servants* para CORBA, o *indexer* verifica se a função a ser executada neste objeto é implementada por um objeto registrado no *Request Dispatcher* ou se é uma função do protocolo CORBA que pela especificação deveria estar em todos os objetos, como as funções básicas de referência citadas anteriormente.

2.2.6 Adaptação Dinâmica

O OiL suporta a adaptação dinâmica, ou seja, ele é capaz de mudar a implementação de uma aplicação sem ter que reiniciá-la ou pará-la. Assim, qualquer objeto criado com o OiL pode ter sua implementação substituída em qualquer momento. Como foi dito anteriormente, as classes do LOOP suportam essa característica através das classes dinâmicas. Elas permitem a alteração transparente de comportamento, i.e alterações nas implementações dos métodos. Entretanto, existem algumas situações que exigem mudanças nos estados dos objetos ou transferência de estados entre versões diferentes de uma aplicação. Nessas situações, é necessário realizar mudanças coordenadas no comportamento e no estado dos objetos. Entretanto, como não é possível alterar os atributos das instâncias de um objeto através das classes, o desenvolvedor deve lidar com esse problema diretamente.

Uma possível solução para esse problema seria implementar as mudanças em uma nova classe, que pode ser uma subclasse da original ou não. Então, a adaptação ocorreria sob demanda através de um processo de substituição dos antigos objetos pelos novos que seria disparado por invocações às operações destinadas a essa finalidade.

A direta programação do mecanismo acima tende a ser muito suscetível a erros por parte do desenvolvedor. Isso levou o OiL a fornecer uma abstração básica, um *guideline* para o desenvolvedor, de suporte a esse caso de adaptação dinâmica. Tal abstração na verdade consiste na definição de um componente adaptador cuja responsabilidade é encapsular a solução citada anteriormente. A seqüência de passos a ser seguida pelo desenvolvedor para implementar um componente adaptador é simples e é formado por quatro passos:

1. Definir a nova implementação das operações dos componentes em uma nova classe.

2. Definir a nova interface do componente.

3. Implementar uma função para adaptar o estado da instância.

4. Fornecer uma lista de operações que disparam o processo de adaptação.

A Listagem 2.15 mostra a definição da interface do componente *adaptador*. Ela é composta pela operação *apply_change* que recebe quatro parâmetros. O primeiro parâmetro é a seqüência de operações que disparam a adaptação. O segundo e o terceiro parâmetros são *strings* que contém a implementação dos códigos de adaptação dos estados e do comportamento respectivamente. Por fim, o último parâmetro fornece a nova definição da interface do componente.

```
interface ComponentAdaptor {
    void apply_change(
    in StringSequence triggers,
    in string state_adaptation_code,
    in string code_adaptation_code,
    in string new_interface_def
    ) raises (CompileError);
};
```

Listagem 2.15: IDL do componente *adaptador*

Quando uma mudança ocorre, a atual classe é modificada de modo que cada uma das operações que disparam a adaptação é trocada por uma operação que executa o código de adaptação do estado sobre a instância do objeto e então elas mudam a classe do objeto para a nova classe definida [Maia et al., 2005].

```
1  Adaptor = oo.class()
2  function Adaptor:__init(class)
3      return oo.rawnew(self, { class = class })
4  end
5  function Adaptor:apply_change(triggers, state, code, iface)
6      local adaptor, errmsg = loadstring("return function(self)
7          \n".. state .."\nend")
8      if not adaptor then
9          oil.assert.exception{ "IDL:CompileError:1.0",
10             message = errmsg, code = "function(self)\n"
11             .. state .."\nend",
12         }
13     end
14     adaptor = adaptor()
15     orb:loadidl(iface)
16     local updater, errmsg = loadstring(code)
17     updater()
18 end
```

Listagem 2.16: Implementação da IDL da Listagem 2.15

A Listagem 2.16 apresenta uma implementação para função *apply_change* da Listagem 2.15. Na linha 1, tem-se a criação da classe *Adaptor*. Nas linhas de 2 a 4, o construtor da classe *Adaptor*. A função *apply_change*, definida nas linhas de 5 a 18, realiza a adaptação dinâmica. Primeiramente, nas linhas de 6 a 12, ela carrega o código relacionado a adaptação dos estados

do objetos através da função *loadstring*. Em seguida, na linha 14, o código de adaptação do estado. Na linha 15, a nova IDL é carrega. Finalmente, o código que contém a adaptação do comportamento é carregado, linha 16, e executado, linha 17.

```
1  CentralServer = { users = {} }
2  function CentralServer:register (name, messenger )
3      self.users[name] = messenger
4  end
5  function CentralServer:unregister (name)
6      self.users[name] = nil
7  end
8  function CentralServer:isonline (name)
9      return self.users [name]
10 end
11 Messenger = {}
12 function Messenger:show(message) print (message) end
```

Listagem 2.17: Aplicação servidora

Por exemplo, considere uma aplicação de troca de mensagens entre usuários cadastrados em um servidor — retirada de [Maia et al., 2006]. A Listagem 2.17 ilustra o servidor dessa aplicação. A Linha 1 cria uma tabela que contém uma tabela *users* que armazena todos os usuários do sistema. As linhas de 3 a 7 descrevem métodos para gerenciar os usuários. O método *register* realiza o cadastro de um usuário e o método *unregister* retira um usuário do sistema. O método *isonline*, definido nas linhas 8 a 9, informa se um dado usuário está ativo. Finalmente, a linhas 11 e 12, descrevem o objeto *Messenger* que define como é a representação de uma mensagem no sistema.

Suponha que o desenvolvedor da aplicação deseje alterar a implementação da função *register* da classe *CentralServer*. Para isso, ele deve fornecer um *script* que utilize a classe *Adaptor* apresentada na Listagem 2.15.

A Listagem 2.18 mostra um *script* que altera o código da função *register* de *CentralService*. Nas linhas 2 a 7, tem-se a definição da nova versão da função *register*. Em seguida, o código relacionado a adaptação do estados do componente. Nas linhas 9 a 17, a definição da IDL da aplicação é provida. Nota-se que a IDL não foi alterada uma vez que apenas uma mudança no comportamento da função foi realizado.

```
Adaptor:apply_change({"register"},[[ ]],
    [[
        function CentralServer:register(name,messenger)
            print('CentralServer:register:',name)
            self.users[name] = messenger
        end
    ]],
    [[
        interface IMessenger{
            void show(in string msg);
        };
        interface IServer{
            void register(in string user, in IMessenger msgr);
            void unregister(in string name);
            IMessenger isonline(in string user);
        };
    ]]
)
```

Listagem 2.18: Aplicação de um mudança de comportamento

Capítulo 3

A Arquitetura de Referência

A arquitetura de referência para sistemas de middleware orientados a aspectos estabelece três níveis arquiteturais [Loughran et al., 2005] [Greenwood et al., 2007]. O nível 1 define conceitos gerais, princípios e padrões. O nível 2 mapeia os interesses transversais envolvidos na construção de sistemas de middlewares como distribuição, coordenação, persistência dentre outros, para as abstrações do nível 1. Por fim, o nível 3 mapeia o nível 2 para uma plataforma de middleware específica. Esse último nível tem por finalidade validar os conceitos da arquitetura de referência, através do mapeamento dos mesmos para as atuais sistemas de middlewares orientadas a aspectos.

Esse trabalho situa-se no nível 3 da arquitetura de referência. Ou seja, busca-se implementar os conceitos da arquitetura de referência em uma plataforma de middleware especifica. Portanto, é preciso definir mapeamentos dos níveis 1 e 2 da arquitetura de referência para o AO-OiL, arquitetura de middleware proposta nesse trabalho.

O restante desse capítulo está organizado da seguinte forma: A seção 3.1 apresenta em detalhes o nível 1 da arquitetura de referência. A seção 3.2 introduz o nível 2 e apresenta um esqueleto do mapeamento do interesse distribuição para as abstrações do nível 1. O nível 3 será abordado no capítulo 4.

3.1 Nível 1

O objetivo do nível 1 é fornecer uma lista de conceitos independentes de mecanismos de distribuição e modelos de orientação a aspectos. A finalidade desses conceitos é criar abstrações que facilitem o desenvolvimento de sistemas de middlewares orientadas a aspectos.

O nível 1 define os seguintes elementos: um *microkernel*, uma linguagem de definição de

pontos de junção denominada de APL (*abstract pointcut language*) e *padrões de binding*.

3.1.1 Microkernel

O *Microkernel* é a denominação dada à forma de design de sistemas operacionais no qual a quantidade de código que executa em modo protegido é mínima. O padrão de projeto *Microkernel* também segue essa mesma filosofia. Portanto, um *microkernel* pode ser definido como um componente que encapsula os serviços fundamentais de uma aplicação, incorpora extensões, componentes que implementam serviços externos, e coordena a comunicação entre elas. Além disso, o *microkernel* deve gerenciar os recursos utilizados pelas extensões e oferecer facilidades de comunicação para quem as utiliza [Schmidt et al., 2000].

Os objetivos do projeto do *microkernel* definido pela arquitetura de referência foram os seguintes: ser pequeno, extensível e adaptável, ter baixo acoplamento entre ele e os componentes do sistema, suportar carga dinâmica de componentes, suportar reconfiguração dinâmica sem que o sistema necessite parar, prover um modelo de programação de componentes simples e dar suporte aos aspectos para a implementação de funcionalidades do sistema como distribuição, coordenação, persistência etc. O *kernel* deve dar suporte a dois tipos de modelos para o desenvolvimento de serviços orientada a aspectos: *opt-in* e *co-opted*. No modelo *opt-in* um componente estende um ou mais aspectos abstratos para adequar um serviço para ele próprio. No modelo *co-opted*, um componente coordena as associações, *bindings*, de funcionalidades entre outros componentes. Os elementos arquiteturais utilizados na construção do *microkernel* foram os seguintes: contêiner, componentes, interfaces e *bindings*. Os contêineres representam recipientes para componentes, interfaces, *bindings* e aspectos. Componentes são as unidades funcionais que interagem com outros componentes via interfaces. As interfaces são utilizadas para associar componentes e podem ser classificadas em interface *Provided* e em interface *Required*. O primeiro tipo de interface expõe as funcionalidades de um componente, enquanto que o outro tipo expõe as dependências de um componente. Finalmente, os *bindings* são elementos que representam as associações entre interfaces *Provided* e *Required*. A Figura 3.1 [Greenwood et al., 2007] exibe cada elemento arquitetural.

A Figura 3.1 ilustra dois tipos de *bindings*: *Binding* e *AO-Binding*. O primeiro representa as associações normais, ou seja, associações entre componentes. O segundo representa as associações orientadas a aspectos. O *AO-Binding* permite que aspectos sejam interpostos entre componentes, tornando possível a aplicação de interesses transversais. Dessa maneira, diferentes composições podem ser estabelecidas sobre os *bindings* normais entre componentes. O *AO-Binding* é formado pelos seguintes elementos:

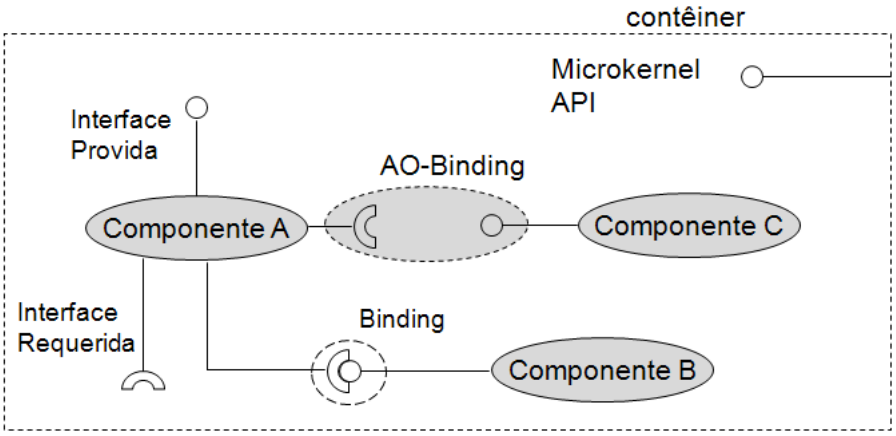


Figura 3.1: Elementos arquiteturais

- Padrões de pontos de junção: baseados na APL.
- Um elemento *advice* que especifica os componentes que serão interceptados, o escopo de instanciação e as operações a serem executadas.

O *microkernel* é um pequeno núcleo funcional composto pelos seguintes serviços básicos: registro de componentes, instanciação de componentes e estabelecimento de associações de componentes. Ele expõe a seguinte API:

- **LOAD/UNLOAD**: Gerenciam a carga e a descarga de componentes dentro de um contêiner.
- **START/STOP**: Inicia ou interrompe as operações de um componente.
- **BIND/UNBIND**: Cria ou destrói associações entre componentes.
- **INSTANTIATE/DESTRUCT**: Cria ou destrói instâncias de componentes.

3.1.2 A Linguagem de Definição de Pontos de Junção

A APL é dita abstrata porque apenas define uma notação cuja finalidade é estabelecer uma maneira de expressar pontos de junção de forma independente de plataformas de orientação a aspectos específicas. Em outras palavras, a APL é uma linguagem genérica que pode ser utilizada como um *framework* para a criação de linguagens de definição de pontos de junção concretas.

Além de ser genérica, a APL deve ser voltada para aplicações baseadas em componentes e deve ser capaz de definir pontos de junção distribuídos.

A principal função da APL é permitir a definição de padrões que agrupam um conjunto de pontos de junção sobre os quais atuarão um conjunto de aspectos. A APL é capaz de expressar duas famílias de padrões: padrões de tipos e padrões de assinatura. O primeiro realiza o casamento de padrão com base nos tipos envolvidos, enquanto que o segundo realiza o casamento com base nas assinaturas de métodos, campos e etc. A equação 3.1 mostra o casamento de padrões baseado em tipos. Ela seleciona todos pontos de junção dos componentes *c* que pertencem ao conjunto de componente *C*, $\forall c \in C$, cujo tipo é *org.gnu.stack, c.name = "org.gnu.stack*"*. Um exemplo de casamento de padrões baseado em assinatura é exibido na equação 3.2 . A expressão apresentada nessa listagem seleciona todos os ponto de junção para as operações cuja assinatura seja *void push(int)* e que estejam localizadas em uma interface *i* do tipo *Provided*, $i.category \in provided$, do componente *List*, *c.name="List"*.

$$\forall\, c \in C,\ c.name = \text{“}org.gnu.stack*\text{”} \tag{3.1}$$

$$\forall cp\ \in CP, \forall c \in C, \forall i \in c.interfaces, \forall o \in o.operations$$
$$c.name = \text{“}List\text{”} \wedge i.category \in \{provided\} \wedge o.signature = \text{“}void\, push(int)\text{”} \tag{3.2}$$

O modelo de pontos de junção da APL é simples, porém extensível, pois deve cobrir uma variedade de modelos de pontos de junção específicos. A arquitetura de referência define dois tipos básicos de pontos de junção: *call* e *execution*. A diferença entre eles está no local no qual o ponto de junção é interceptado. A Figura 3.2 esclarece essa diferença. Um ponto de junção é denominado de *call* quando ele ocorre em uma interface *Required* de um componente, enquanto que um ponto de junção é denominado de *execution* quando ocorre em uma interface *Provided* de um componente. Por exemplo, se um dado componente X invoca um método de um outro componente Y, esse ponto do fluxo de execução é um ponto de junção do tipo *call*. Por outro lado, quando algum outro componente invoca um método exposto por X, esse ponto de ponto de junção é classificado como *execution*. Vale ressaltar que essa classificação é relativa, ou seja, é preciso ter um referencial para que se possa classificar um ponto de junção. No caso do exemplo citado, esse referencial é o componente X.

Uma linguagem de definição de pontos de junção pode ser vista como uma linguagem específica de domínio [Van Deursen et al., 2000]. Ou seja, ela endereça um único problema que no

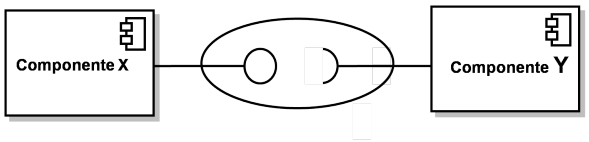


Figura 3.2: Classificação dos pontos de junção de acordo com a localização

caso é agrupar pontos de junção relacionados através de quantificadores e variáveis quantificadas. O domínio para uma aplicação baseada em componente definido pela APL é determinado pela seguinte tupla < CP; C; I ;O > onde: CP é um conjunto de contêineres sobre o qual a aplicação é instalada, C é o conjunto de componentes, I é o conjunto de interfaces e O é o conjunto de operações. Para cada elemento do domínio, a APL associa dois tipos de propriedades: estáticas e dinâmicas. As propriedades estáticas representam o conjunto de propriedades mínimas que devem existir no middleware. Por outro lado, as propriedades dinâmicas denotam um conjunto de propriedades que podem estar presentes ou não no middleware. Essas propriedades estendem o poder de expressividade da APL. Uma propriedade dinâmica é um par < chave; valor > associada a um elemento do domínio.

Uma expressão de ponto de junção na APL é composta por quatro sub-expressões de lógica de primeira ordem que selecionam os quatro elementos comentados no parágrafo anterior. Cada sub-expressão pode utilizar quantificadores universais, variáveis quantificadas, operadores booleanos e expressões de comparações sobre valores de propriedades através dos operadores $\in$ e $\notin$. Por exemplo, a Equação 3.3 apresenta um exemplo de uma expressão de ponto de corte descrita através da APL. Essa expressão captura todos os pontos de junção do tipo *call* associados aos métodos: (i) *object* pertencente ao componente *Server Broker* e (ii) *dispatch* pertencente ao componente *Request Dispatcher*.

$$\begin{aligned} &\forall cp \ \in CP, \forall c \in C, \forall i \in c.interfaces, \forall o \in o.operations \\ &(c.name = \text{“}ServerBroker'' \wedge o.signature = \text{“}object'') \vee \\ &(c.name = \text{“}RequestDispatcher'' \wedge o.signature = \text{“}dispatch'') \wedge call(o) \end{aligned} \tag{3.3}$$

Além de definir uma forma de expressar pontos de junção de maneira genérica, a APL contempla mecanismos para a passagem de informações contextuais de um ponto de junção para o *advice*. Na arquitetura de referência esse mecanismo é feito através de propriedades anexadas as invocações de funções, propriedades chamadas de *propriedades de contexto*. Existem duas operações definidas para cada *propriedade de contexto*: (i) Uma para atribuir um valor a uma

propriedade e (ii) uma para ler o valor de uma propriedade. Essas propriedades podem ser vistas como metadados.

As operações utilizam uma linguagem de descrição de interface, a *Interface Description Language*, para definir quais propriedades de contexto têm cada operação. A Listagem 3.1 mostra um exemplo dessa linguagem. Nessa listagem é definido que a operação *generic_method* tem a propriedade de contexto X.

```
1 [require("X")]
2 operation generic_method()
```

Listagem 3.1: Exemplo da linguagem de descrição de interface

A arquitetura de referência define o seguinte conjunto de propriedades de contexto para todas as invocações, assim qualquer aspecto aplicado em uma invocação tem a garantia que estas informações estarão presentes:

- Identificador do componentes fonte.
- Identificador do componente alvo.
- Identificador do *binding*.
- Valores reificados das operações invocadas e argumentos.

A noção de operações para o controle de fluxos, do inglês *control-flow operations* (*CFlow*), é uma característica importante de um middleware que a arquitetura de referência deve contemplar. O *CFlow* é um dos mecanismos da orientação a aspecto que permite a criação de expressões de ponto de corte dinâmicas. Ele permite capturar dinamicamente a pilha de execução de uma função, controlar a atuação do aspecto de acordo com o contexto atual da pilha e criar relações entre pontos de junção baseadas no controle de fluxo da aplicação. Entende-se por uma expressão de pontos de corte dinâmica, toda expressão que precisa ser avaliadas em tempo de execução. No caso do *CFlow*, o estado atual da pilha de chamadas das funções precisa ser analisada a cada invocação.

Porém, em um ambiente baseado em componentes, existe um problema em implementar tais operações. O problema advém do fato de que os componentes são caixas-pretas, logo não existe como determinar se as invocações feitas por um dado componente a outros componentes são resultados de invocações feitas a operações do mesmo. Em resumo, não existe uma maneira de manter um *tracing* do caminho de execução de um componente encapsulado apenas com base em análise externas ao componente.

Contudo, a arquitetura disponibiliza um mecanismo de *CFlow* mínimo, onde a análise da pilha de execução limita-se a um nível de profundidade apenas. Já que apenas as interfaces dos componentes são acessíveis. Assim, a semântica do *CFlow* fica restrita à análise do elemento chamador e do elemento chamado. É evidente que esse *CFlow* é muito restrito, mas pelo menos evita a ambigüidade gerada pelos níveis mais profundos ao mesmo tempo em que evita a quebra do encapsulamento do componente.

Evidentemente, em uma comparação com a implementação tradicional do *CFlow*, a solução apresentada é bastante pobre e insuficiente para simular a maioria das aplicações de tal mecanismo. Sem dúvida, esse é um dos pontos fracos da arquitetura de referência. Pois, nas aplicações nas quais a execução dos aspectos é condicionada à execução de certas operações, o mecanismo fornecido pela arquitetura de referência é insuficiente. Isso obriga ao programador a buscar alternativas e soluções *ad-hoc* para solucionar o problema. Portanto, a freqüência com que ocorre esse cenário e o impacto das soluções específicas na arquitetura dos sistemas de middleware que seguem esse modelo determinarão se a arquitetura de referência necessitará de modificações a fim de contornar essa limitação.

Porém, atualmente não é possível determinar o real impacto desse problema, pois não existem na literatura implementações da arquitetura de referência que permitam essa análise e nem estudos sobre a importância do *CFlow* nos sistemas de middleware. Logo, trata-se de uma questão em aberto.

Essa limitação é inerente a filosofia do desenvolvimento baseado em componentes. Uma possível solução para esse problema é a abertura disciplinada da implementação interna dos componentes. Dessa maneira, cria-se uma solução de compromisso, na qual a filosofia de caixa-preta dos componentes é relaxada em troca de um mecanismo de *CFlow* mais preciso.

Por fim, dado que um aspecto é formado por, pelo menos, uma expressão de ponto de junção associado a, pelo menos, um *advice* que pode atuar antes, depois ou durante os pontos de junção definidos pela expressão em questão, a arquitetura de referência define os seguintes tipos de *advice* que denotam cada um desses momentos respectivamente: *before*, *after* e *around*.

3.2 Nível 2

O nível 2 consiste de um conjunto de interesses transversais descritos através das abstrações do nível 1. O objetivo principal desse nível é mapear os conceitos da arquitetura de referência para um número significante de interesses transversais a fim de descobrir problemas e eventualmente descobrir pontos ou questões não tratadas pela arquitetura.

Como a arquitetura de referência pretende ser a base para a construção de *toolkits* para a construção de sistemas de middleware, ela deve cobrir os principais aspectos envolvidos na construção desse tipo de sistema, sendo assim o conjunto de interesses deve ser escolhido para definir as características fundamentais de um middleware. [Loughran et al., 2005] propõe uma lista de interesses fundamentais em um middleware: distribuição, coordenação, mobilidade, persistência, segurança e transação. Este trabalho detém-se em dois interesses dessa lista: distribuição e coordenação, pois indiscutivelmente esses interesses são fundamentais em um middleware. Adicionalmente, esse trabalho trata o conceito de concorrência já que é um conceito entrelaçado e espalhado pelo código do OiL.

3.2.1 Distribuição

Em um ambiente distribuído, as funcionalidades de instalação, localização e comunicação remota são interesses fundamentais. A Figura 3.3 mostra o modelo de referência para a distribuição.

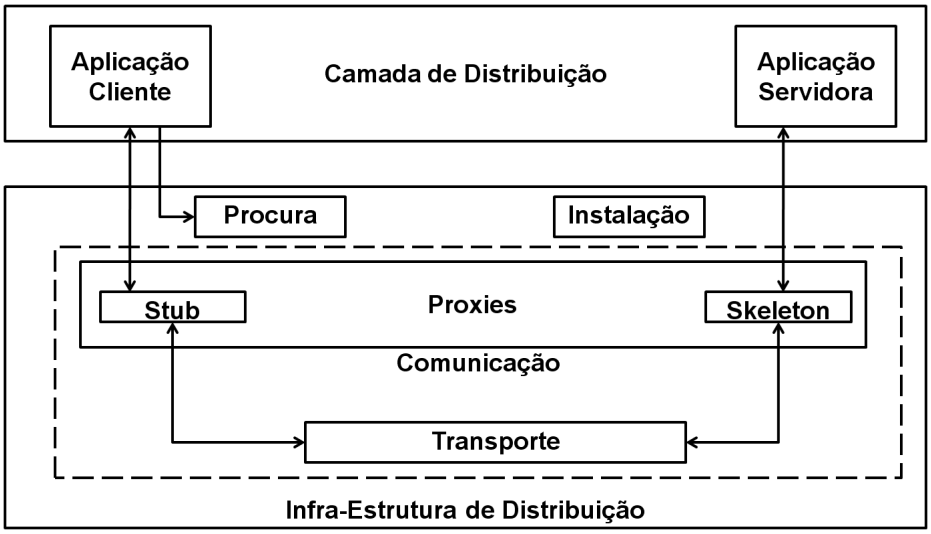


Figura 3.3: Modelo de referência para o interesse distribuição

A instalação é o processo de colocação de componentes na topologia de um middleware. Ela é fortemente relacionada a arquitetura da aplicação, desde que a arquitetura descreva quais os componentes existem e como eles se relacionam uns com os outros. Em ambientes baseados em componentes, geralmente, utiliza-se de linguagens de definição arquitetural para descrever especificações abstratas da arquitetura da aplicação, as quais são utilizadas para validar e coordenar

o processo de instalação de componentes. Essas linguagens de definição arquitetural são conhecidas na literatura como ADLs (*Architecture Definition Language*). O interesse de localização fornece facilidades para que os componentes possam localizar e acessar uns aos outros.

Os *bindings* estáticos são muito restritivos para um ambiente distribuído, assim a localização é feita sob demanda e em tempo de execução. Quando a procura termina, o componente recebe uma referência remota para o componente requisitado. O protocolo para acessar essa referência é dependente de plataforma de middleware. Esse interesse de distribuição pode interagir com outros interesses como segurança, transporte, mobilidade e escalabilidade, além de interagir com conceitos de alto nível como espaços de nomes, grupos, controle de acesso dentre outros.

Finalmente, a comunicação remota provê uma camada de distribuição que permite que um componente cliente comunique-se com um servidor remoto. Em geral, a infra-estrutura de distribuição utiliza *proxies* (*stubs* e *skeletons*) que podem ser acessados pelos componentes do cliente como objetos comuns. Assim os *stubs* e os *skeletons* abstraem o substrato do middleware, apresentando-o como uma facilidade de programação nativa para o usuário da aplicação [CORBA, 1999]. Um *stub* é uma função do cliente que realiza uma invocação remota via uma chamada de função local. Similarmente, um *Skeleton* é uma função que permite que uma invocação remota recebida pelo servidor seja despachada para o *servant* apropriado, tornando-a uma invocação local [Henning and Vinoski, 1999].

A instalação e a localização são funcionalidades associadas, pois ambas ocorrem em tempo de execução e são tipicamente parametrizadas por arquivos de configuração. Em geral, um componente centralizado conhecido por todos é utilizado para realizar a instalação e a localização de novos componentes. Do lado do cliente, um componente que precisa localizar outro pode fazer uso de uma fábrica de componentes. Essa fábrica torna transparente para o cliente a localização e a instalação dos componentes. Ela deve oferecer operações: *resolve* e *create*. A primeira permite a criação de componentes *proxy* para o atual componente servidor. A finalidade dessa função é permitir a criação de vários tipos de *proxies* que suportam diferentes tipos de protocolos de comunicação. A segunda permite ao cliente criar um novo componente e obter um instância do componente *proxy* para a interface requisitada. Inicialmente, o cliente tem uma associação local com a fábrica, ao invocar o método resolve da fábrica, o cliente será associado a um *proxy* para o componente servidor.

A Figura 3.4 mostra o estado de um cliente antes e após a invocação da operação *resolve*. Inicialmente o cliente possui apenas a associação com a fábrica de componentes, Figura 3.4(a). Quando o cliente invoca a função resolve da fábrica, os *bindings* apropriados são criados. No caso da Figura 3.4(b), uma associação entre o cliente e o *proxy* do servidor é criada.

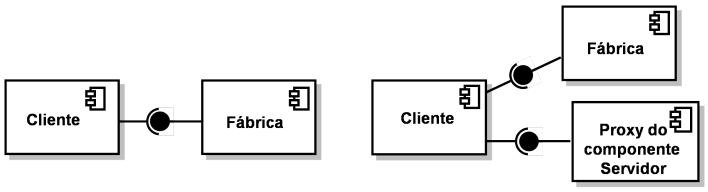


Figura 3.4: (a) Estado inicial do componente. (b) Estado final do componente

Para modularizar o interesse de localização, aspectos podem ser aplicados sobre a fábrica de componentes durante a instanciação, evitando a codificação de protocolos de resolução dentro dela. Uma definição de uma expressão de ponto de junção para tal aspecto pode ser a sugerida na equação 3.4.

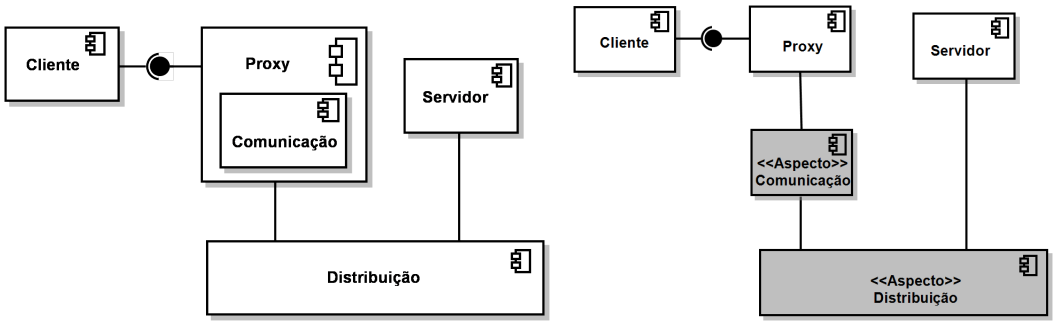
$$\begin{aligned} &\forall i \in I, \forall o_1, o_2, \in O, i.category \in \{i.required\} \\ &\wedge (o_1.signature = Factory.resolve(...) : Interface) \\ &\vee (o_2.signature = Factory.create(...) : Interface) \end{aligned} \tag{3.4}$$

A equação 3.4 descreve uma expressão de ponto de junção que intercepta os métodos *resolve* e *create* da fábrica. A expressão contida na equação 3.4 diz que: selecione todos os pontos de junção para operações cuja a assinatura case com *Factory.resolve(...): Interface* ou ($\vee$) com *Factory.create(...) : Interface* e ($\wedge$) que pertençam a uma interface do tipo *Required*.

Através da interceptação desses métodos é possível implementar, via aspectos, diversos protocolos e mecanismos que endereçam o problema da localização e do acesso de componentes distribuídos.

O interesse de comunicação também pode ser melhor modularizado através de aspectos. Os aspectos podem atuar durante as invocações realizadas sobre os *proxies* ou *stubs*, permitindo a implementação de diversos protocolos como *broadcasting*, *time-stamping* dentre outros. A Figura 3.5 compara as abordagens tradicional e a aspectual para modelar o conceito de comunicação. A arquitetura tradicional do conceito de distribuição é apresentada na Figura 3.5(a). Note que o *proxy* tem a responsabilidade de implementar os protocolo de comunicação, o que não ocorre na versão aspectizada exibida na 3.5(b).

Por fim, o mapeamento do interesse distribuição para o *microkernel* revelou que as operações *bind* e *unbind* são diretamente afetadas por tal interesse, uma vez que o estabelecimento de associações diretas entre componentes em espaços de endereçamento diferentes não é adequada.



(a) Arquitetura tradicional do conceito de distribuição (b) Aspectização do conceito de comunicação

Figura 3.5: Conceito de distribuição

Nos sistemas em que o interesse distribuição atua, as referências para os componentes fontes e alvo devem ser remotas, ou seja, as referências não são meras referências. Entretanto, a forma real dessas referências é dependente da plataforma de middleware. A arquitetura de referência assume que essas referências são URL's.

Dessa maneira, o *bind* e o *unbind* devem ter a competência de estabelecer e encerrar caminhos de comunicação entre os componentes. As outras operações do *microkernel* também podem sofrer alterações, pois elas podem necessitar de associações distribuídas entre o *microkernel* e os componentes que requisitam essas operações.

3.2.2 Coordenação

No desenvolvimento baseado em componentes, a coordenação pode ser definida como um padrão de interação que governa a comunicação e a invocação de operações entre dois ou mais componentes [Loughran et al., 2005]. Nesse contexto, é altamente desejável que os componentes não imponham restrições sobre outros componentes que compõem com eles um sistema. Ou seja, os componentes, idealmente, não devem decidir por eles mesmos como o processo de composição deve ser feito, como ocorre na programação orientada a objetos tradicional.

A modelagem da coordenação como um aspecto permite separá-la do código dos componentes. Conseqüentemente, diferentes protocolos de coordenação podem ser programados para controlar a coordenação de diferentes tipos de componentes.

Os protocolos de interação são encapsulados no aspecto de coordenação que descreve como e quando os componentes interagem. Dessa maneira, o chamando *glue code* é promovido a uma porção de software identificável e reutilizável [Arbab, 2005]. *Glue code* é a denominação

dada ao código responsável por realizar a conexão de componentes e coordenar a interação entre componentes. Em geral, esse tipo de código é feito utilizando linguagens de *script*.

A principal vantagem da implementação da coordenação como um aspecto é o alto grau de flexibilidade, uma vez que, sistemas diferentes podem ser arquitetados utilizando o mesmo conjunto de componentes, mas com diferentes regras de composição.

A separação do conceito coordenação inicia-se com a especificação da coordenação como parte da interface pública dos componentes. A partir desse momento, o desenvolvedor precisa conhecer não apenas a lista de operações das interfaces, mas também precisa conhecer o protocolo de comunicação.

O interesse de coordenação possui diferentes utilidades em uma aplicação distribuída baseada em componentes, podendo ser usado, por exemplo, como um *adapter* ou *binding*. Nos casos em que as interfaces *Provided* e *Required* não são compatíveis, mas suas semânticas são complementares, o interesse de coordenação pode atuar como um adaptador entre as interfaces. A arquitetura de referência faz uso do interesse de coordenação através do conceito de *Binding object*. Dessa maneira, a coordenação é responsável pelo processo de *binding* entre diferentes componentes de acordo com um protocolo específico. Nesse modelo, os componentes interagem através de sinais dando origem ao modelo *compound bind*, *bind composto*, como mostra a Figura 3.6 extraída de [Loughran et al., 2005].

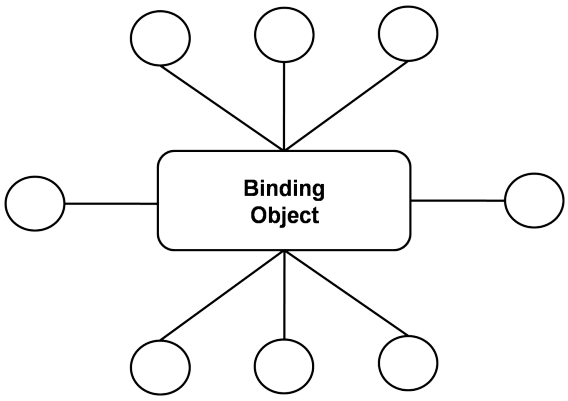


Figura 3.6: *Bind* composto

Um ambiente distribuído pode necessitar de vários modelos de coordenação para atingir seus objetivos. Cada modelo de coordenação tem suas peculiaridades, entretanto a separação desses modelos do código dos componentes evita o forte acoplamento da abordagem tradicional, na qual o modelo de coordenação é codificado dentro dos componentes. Os modelos *publish-and-subscribe*, *shared data space* e *channel* são exemplos de modelos de coordenação bastante

difundidos. Nos sistemas *publish-and-subscribe*, os publicadores, *publisher*, submetem informações no sistema usando repositório de informações, tópicos ou assuntos. Os interessados em receber essas informações, *subscribes*, devem subscrever-se nesse repositório. Portanto, um componente pode notificar um conjunto de componentes interessados com uma única operação. No modelo *shared data space*, todos os componentes escrevem e lêem dados em um espaço de endereçamento compartilhado por todos. No modelo *channel*, um canal é uma conexão ponto-a-ponto entre dois componentes, um fonte e um consumidor. O fonte escreve dados no canal, enquanto que o consumidor lê dados do canal. Nesse modelo a comunicação flui em um único sentido: do componente fonte para o componente consumidor. Os canais podem ser síncronos, assíncronos, FIFOS, LIFOS e etc.

O mapeamento do interesse de coordenação para o modelo de componentes da arquitetura de referência é simples. As interfaces *Provided* e *Required* são suficientes para realizar tal tarefa. As interfaces *Required* são as mesmas dos outros componentes do modelo, logo nenhuma alteração é necessária. Por sua vez, as interfaces *Provided* são descritas de acordo com o aspecto de coordenação, dependendo do tipo de uso do aspecto em questão e do modelo de coordenação empregado. No caso do aspecto ser utilizado como um adaptador ou um objeto *binding* para coordenar um conjunto de componentes conhecidos e com um conjunto de operações ou sinais conhecidos, a interface *Provided* deve ter a definição de um método para cada uma das operações ou sinais que o aspecto é capaz de interceptar.

No caso do aspecto ser utilizado para implementar um modelo de coordenação específico, a descrição das interfaces *Provided* mudam. No caso do modelo *publish-and-subscribe*, por exemplo, as interfaces *Provided* do aspecto de coordenação devem incluir as primitivas usuais utilizadas pelos componentes quando as interações ocorrem segundo esse modelo. Nesse caso específico, o aspecto iria conter uma operação para realizar a publicação, operação *publish*, e uma operação para realizar a inscrição no repositório de informações, operação *subscribe*. A Figura 3.7 mostra o modelo de componentes para o modelo de coordenação *publish-and-susbcribe*.

Figura 3.7: Modelo de componentes do aspecto de coordenação

O mapeamento para o *microkernel* demonstrou que as suas operações são afetadas por este aspecto. O aspecto de coordenação deve começar a atuar quando os componentes começam a se comunicar, e parar quando a aplicação terminar. Assim, as operações *START* e *STOP* são

afetadas por esse aspecto. Quando uma dessas operações é chamada, o aspecto deve tomar alguma atitude. O aspecto de coordenação faz parte da arquitetura da aplicação, assim quando a aplicação é carregada ou descarregada, ele também é carregado ou descarregado. Assim, as operações *LOAD* e *UNLOAD* também são afetadas por esse aspecto. Por sua vez, as operações *INSTANTIATE* e *DESTRUCT* são afetadas pelo aspecto de coordenação, pois esse aspecto é instanciado quando o aspecto de distribuição é carregado ou quando um componente envia um sinal, e é destruído quando o aspecto de distribuição é destruído. Por fim, a operação *BIND* deve gerenciar o processo de associação entre o aspecto de distribuição e o aspecto de coordenação, quando esse último estiver presente na aplicação.

O mapeamento do aspecto de coordenação para a APL também é bem simples. A definição utilizando a APL do aspecto de coordenação nos casos em que ele atua como um objeto *binding* ou um adaptador consiste em selecionar as interações entre dois componentes quaisquer, A e B. Por exemplo, a equação 3.5 exibe uma expressão que seleciona todas as invocações a operações localizadas nas interfaces *Required* do componente A, ponto de junção do tipo *call*, tendo como alvo o componente B. Similarmente, a Listagem 3.6 mostra uma expressão que seleciona todas as invocações a operações localizadas nas interfaces do tipo *Required* do componente B cujo alvo é o componente A.

$$\begin{gathered} \forall c \in C, \forall i \in c.interfaces, \forall o \in o.operations \\ (c.name = A \wedge i.category \in \{required\}) \wedge \\ (o.signature = \text{“}B. * (?parameters) :?result) \end{gathered} \tag{3.5}$$

$$\begin{gathered} \forall c \in C, \forall i \in c.interfaces, \forall o \in o.operations \\ (c.name = B) \wedge (i.category \in \{required\}) \wedge \\ (o.signature = \text{“}A. * (?parameters) :?result) \end{gathered} \tag{3.6}$$

Caso o aspecto empregue um modelo de coordenação, então ele deve capturar as chamadas às operações comuns de tal modelo. Por exemplo, se o aspecto implementa o modelo *publish-and-subscribe*, então essas operações devem ser interceptadas pela APL. A equação 3.7 exibe uma possível definição dessa expressão descrita utilizando a APL. Essa expressão seleciona todas as invocações aos métodos *pusblish* ou *subscribe* que ocorrem em interfaces de componentes do

tipo *Required*.

$$\forall c \in C, \forall i \in c.interfaces, \forall o \in o.operations(i.category \in \{required\}) \wedge \quad (3.7)$$
$$((o.signature = publish(?parametrs) \vee o.signature = subscribe(?prameters))$$

3.2.3 Concorrência

No ambiente distribuído, as aplicações são inerentemente concorrentes, pois a aplicação é dividida processos que executam em unidades de distribuição (processador ou computador) independentes. A execução seqüencial desses processos representaria um enorme desperdício do poder computacional em várias aplicações. Por esta razão, a programação concorrente é fundamental nos sistemas de *middleware*, uma vez que ela permite a execução simultânea de vários processos da aplicação.

```
function invoke(self, reference, operation, ...)
    --[[VERBOSE]] verbose:invoke(true, "invoke remote operation")
    local context = self.context
    local requester = context.requester
    local result, except = requester:getchannel(reference)
    if result then
        local channel = result
        local mutex = context.mutex
        --[[VERBOSE]] verbose:invoke(true, "get a mutex before a new request",channel)
        mutex:locksend(channel)
        result, except = requester:newrequest(channel, reference, operation, ...)
        --[[VERBOSE]] verbose:invoke(true, "drop a mutex after a new request",channel)
        mutex:freesend(channel)
        if result then
            result[InvokerKey] = self
            result[ChannelKey] = channel
            result = Request(result)
        end
    end
    --[[VERBOSE]] verbose:invoke(false)
    return result, except
end
```

Figura 3.8: Entrelaçamento dos conceitos de distribuição e concorrência no componente *Invoker*

Todavia, a programação concorrente utilizando as linguagens de programação tradicionais é uma tarefa complexa que envolve vários conceitos como a definição de áreas com acesso sincronizado, identificação de tarefas que possam executar simultaneamente, definição da politica de escalonamento dentre outras questões. Porém, esses conceitos não são adequadamente modularizados pelos mecanismos de decomposição dos paradigmas tradicionais. Dessa maneira, a

concorrência é típico conceito que sofre dos efeitos negativos do espalhamento e entrelaçamento de seu código no resto do sistema [Kiczales et al., 1997].

As figuras 3.8 e 3.9 exibem o entrelaçamento de código relativo a concorrência dentro de um componente distando a distribuição, o componente *Invoker*, apresentado na seção 2.2.2. As linhas em destaque das referidas figuras marcam os trechos de código relacionados a implementação da concorrência dentro do modulo *Invoker*. A Figura 3.8 apresenta a função *invoke* utilizada para realizar uma invocação remota. Ela mostra seis pontos de transição entre os conceitos de concorrência e distribuição. A Figura 3.9 exibe o código relativo à função *receivefrom* que é utilizada para receber as respostas das requisições. Tal figura possui 8 pontos de transição entre conceitos.

```
function receivefrom(self, channel, request, probe)
    if request.success == nil then
        local requester = self.context.requester
        local mutex = self.context.mutex
        if not mutex or mutex:lockreceive(channel, request) then
            local result, except, failed
            repeat
                result, except, failed = requester:getreply(channel, probe)
                if result then
                    mutex:notifyreceived(channel, result)
                else
                    for requestid, request in pairs(failed) do
                        if type(requestid) == "number" then
                            request.success = false
                            request.resultcount = 1
                            request[1] = except
                            mutex:notifyreceived(channel, request)
                        end
                    end
                    break
                end
            until result == request or (probe and result == true)
            mutex:freereceive(channel)
        end
    end
    local handler = self[request.success]
    if handler then
        return handler(self, channel, request, probe)
    end
end
```

Figura 3.9: Entrelaçamento dos conceitos de distribuição e concorrência no componente *Invoker*

Como é possível notar nas figuras 3.8 e 3.9, o OiL utiliza um modelo de sincronização baseado em *mutexes*. Os *mutexes* são objetos de sincronização que provêem exclusão mutua a certas regiões do fluxo de execução do programa. Eles delimitam as áreas em que o acesso é sincronizado. A modelagem do conceito de concorrência como aspecto é bastante direita: o aspecto deve

interceptar as áreas que precisam ser protegidas pelos mecanismos de sincronização e injetar o código necessário.

Capítulo 4

AO-OiL

Esse capítulo apresenta o Nível 3 da arquitetura de referência. Esse nível consiste do mapeamento dos elementos da arquitetura de referência para uma plataforma de middleware específica, no caso o AO-OiL. O objetivo desse mapeamento é validar os conceitos e visões da arquitetura de referência [Greenwood et al., 2007], bem como prover um middleware OA simples, leve e flexível.

A seção 4.1 descreve o mapeamento do modelo de componentes da arquitetura de referência para o modelo de componentes do OiL. A seção 4.2 mostra como as operações do *microkernel* são suportadas pelo OiL. A seção 4.3 apresenta o mapeamento do modelo de pontos de junção da arquitetura de referência para o AO-OiL. Seção 4.4 especifica um mapeamento entre a APL e RE-AspectLua. Finalmente, a seção 4.5 apresenta o mapeamento dos conceitos transversais para o AO-OiL.

O processo de criação do AO-OiL é dividido em duas fases: (i) refatoração dos elementos existentes no OiL com base na arquitetura de referência, de forma a definir quais serão os elementos base e quais serão os aspectos e (ii) implementação dos elementos mapeados. Portanto, o mapeamento para o nível 3 dos elementos do OiL é a primeira etapa da construção do AO-OiL, pois permite definir como o modelo de componentes e o *microkernel* da arquitetura de referência são mapeados para o OiL, e como o modelo de pontos de junção e a APL são mapeados para o RE-AspectLua.

A Figura 4.1 apresenta uma visão geral do AO-OiL dividida em 3 camadas: (1) *microkernel*: desenvolvido na linguagem Lua, (2) os aspectos: desenvolvidos em RE-AspectLua independentemente do código base da aplicação; (3) a ligação entre os aspectos e o middleware: feita através dos *scripts* de conexão em RE-AspectLua, camada intermediária da figura. É importante ressaltar que o emprego de RE-Aspectlua como linguagem de conexão trás uma grande vantagem ao

modelo: podem haver vários aspectos de distribuição e/ou coordenação atuando em uma mesma aplicação. Nesse caso, a camada intermediária seria responsável por empregar um ou outro de acordo com contexto.

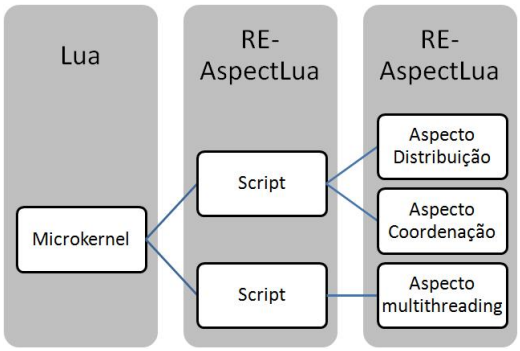


Figura 4.1: Visão geral do AO-OiL

4.1 Mapeamento do Modelo de Componentes

A modelagem da arquitetura de referência é baseada em componentes. Assim, todos os elementos são descritos através de componentes e esses são descritos através de interfaces providas (*Provided*) e requeridas (*Required*). O OiL também segue o modelo de programação baseado em componentes. No OiL, todas as composições e interações são construídas através das portas dos componentes. Tais portas são implementações diretas do conceito6y7 de interfaces definido na arquitetura de referência. Logo, as facetas concretizam as interfaces do tipo *Provided*, enquanto que os receptáculos concretizam as interfaces *Required*. Dessa maneira, pode-se concluir que existe um mapeamento 1 para 1 dos componentes da arquitetura de referência para os componentes do OiL.

Entretanto, o OiL não contempla o conceito *binding* como uma construção de primeira ordem, como acontece na arquitetura de referência. Na arquitetura de referência, há definição de: (i) tipos de associações, no caso *binding* e *AO-binding*, (ii) tipos para as interfaces, *Provided* e *Required*, e é possível definir regras para a ligação entre tais interfaces. Tais regras evitam que os programadores construam configurações inválidas da arquitetura. No caso do OiL, o processo de *binding* é de responsabilidade do desenvolvedor e se resume a uma atribuição de uma faceta a um receptáculo. Porém, no AO-OiL, o conceito de *Binding* será modelado como um conceito de primeira ordem, e portanto, serão definidas estruturas que representam os conceitos de *binding*

e *AO-binding* correspondentes àquelas definidas na arquitetura de referência. A Listagem 2.10, no capítulo 2, mostra como o processo de *binding* é simples no OiL. A Tabela 4.1 resume o mapeamento do modelo de componentes da arquitetura de referência para o OiL.

Elemento da Arquitetura de Referência	**Elemento do OiL**
Componente	Componente
Interface *Provided*	Porta do tipo Faceta
Interface *Required*	Porta do tipo Receptáculo
Binding	Não existe correspondente de primeira ordem
AO-Binding	Não existe correspondente de primeira ordem. Porém, no AO-OiL, o *AO-Binding* é definido como um conceito de primeira ordem

Tabela 4.1: Mapeamento dos elementos da arquitetura de referência para o OiL

A modelagem do *AO-binding* como um conceito de primeira ordem permite estabelecer diferentes composições entre elementos do modelo, esclarecer as funcionalidades suportadas por um aspecto no modelo, como por exemplo, a criação de cadeias de *advice*, reificar chamadas de operações entre chamadas tipadas e genéricas, e solucionar o seguinte problema: uma instância de um componente pode algumas vezes servir para um programa inteiro, enquanto que em outros momentos, uma instância aspectual do componente pode ser necessária para cada componente executando em algum dos pontos de junção especificados por uma expressão de ponte de junção.

4.2 Mapeamento do OiL para o Microkernel

O OiL apresenta um modelo arquitetural orientado a objetos que oferece serviços aos seus usuários através de API's. Os serviços básicos oferecidos são: instanciação de componentes, registros de componentes e serviços de comunicação. Por outro lado, como dito anteriormente, a arquitetura de referência define um *microkernel* que contém as operações básicas para a construção de serviços de *middleware* sobre ele. Assim, a arquitetura do AO-OiL baseada na arquitetura de referência, consiste de um pequeno núcleo funcional, *microkernel*, e de um conjunto de extensões. As extensões são utilizadas para estender as funcionalidades do middleware. Elas são implementadas como aspectos que interceptam os métodos da API do *microkernel* para adicionar funcionalidade. A API do *microkernel* pode ser mapeada para o OiL da seguinte forma:

- LOAD/UNLOAD : Serviço de registro de componentes, implementado pelo componente *Server Broker*

- INSTANCIATE/DESTRUCT: Serviço de criação de componentes. Esse serviço, na verdade, é disponibilizado pelo LOOP.

- BIND/UNBIND: O processo de *binding* é feito através de atribuições. Logo, esse serviço não é suportado diretamente pelo OiL. Mas será definido no AO-OiL.

O serviço de comunicação do OiL não tem um mapeamento direto para o *microkernel*. Entretanto, ele pode ser mapeado como uma extensão a nível de middleware na arquitetura de referência. Assim, uma implementação do OiL baseada no padrão de projeto *microkernel* seria composta apenas pelos componentes *Server Broker* e *Request Dispatcher*. Os outros componentes, apresentados na Figura 2.6 e na Figura 2.7, estão envolvidos no suporte a infra-estrutura de distribuição, no caso específico do OiL, o suporte ao protocolo CORBA. Esses componentes são adicionados ao *microkernel* para criar a camada de distribuição. A Figura 4.2 exibe a arquitetura do AO-OiL gerada após o mapeamento.

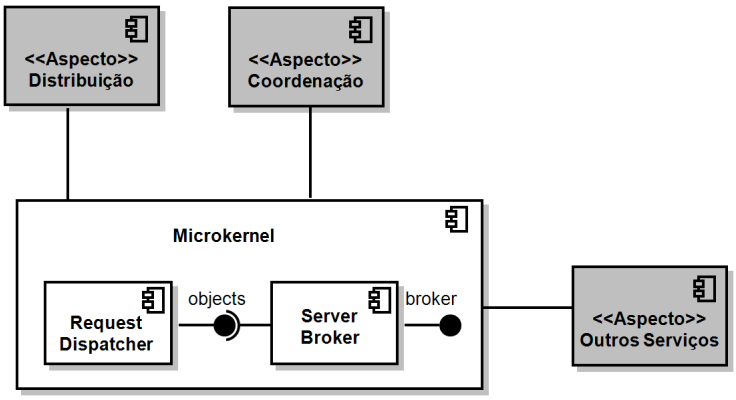


Figura 4.2: A arquitetura do AO-OiL

Além disso, o *Server Broker* oferece as operações *retrieve*, *pending*, *step* e *tostring*. Essa última é utilizada para obter uma IOR a partir de um objeto. As funções *pending* e *step* são utilizadas para verificar se existem requisições pendentes e para executar apenas a próxima requisição, respectivamente. Finalmente, a operações *retrieve* permite recuperar um objeto *servant* a partir de seu identificador. Essas operações não estão presentes na arquitetura de referência. No AO-OiL, essas funcionalidades serão implementadas como componentes de serviços do *microkernel*, pois tais funcionalidades são simples e essenciais. A Tabela 4.2 resume como os serviços do OiL serão mapeados para o AO-OiL

Funcionalidade do OiL	Microkernel	Aspecto
Serviço de registro de componentes	X	
Serviço de criação de componentes	X	
Processo de *binding*	X	
Serviço de distribuição		X
Serviço de coordenação		X

Tabela 4.2: Resumo do mapeamento das funcionalidades do OiL

4.3 Mapeamento do Modelo de Pontos de Junção

A arquitetura de referência foi pensada para atuar nas interfaces dos componentes de maneira não intrusiva, o que levou a definição de dois tipos de pontos de junção: *call* e *execution*. Porém, RE-AspectLua foi projetada para atuar em objetos (tabelas), funções e variáveis. Isso justifica o fato dela possuir um conjunto de pontos de junção mais rico do que o definido pela arquitetura de referência. Esse conjunto é formado pelos seguintes pontos de junção: *call*, para interceptar chamadas de funções, *callone*, para interceptar uma única vez uma chamada de função, *introduction*, para introduzir funções em objetos e, por fim, os pontos de junção *get/set* que atuam respectivamente na interceptação de escrita e leitura de variáveis. Sendo assim, o modelo de pontos de junção do RE-AspectLua é intrusivo, portanto os dois modelos diferem nesse sentido.

Apesar das claras diferenças de projetos entre as duas linguagens, um mapeamento entre elas pode ser realizado. Os pontos de junção *call* e *execution* da arquitetura de referência são chamadas de funções cuja a única diferença está no local aonde ocorre a invocação. Portanto, esses dois tipos de pontos de junção podem ser mapeados para os pontos de junção *call* e *callone* do RE-AspectLua, pois os mesmos interceptam invocações de funções sobre qualquer objeto da linguagem Lua.

Ambas as abordagens contém os seguintes tipos de *advices* em seus modelos de pontos de junção: *before*, *after* e *around*. Dessa maneira, conclui-se que entre os dois modelos de pontos existe um mapeamento 1 para 1.

4.4 Mapeamento da APL para RE-AspectLua

RE-AspectLua é uma linguagem concreta que descreve aspectos abstratos e suas composições, e implementa a dicotomia conceitual aspecto-base. Por sua vez, a APL é uma linguagem abstrata cujo objetivo é ser um modelo de referência para as linguagens de definição de pontos de junção. A APL descreve apenas o mecanismo de composição que define a interação en-

tre um componente que representa um interesse transversal com outro(s) componente(s) base, pois a arquitetura de referência não contempla explicitamente a dicotomia aspecto-base. Entretanto, em uma ligação aspectual, via o *AO-Binding*, um componente faz o papel de aspecto, podendo-se então afirmar que a arquitetura de referência contempla implicitamente a dicotomia aspecto-base. Apesar dessas pequenas diferenças, as duas linguagens possuem um objetivo em comum que é realizar a composição de elementos aspectuais e elementos base na aplicação em um sistema único. Logo, um mapeamento das construções utilizadas para realizar a composição dos elementos determina se as linguagens são de fatos equivalentes.

A APL define uma expressão de ponto de junção como uma expressão formada por quatro sub-expressões, uma para cada elemento arquitetural, apresentados na seção 3.1.2, descrita através de lógica de primeira ordem. Cada sub-expressão da APL pode ter quantificadores e variáveis quantificadas, lista de propriedades, seletores de pontos de junção, operadores lógicos e operadores da teoria dos conjuntos $\in$ e $\notin$ que atuam sobre contêineres, componentes, interfaces e operações.

Em contra-partida, no RE-AspectLua cada expressão de ponto de junção é formada por *wildcards*, padrões de tipos e padrões de assinatura que atuam sobre objetos e funções. O RE-AspectLua não define nenhum tipo de operador ou propriedade de contexto. Assim, uma expressão de ponto de junção no RE-AspectLua é uma simples, porém expressiva enumeração de padrões de casamento.

Os quantificadores da APL podem ser mapeados diretamente para os *wildcards* do RE-AspectLua; por exemplo o operador $\forall$ (para todo) pode ser mapeado para o *wildcard* *. As variáveis quantificadas podem ser expressas no RE-AspectLua através de padrões de casamentos de tipos e de assinatura especificados no campo *list*, de modo que um tipo representaria um conjunto de componentes e o padrões de assinatura representariam operações. Por exemplo, a expressão da APL apresentada na expressão 4.1 seleciona todos os componentes cujo o nome é *ComponentA*. Ela pode ser rescrita em RE-AspectLua na forma exibida na equação 4.2

$$\forall\, cp \in C,\, c.name = \text{“}ComponentA\text{“} \tag{4.1}$$

$$list = ComponentA.* \tag{4.2}$$

Os seletores de ponto de junção da APL são mapeados diretamente para o RE-AspectLua. Eles são mapeados para o campo *designator* do RE-AspectLua, onde tal campo define o tipo de ponto de junção. Como foi dito anteriormente, a arquitetura de referência define dois tipos de seletores: *call* e *execution*. O *call* captura todos os pontos de junções do componente que realiza uma invocação a uma dada função, i.e requisita um serviço, enquanto que o *execution* captura todos os pontos de junções relacionados a um componente que executa uma dada função, ou seja, que provê um serviço. Ambos são mapeados para o *designator call* de RE-AspectLua como mostra a seção 4.3.

Por sua vez, os operadores não podem ser mapeados para RE-AspectLua, pois essa última não contempla o conceito de operadores. RE-AspectLua baseia-se apenas em expressões que codificam enumerações de pontos de junção. A única exceção é o operador lógico *OR* que pode ser simulado passando mais de uma expressão para o aspecto que está sendo definido. As propriedades de contexto e os mecanismos de passagem de informações contextuais também não são contempladas no RE-AspectLua. Logicamente, um mapeamento dessas construções não é possível.

Para resolver essas carências, esse trabalho estende RE-AspectLua adicionando as seguintes propriedades: (i) um conjunto de operadores booleanos, (ii) um conjunto de propriedades de contexto associadas a todas as invocações e (iii) um mecanismo de passagem de informações contextuais.

O conjunto de operadores lógicos é formado pelos seguintes operadores: && (operador *AND* lógico), || (operador *OR* lógico) e ! (operador *NOT* lógico). Nessa versão estendida de RE-AspectLua, uma expressão de ponto de junção é formada por uma expressão, descrita dentro do campo *expression*, que indica a lista de pontos atingidos pelo RE-AspectLua, e um tipo, campo denominado *type*, que indica o momento em que o aspecto irá atuar (*before*, *after*, *type*). As expressões são descritas pelo seguinte padrão: "designator (identificador) OPERADOR designator (identificador)". Aonde os *designators* indicam o tipo de ponto de junção, o conjunto de *designators* de RE-AspectLua foi mantido, e o identificador é uma *string* que referencia ou um elemento existente no código base ou um conjunto de elementos através do *wildcard* '*'. Por exemplo, a expressão 4.3 seleciona todos os métodos do componente *ComponentA* exceto o método *getBalance*.

$$expression = \{call(ComponentA.get*) \;\&\&\; !call(ComponentA.getBalance)\} \qquad (4.3)$$

As propriedades de contexto são valores de pares <chave; valor> associados às invocações

de funções. A classe *Context*, definida no escopo deste trabalho, oferece métodos para recuperar e para adicionar propriedades de contexto. Para criar uma propriedade, *Context* disponibiliza o método *setProperty(key,value)*, que recebe como parâmetros um identificador, *key*, e um objeto representando a propriedade em si, *value*. Para obter o valor de uma propriedade, *Context* oferece o método *getProperty(key)*, que recebe a chave da propriedade. Além disso, essa classe oferece métodos para recuperar propriedades contextuais a partir de um arquivo, *loadFile(file)*, e para gravar propriedades contextuais em um arquivo, *saveFile(file)*. A Tabela 4.3 resume a API da classe *Context*.

Método	**Descrição**
getProperty(key)	Recupera a propriedade *key*
setProperty(key,value)	Atribui o valor *value* a propriedade indicada por *key*
loadFile(file)	Recupera propriedades de contexto a partir de um arquivo
saveFile(file)	Grava as propriedades contextuais em um arquivo

Tabela 4.3: API da classe *Context*

A APL é baseada em componentes, e não faz qualquer suposição sobre como esses componentes são implementados (classes, funções, tabelas etc.) ou da linguagem escolhida para tal, portanto, ela permite expressar pontos de junção em nível de componente. Entretanto, em algumas situações é necessário capturar elementos em nível de linguagem. Para isso, as propriedades contextuais fornecem um meio elegante de expor propriedades que estão além dos limites dos componentes, aumentando o poder de expressividade da linguagem.

A passagem de informações contextuais é feita através das propriedades contextuais. Para isso, a classe *JoinPoint* foi criada a fim de representar o contexto de um ponto de junção. A Tabela 4.4 apresenta a API de tal classe. Ela é constituída pelas propriedades contextuais associadas a invocações no ponto de junção que a classe representa, pelas propriedades estáticas e pelas propriedades dinâmicas associadas ao ponto de junção.

Método/Campo	**Descrição**
getArgs()	Método que retorna os argumentos do ponto de junção
getSignature()	Método que retorna a assinatura do ponto de junção
getTarget()	Método que retorna o objeto alvo
Context	Campo que contém as propriedades contextuais

Tabela 4.4: API da classe *JoinPoint*

O código da Listagem 4.1 exemplifica a criação de uma propriedade de contexto em RE-AspectLua. A linha 1 recupera o contexto do ponto de junção *ao_oil.kernel.Client.proxy* através

da função *search_instance*, definida em RE-AspectLua, e armazena-o na variável *context*. Na linha 2, a propriedade *distribuition* é associada ao componente *proxy*. A criação da propriedade é feita através da função *setProperty* da classe *Context*. Essa função recebe o nome da variável a ser criada e o valor. Listagem 4.1 cria a propriedade "*distribuition*" com o valor "*corba*". A definição dessa propriedade permite que os aspectos de distribuição decidam sobre quais proxies irão atuar. Por exemplo, caso a aplicação use infra-estrutura de comunicação diferentes, mas que atuem interceptando as invocações aos *proxies*, ou seja, ambos os protocolos atuam sobre o mesmo conjunto de pontos de junção, essa propriedade pode ser associada aos *proxies* para selecionar o tipo protocolo de distribuição aplicar sobre aquele *proxy*.

```
1 local context = LuaMOP:search_instance("ao_oil.kernel.Client.proxy").jp.
    context
2 local distribution = context:setProperty("distribution,  corba ")
```

Listagem 4.1: Criando uma propriedade de contexto para um ponto de junção

Para recuperar uma dada propriedade de contexto de um ponto de junção, primeiramente deve-se recuperar o ponto de junção desejado, através da classe *Context*,e em seguida acessar a propriedade desejada através das propriedades de invocações do ponto de junção. A primeira linha da Listagem 4.2 recupera o ponto de junção *ao_oil.kernel.Client.proxy* e armazena-o na variável *context*. Em seguida, a linha 2 acessa a propriedade de contexto determinada pela chave *distribution* através da função *getProperty* da classe *Context*.

```
1 local context = LuaMOP:search_instance("ao_oil.kernel.Client.proxy").jp.
    context
2 local distribution = context:getProperty("distribution")
```

Listagem 4.2: Recuperando uma propriedade de contexto de um ponto de junção

Por outro lado, o RE-AspectLua suporta a adição de novas características nas classes, conceito denominado de *introduction*. Enquanto que a arquitetura de referência não considera esse conceito importante e não define nenhuma construção que contemple essa característica. Entretanto, essa posição é bastante discutível, pois grande parte dos *frameworks* orientados a aspectos oferece suporte ou mecanismos que suprem a funcionalidade oferecida pelas *introductions*. [Loughran et al., 2005] oferece uma vasta lista de *frameworks* que suportam a orientação a aspectos, nessa lista é possível notar a forte presença das *introductions*.

Finalmente, a precedência entre aspectos pode ser mapeada da arquitetura de referência para o RE-AspectLua. A arquitetura de referência diz que a precedência dos aspectos pode ser alterada em tempo de execução, através de algum mecanismo reflexivo. RE-AspectLua oferece um

mecanismo de precedência que permite alterar a ordem dos aspectos. Tal mecanismo é implementado como uma API que permiti manipular a ordem dos aspectos. Conclui-se que existe um mapeamento direto entre ambas as abordagens. A Tabela 4.5 exibe um resumo das características da APL que podem ser mapeadas para o RE-AspectLua.

O mapeamento entre a APL para RE-AspectLua revelou algumas carências de RE-AspectLua como a falta de operadores e propriedades de contexto. Resolvidas essas questões, pode-se afirmar que RE-AspectLua passou a ter o poder de expressão para representar as expressões de ponto de corte exigidas pelos conceitos transversais presentes nos sistemas de middleware.

4.5 Mapeamento dos conceitos transversais para o AO-OiL

Esta a seção apresenta o resultado do mapeamento dos conceitos transversais de distribuição e coordenação para a arquitetura do AO-OiL. O objetivo dessa seção é identificar os componentes aspectuais envolvidos na construção do middleware.

APL	**RE-AspectLua**
Quantificadores	*Wildcards*
Variáveis quantificadas	Padrões de casamentos de tipos e de assinaturas
Lista de propriedades	Mapeamento direto
Seletores de pontos de junção	*Designator*
Operadores	Mapeamento direto
As propriedades de contexto	Mapeamento direto
Mecanismos de passagem de informações	Propriedades associadas às invocações de operações
Precedência baseada em reflexão	Precedência baseada em API

Tabela 4.5: Resumo do mapeamento do APL para Re-AspectLua

4.5.1 Distribuição

Para a arquitetura de referência, o conceito de distribuição pode ser classificado em três categorias [Greenwood et al., 2008]: (i) o serviço de distribuição não possui nenhuma relação com a orientação a aspecto; (ii) o serviço de distribuição é estruturado como um serviço aspectual do middleware, (iii) o serviço de distribuição é estruturado sobre um modelo de componentes aspectuais distribuídos que oferece suporte para composição aspectual no modelo de distribuição do middleware.

A primeira alternativa é descartada rapidamente para os propósitos desse trabalho, pois o serviço de distribuição é um interesse bastante complexo para ser codificado dentro do *microkernel*. Sendo assim, essa alternativa contempla os sistemas de middleware tradicionais.

A estruturação da distribuição como um serviço aspectual é alcançada através da interceptação de pontos da aplicação cliente para inserir o aspecto relacionado à distribuição. A Figura 4.3 ilustra essa abordagem.

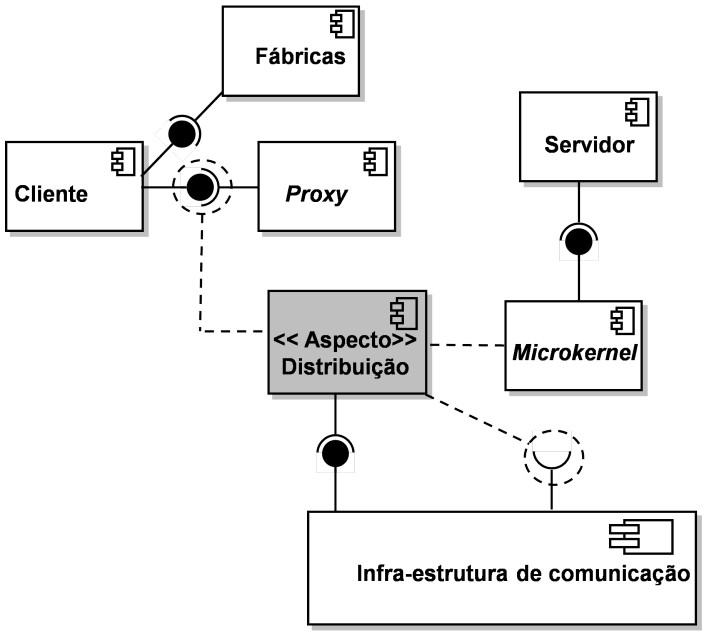


Figura 4.3: Distribuição com interceptadores

O cliente comunica-se localmente com um conjunto de *proxies* para as aplicações servidoras. Então, os aspectos interceptam as invocações de métodos feitas aos *proxies* para aplicarem *advices* sobre tais invocações, transformando-as em invocações remotas. Do lado do servidor, os aspectos não interceptam invocações, eles realizam a desserialização das chamadas remotas e as entrega para o *microkernel*, que notifica a aplicação servidora.

Por sua vez, a estruturação da distribuição através de composições aspectuais no modelo de distribuição é feita através da exposição de pontos de junção destinados à aspectos remotos. Esses pontos de junção seriam ”*hooks*“ ou ”*design rules*“ [Dósea et al., 2007], interfaces que governam como o código base expõe pontos de junção e como os aspectos os utilizam, que o modelo de distribuição disponibiliza para que aspectos relativos ao interesse de distribuição atuem. A Figura 4.4 apresenta essa abordagem. Nela é possível ver um ponto de corte remoto sendo capturado pelo aspecto de distribuição, ele decide se o *advice* remoto ou o *advice* local

será chamado.

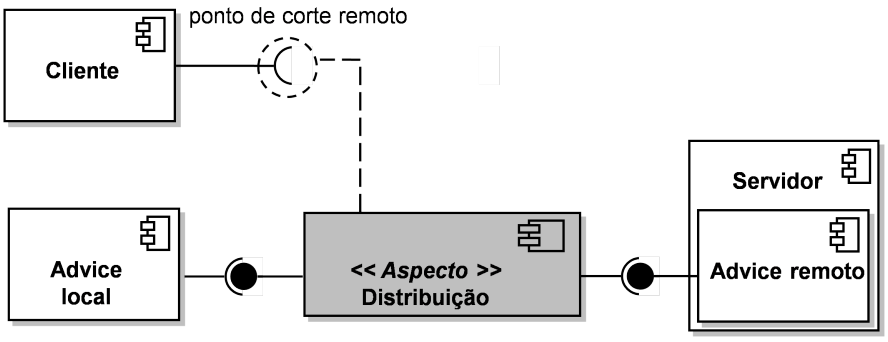


Figura 4.4: Distribuição com pontos de junção remotos explícitos

A primeira abordagem foi escolhida em detrimento da segunda, pois a primeira deixa o aspecto de distribuição mais transparente e melhor adequar-se ao modelo de distribuição do OiL, que é baseado em *proxies*. Assim, essa abordagem permite uma maior reutilização dos componentes do OiL relativos à distribuição, pois eles serão anexados na arquitetura via aspectos quando houver necessidade.

Diferentes tipos de aspectos de distribuição podem estar presentes em uma aplicação, atuando sobre diferentes tipos de *proxies*. Essa situação pode gerar ambigüidades na atuação dos aspectos, pois os pontos de junção são os mesmos, independentemente do aspecto de distribuição usado, resultando na interceptação das invocações por todos os aspectos. A seção 5.4.1 apresenta um cenário onde ocorre a referida situação e como utilizar RE-AspectLua para direcionar a interceptação para diferentes locais.

4.5.2 Coordenação

Os sistemas distribuídos exigem modelos de comunicação flexíveis, dinâmicos, escaláveis e desacoplados da natureza das aplicações. Dessa maneira, o modelo de coordenação fornecido pelo middleware para as aplicações deve atender tais requisitos. Sendo assim, o modelo de coordenação adotado nessa pesquisa foi o *publish and subscribe*. A escolha desse modelo foi fundamentada não apenas nos requisitos exigidos pelos novos sistemas distribuídos, mas também pelo fato de que o OiL é uma arquitetura do tipo *broker*, e esse tipo de arquitetura é adequada para sistemas *Publish/Subscribe*.

Como o próprio nome indica, o modelo de interação *Publish/Subscribe* é baseado em dois elementos principais: o *Publisher* e o *Subscriber*. O primeiro elemento é responsável por publi-

car eventos, enquanto que o segundo tem a capacidade de registrar seu interesse em um conjunto de eventos determinado. De maneira resumida, os *publishers* produzem eventos e os *subscribers* consomem eventos.

O modelo tradicional do *Publish and Subscribe* é baseado em um serviço de eventos que gerencia as inscrições nos eventos do sistema e eficientemente entrega as notificações a quem estiver interessado em recebê-las.

O serviço de eventos provê um desacoplamento entre *publishers* e *subscribers* em três dimensões: espaço, tempo e sincronização [Eugster et al., 2003]. O desacoplamento espacial significa que os participantes não precisam se conhecer, ou seja, os produtores de eventos não precisam ter referências para os consumidores e não precisam saber quantos consumidores estão participando da interação. Os consumidores não precisam ter referências para os produtores de eventos que lhes interessam. O desacoplamento temporal indica que os participantes da interação não precisam estar ativos ao mesmo tempo para que a comunicação ocorra. Por fim, o desacoplamento de sincronização indica que a comunicação não é bloqueante, isto é, os produtores não são bloqueados durante a produção dos eventos e os consumidores podem ser notificados de maneira assíncrona, enquanto realizam outras atividades concorrentemente. O desacoplamento da produção e do consumo de informação aumenta a escalabilidade do modelo.

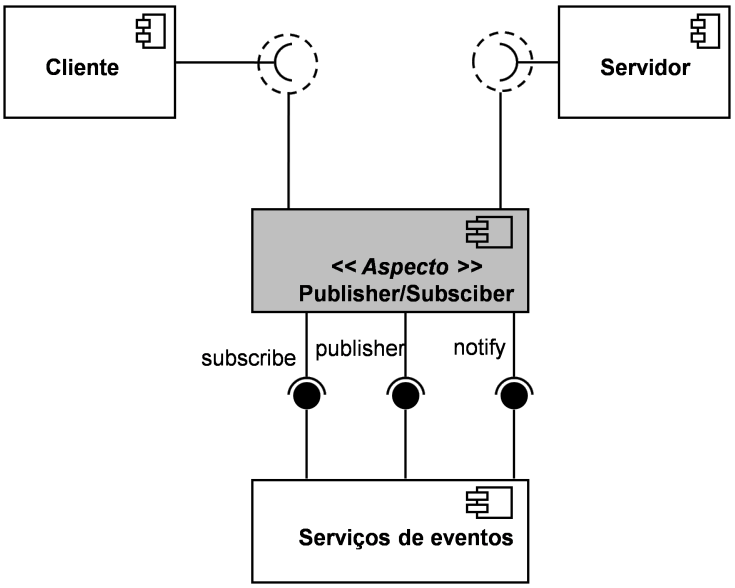


Figura 4.5: Arquitetura do aspecto coordenação

A modelagem da coordenação como aspecto é bastante simples, o aspecto deve instanciar os componentes que oferecem suporte às primitivas do modelo utilizadas pelos componentes

durante a interação. A Figura 4.5 ilustra a modelagem da coordenação como um aspecto.

O aspecto *publish/subscribe* instancia e configura o serviço de eventos de acordo com a aplicação. Quando algum componente da aplicação invoca as primitivas do modelo, tal como *subscribe()* e *publish()*, o aspecto intercepta essas chamadas e aplica *advices* sobre elas. Os *advices* utilizam o serviço de eventos para registrarem a ocorrência de eventos, interessados em eventos e tipos de eventos que o sistema pode receber. O serviço de eventos tem a responsabilidade de armazenar e gerenciar as inscrições e entregar eficientemente as notificações.

Capítulo 5

Arquitetura e Implementação do AO-OiL

O objetivo do presente capítulo é apresentar os detalhes de implementação do AO-OiL. Serão abordados os vários aspectos envolvidos no processo de desenvolvimento do middleware. Esse capítulo está organizado da seguinte forma: as seções 5.1 e 5.2 apresentam as sub-arquiteturas básicas do AO-OiL: a arquitetura servidora e a arquitetura cliente. A seção 5.3 apresenta a API principal do AO-OiL. Finalmente, a seção 5.4 apresenta os aspectos desenvolvidos no curso do presente trabalho.

O processo de implementação do AO-OiL iniciou-se a partir da definição e implementação do *microkernel* e da montagem da interface a ser disponibilizada para o programador. Em seguida, foram refatorados os elementos envolvidos na implementação dos conceitos de distribuição e coordenação no OiL. A refatoração buscou diminuir o acoplamento desses módulos em relação ao *kernel* do middleware através do uso de aspectos. No AO-OiL esses elementos não são conectados diretamente à arquitetura do middleware. O processo de composição desses elementos com a arquitetura do OiL é feito por meio de aspectos. O AO-OiL fornece um conjunto de aspectos que redefinem o comportamento das operações básicas do *Microkernel* e adicionam novas funcionalidades.

A arquitetura do AO-OiL, apresentada na Figura 5.1, é baseada no padrão de projeto *Microkernel*, logo ela fornece apenas os serviços essenciais do middleware e abstrações gerais que permitem a incorporação de extensões. Os principais objetivos dessa arquitetura são: (i) ser pequena, (ii) suportar carga dos componentes em tempo de execução, (iii) prover um modelo simples de desenvolvimento de *plugins* para componentes que atuem no *microkernel* e (iv) permitir o suporte a aspectos. Dentre essas características citadas, o suporte a aspectos tem um papel crucial no desenvolvimento deste trabalho, pois os aspectos permitem a adição de extensões na arquitetura, além de permitirem a personalização dos serviços de acordo com o contexto

da aplicação. Portanto, o processo de personalização do middleware ocorre através da chamada *personalização incremental*. A personalização incremental é o processo de personalização que ocorre através de adição de funcionalidades na arquitetura.

Através da Figura 5.1 é possível observar que a arquitetura do AO-OiL pode ser divida em duas sub-arquiteturas: arquitetura servidora e arquitetura cliente. A arquitetura servidora gerencia a criação, o registro, associações e o ciclo de vida dos *servants*. Enquanto que, a arquitetura cliente fornece meios para acessar as funcionalidades básicas do middleware e os serviços exportados pelos *servants* de forma transparente.

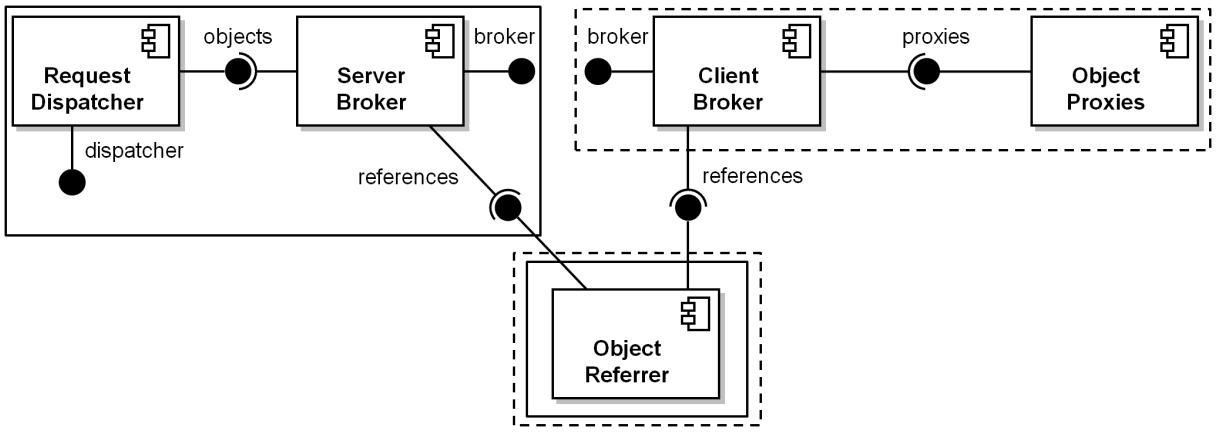


Figura 5.1: A arquitetura base do AO-OiL.

Os elementos envoltos por um quadrado de linhas cheias na Figura 5.1 formam a estrutura da arquitetura servidora do AO-OiL. Fazem parte dessa arquitetura estes elementos: *Server Broker*, *Request Dispatcher* e o *Object Referrer*. Enquanto que os elementos que compõem a infra-estrutura da arquitetura cliente estão envolvidos por um quadrado com linhas tracejadas na mesma figura. Dessa maneira, a arquitetura cliente é composta pelos seguintes elementos: *Client Broker*, *Object Proxies* e o *Object Referrer*. As seções 5.1 e 5.2 descrevem em detalhes os elementos e as funcionalidades providas por cada arquitetura.

O componente *Object Referrer* é o único componente que está presente em ambas as arquiteturas. Esse componente tem por finalidade permitir a conversão de um endereço de memória em uma referência para uma construção Lua e o processo inverso, ou seja, a conversão de uma referência para um endereço de memória. Esse componente faz-se necessário uma vez que em Lua não é possível obter uma referência para um objeto a partir de um endereço de memória. O *Object Referrer* exporta a faceta *references* que fornece meios para codificar e decodificar referências.

5.1 A Arquitetura Servidora

O *Server Broker* e *Request Dispatcher* formam o núcleo da arquitetura servidora. O *Server Broker* é responsável por registrar novos *servants* e controlar o processamento dos mesmos. Pode-se afirmar que esse componente representa o contato do lado servidor para o desenvolvedor. Ele exporta a faceta *broker* que é a interface de controle do servidor, oferecendo as operações de inicialização, execução, desligamento, suspensão dentre outras. O *Server Broker* possui o receptáculo *references* que solicita os serviços oferecidos pelo componente *Object Referrer*. A Tabela 5.1 resume as funcionalidades do *Server Broker*.

Método	**Descrição**
initialize()	Inicializa o servidor, componente *Server Broker*.
object(impl,key)	Cria um novo *servant* a partir da tabela *impl*, a qual contém a implementação do *servant*. O parâmetro opcional *key* pode ser usado para criar referências persistentes.
tostring(object)	Retorna uma referência textual que identifica o *servant*.
pending()	Verifica se existem requisições pendentes.
step()	Processa apenas uma requisição por chamada.
run()	Inicia o processamento de todas as requisições continuamente.
shutdown()	Desliga o servidor, tratando todas as requisições pendentes e fechando todas as conexões.

Tabela 5.1: Funcionalidades do componente *Server Broker*

O componente *Request Dispatcher* mantém um repositório de *servants* ativos e realiza os despachos dos serviços solicitados, invocando os métodos que implementam tais serviços nos objetos apropriados. Esse repositório de objetos é uma tabela *hash* que associada a cada *servant* uma chave que o identifica de maneira unívoca no repositório de objetos. O componente *Request Dispatcher* possui duas facetas: *object* e *dispatcher*. A faceta *object* registra cada objeto, associando um identificador único a cada um deles. O objeto registrado deve ser uma construção Lua que contém todas as funções exportadas por esse objeto na sua interface, caso ela exista. Por sua vez, a faceta *dispatcher* tem a finalidade de recuperar a referência de um objeto a partir de seu identificador. Isso é feito por meio da função *dispatcher* da faceta em questão. A Tabela 5.2 resume as operações disponibilizadas pelo componente em questão.

O processo de inicialização do middleware é implementado pelo método *initialize* da faceta *broker* do componente *Server Broker*. Esse procedimento inicializa os componentes *Server Broker* e *Request Dispatcher*, além de permitir a carga de módulos no momento da criação do middleware. Para evitar que o desenvolvedor esqueça de inicializar corretamente as estruturas do

Método	**Descrição**
register(impl, key, ...)	Associa o novo objeto *servant impl* a chave *key*.
unregister(key)	Remove o objeto *servant* identificado por *key*
retrieve(key)	Recupera um objeto *servant* a partir de uma dada chave
dispatcher(key, operation, default, ...)	Realiza a invocação da operação *operation* no *servant* associado a *key*. Caso tal operação não exista, a ação padrão, denotada pela variável *default*, é invocada.

Tabela 5.2: Funcionalidades do componente *Request Dispatcher*

lado do servidor, o AO-OiL implicitamente chama o processo de inicialização durante a criação do mesmo. Dessa maneira, o processo de criação de novas instâncias do middleware e o processo de inicialização das mesmas ocorrem concomitantemente.

A Figura 5.2 ilustra os passos necessários para realizar a criação um *servant* no AO-OiL.

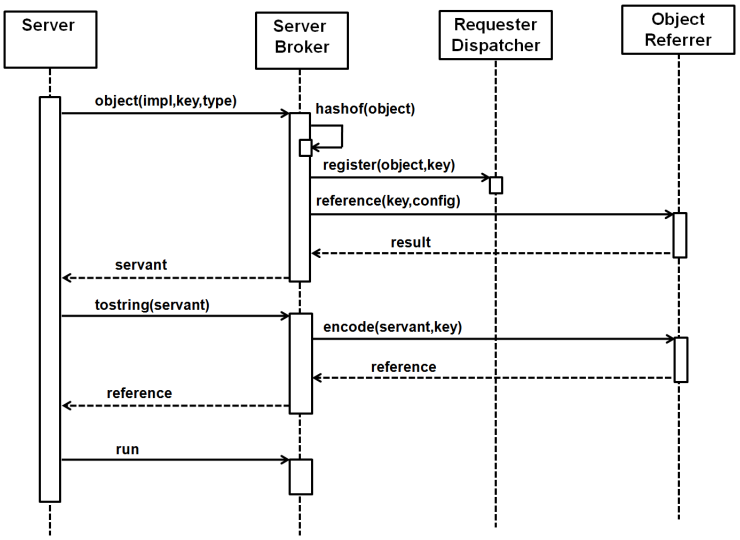


Figura 5.2: Criação e registro de um *servant*

Os passos para a realização da criação e do registro dos componentes são enumerados a seguir:

1. Para criar um *servant* no AO-OiL, usa-se função *object* da faceta *broker* do componente *Server Broker*. Esse método recebe uma tabela, que contém a implementação do *servant* e opcionalmente uma chave que a identifica. Caso não seja passado nenhuma chave para o método em questão, o mesmo se encarrega de gerar uma chave que identifique tal tabela, através do método *hashof*.

2. Em seguida é realizado o registro do *servant*. O *Server Broker* invoca a função *register* da faceta *objects*, exportado pelo componente *Request Dispatcher*, passando a chave e o objeto e para obter o *servant*, que é retornado ao controle do desenvolvedor da aplicação. O método *register* insere o par (chave, *servant*) no repositório de *servant*.

3. Os clientes precisam ter uma a forma de se contactar aos *servants*, logo se faz necessário criar referências para os objetos *servants*. Essas referências são criadas através do método *referenceto* da faceta *references* do *Object Referrer*. O método *referenceto* cria as referências a partir do identificador do objeto que está sendo registado.

Para iniciar o processamento das invocações, o AO-OiL invoca a função *run* da faceta *broker*. Neste momento, o AO-OiL fica a espera de requisição de serviço aos *servants* instalados nele. Para invocar um método desejado por uma solicitação de serviço, o AO-OiL invoca o método *dispatch* da faceta *dispatcher* do componente *Request Dispatcher*, passando a operação a ser invocada e seus parâmetros.

5.2 A Arquitetura Cliente

A arquitetura cliente é formada por três elementos: O *Client Broker*, *Object Proxies* e o *Object Referrer*. Como este componente já foi anteriormente explicado, essa seção irá focar-se nos dois primeiros componentes dessa sub-arquitetura do AO-OiL.

O *Client Broker*, é o componente responsável por criar os objetos *proxies*, através do receptáculo *proxies*, a partir de informações extraídas das formas textuais pelo *Object Referrer* por meio do receptáculo *references*. A Tabela 5.3 resume as operações disponibilizadas pelo componente *Client Broker*.

Por sua vez, o componente *Object Proxies* é uma fábrica de objetos *proxies*. A função *proxyto*, exportada pela faceta *proxies* do componente em questão, retorna um objeto *proxy* mediante a passagem de uma referência decodificada. O acesso de um cliente a um *servant* é feito através de requisições de operações executadas sobre o *proxy* associado ao *servant* em questão.

Método	**Descrição**
proxy(reference)	Cria um *proxy* a partir de uma referência para o objeto *servant*.
fromstring(reference)	Transforma uma *string*, referência textual, em uma referência que contém informações sobre o objeto que o *proxy* irá representar.

Tabela 5.3: Funcionalidades do componente *Client Broker*

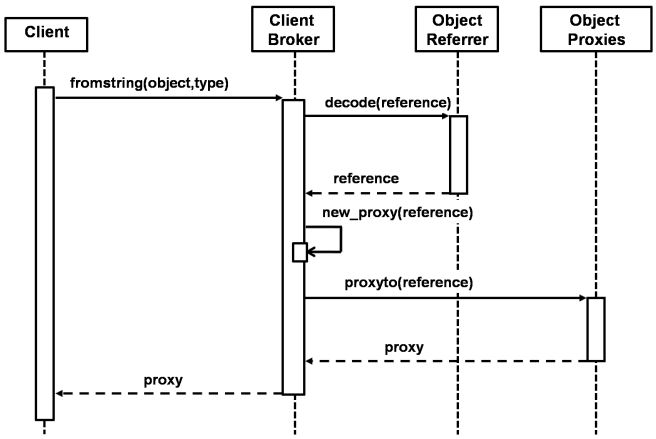


Figura 5.3: Processo de criação de um *proxy*.

Uma invocação pode ser divida em duas partes: a criação de um *proxy* para o *servant* que implementa os serviços solicitados e a execução da invocação sobre o *proxy*. A Figura 5.3 ilustra o processo de criação de um *proxy*. Os passos de criação um *proxy* são os seguintes:

1. O cliente pede ao componente *Client Broker*, através do método *fromstring* da faceta *broker*, um *proxy* para o *servant* a partir de uma referência textual para o mesmo.

2. O *Client Broker* comunica-se com a faceta *references* do componente *Object Referrer* para decodificar a referência textual em uma referência para o *servant*.

3. De posse da referência decodificada, o *Client Broker* a usa para invocar o método *proxyto* da faceta *proxies* do componente *Object Proxies* para a criação do *proxy*. Esse é, então, retornado ao cliente, que o usará para fazer requisições de serviços ao *servant* representado pelo *proxy*.

O diagrama de seqüência da Figura 5.4 ilustra o caminho de execução de uma invocação do lado do cliente. Os passos realizados para efetivamente executar uma requisição do lado cliente são os seguintes:

1. Quando o cliente realiza uma chamada sobre o objeto *proxy*, essa chamada é capturada por ele e transformada em uma chamada ao método *dispatch* da faceta *dispatcher* do componente *Request Dispatcher*.

2. O *Request Dispatcher* realiza a invocação da operação solicitada e devolve o resultado ao *proxy*.

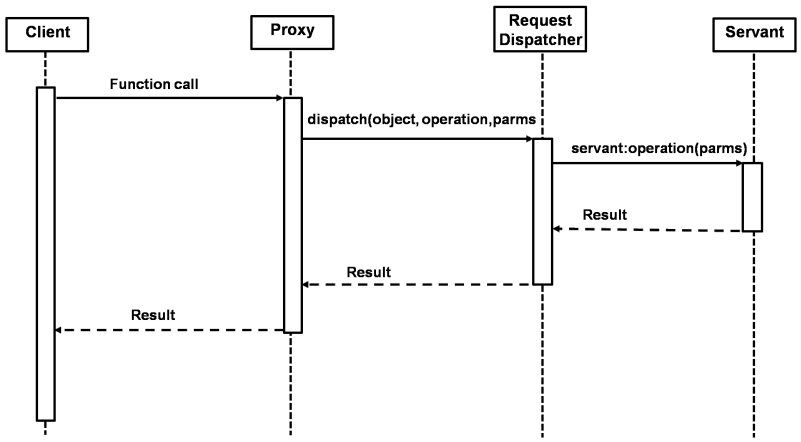


Figura 5.4: Caminho de execução de uma invocação

3. O *proxy* verifica se a resposta é uma construção Lua válida, caso seja o resultado é retornado para a aplicação, senão uma exceção é gerada.

5.3 A API principal do AO-OiL

Evidentemente, a direta manipulação das API's que implementam as sub-arquiteturas do AO-OiL é indesejável. Além de ser propensa a erros, devido a estrita seqüência de invocações que deve ser realizado, é um processo que exige conhecimento detalhado das arquiteturas. Visando facilitar a utilização, o AO-OiL oferece uma API principal de mais alto nível. Essa API fornece mecanismos para acessar as funcionalidades básicas do AO-OiL: criação e destruição de *servants*, *proxies*, e referências, inicialização, execução, suspensão e desligamento do middleware, criação de associações entre componentes e carga de módulos. Essa funcionalizadas são implementadas pela classe **KERNEL**. A Figura 5.5 ilustra o diagrama de classes completo da arquitetura base do AO-OiL. A classe **KERNEL** representa uma instância do AO-OiL. A Tabela 5.4 resume a API fornecida por ela.

5.4 Aspectos

Essa seção apresenta como os conceitos transversais de distribuição e coordenação são implementados no AO-OiL.

O conceito de distribuição é um conceito que tende a ser bastante complexo, pois além da grande variedade de protocolos de comunicação que precisam ser codificados, ele pode interagir

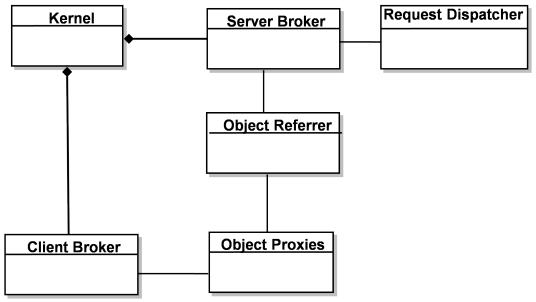


Figura 5.5: Diagrama de classes do AO-OiL.

com outros interesses como segurança, transporte, mobilidade e escalabilidade. A implementação desse conceito como aspecto evita a codificação de vários protocolos dentro das fábricas de componentes.

O conceito de coordenação estabelece as regras do processo de interação entre os componentes tanto no nível de aplicação quanto no nível de middleware. Idealmente, os componentes não devem decidir por eles mesmos como o processo de coordenação deve ser feito. Para atingir esse requisito, o protocolo de coordenação deve estar totalmente separado do código dos componentes. A modelagem da coordenação como um aspecto permite separá-la do código dos componentes. Conseqüentemente, diferentes protocolos de coordenação podem ser programados para controlar a coordenação de diferentes tipos de componentes.

Método	**Descrição**
KERNEL:init()	Inicializa o AO-OiL
KERNEL:run()	Inicia o processamento de todas as requisições continuamente.
KERNEL:pending()	Verifica se existem requisições pendentes.
KERNEL:step()	Processa apenas uma requisição por chamada.
KERNEL:shutdown()	Desliga o servidor, tratando todas as requisições pendentes e fechando todas as conexões.
KERNEL:new_servant()	Instância novos *servants*
KERNEL:delete_servant()	Destroí *servants* criados
KERNEL:load()	Instância novos *proxies*
KERNEL:unload()	Destroí *proxies*
KERNEL:bind()	Cria associações entre componentes
KERNEL:unbind()	Destrói associações entre componentes
KERNEL:tostring()	Cria uma referência textual para um *servant*
KERNEL:load_module()	Realiza a carga dinâmica de módulos

Tabela 5.4: Interface principal de programação do AO-OiL

5.4.1 Distribuição

A implementação do conceito de distribuição como um serviço aspectual do middleware exige dois conjuntos de aspectos. O primeiro conjunto de aspectos irá interceptar a instância do middleware ao qual se deseja adicionar a distribuição, a fim de obter parâmetros de configuração, como porta, *host* dentre outros, e para realizar a introdução de novos procedimento. Enquanto que o segundo conjunto de aspectos irá interceptar as invocações realizadas nos *proxies* com o objetivo de transformar uma chamada local em uma chamada remota.

Para mostrar a flexibilidade e o alto grau de personalização da arquitetura proposta, o AO-OiL fornece módulos aspectuais que implementam o conceito de distribuição através de diferentes infra-estruturas de distribuição. Um dos módulos adiciona o protocolo CORBA, enquanto que o outro adiciona o protocolo LuDo [Maia et al., 2005].

O projeto do AO-OiL também enfatiza o problema da complexidade interna dos *middlewares*, ou seja, complexidade relacionada ao desenvolvimento do próprio middleware e de extensões que atuem sobre ele. Unindo essa filosofia ao fato de que o conceito de distribuição para as plataformas de middleware é fundamental, o AO-OiL disponibiliza um conjunto de componentes cujo objetivo é facilitar o desenvolvimento e a integração de protocolos de comunicação. Esse conjunto de componentes constituem a camada denominada de *distribution*. A referida camada é formada por estes elementos: *Acceptor*, *Operation Invoker*, *Request Receiver* e *Connector*. As funcionalidades desempenhadas pelo *Operation Invoker* e pelo *Request Receiver*, descritas na seção 2.2.2, foram mantidas, contudo o código desses componentes sofreram pequenas modificações para aproveitar a estrutura da nova arquitetura. O componente *Acceptor* gerencia a criação de canais de comunicação e a aceitação de requisições do lado do servidor. O processo de aceitação de requisições inclui a espera por atividades nos canais de comunicação. O *Acceptor* funciona de maneira serializada, ou seja, quando ele obtém uma requisição, ele a executa diretamente. Por fim, o *Connector* administra e cria os canais de comunicação do lado do servidor.

A seguir, são apresentados em detalhes as implementações dos módulos de suporte aos protocolos CORBA e LuDo.

5.4.1.1 Corba

O aspecto **Corba** oferece duas interfaces aspectuais, *icorba_server* e *icorba_client*, que oferecem um conjunto de pontos de junção abstratos que permitem a implementação de funcionalidades relacionadas, respectivamente, com aspectos do lado do servidor e do lado do cliente da especificação CORBA.

A interface *icorba_server* oferece dois pontos de junção abstratos que permitem:

- Personalizar a criação dos *servants*.
- Criar referências para *servants*.
- Criar referências textuais a partir de *servants*.
- Personalizar o processamento de requisições do middleware.

Além disso, essa interface introduz novas funcionalidades necessárias para a verificação de tipos requirida pelo protocolo CORBA e pontos de junções abstratos para a configuração do processo de carga do próprio aspecto. Além disso, o aspecto **Corba** também oferece dois aspectos: *corba_server* e *corba_client* que implementam as interfaces citadas, respectivamente.

```
1  local corba_server = Aspect:new('CorbaServer')
2  local iserver = AspectInterface:new( {name = 'icorba_server'},
3      {
4          {refine = 'new', action = CorbaServer.new},
5          {refine = 'init', action = CorbaServer.init },
6          {refine = 'run', action = CorbaServer.run },
7          {refine = 'newservant', action = CorbaServer.new_servant },
8          {refine = 'tostring', action = CorbaServer.tostring },
9          {refine = 'referenceto', action = CorbaServer.referenceto },
10         {refine = 'dispatch', action = CorbaServer.dispatch },
11         {refine = 'loadidl', action = CorbaServer.loadidl },
12         {refine = 'loadidlfile', action = CorbaServer.loadidlfile },
13         {refine = 'getLIR', action = CorbaServer.getLIR },
14         {refine = 'getIR', action = CorbaServer.getIR },
15         {refine = 'setIR', action = CorbaServer.setIR },
16         {refine = 'newencoder', action = CorbaServer.newencoder },
17         {refine = 'newdecoder', action = CorbaServer.newdecoder },
18         {refine = 'newexcept', action = CorbaServer.newexcept },
19         {refine = 'aspects',action = CorbaServer.load_aspects},
20     }
21 )
22 corba_server:interface(iserver)
```

Listagem 5.1: Definição da interface *icorba_server*

A Listagem 5.1 exibe o código responsável pela definição da interface *icorba_server* e do aspecto *corba_server*. Na primeira linha, o aspecto **Corba Server** é criado. Em seguida, nas linhas 2 a 21, é definido o conjunto de pontos de junção abstratos, refinamentos, que forma a interface *icorba_server*. Por exemplo, na linha 10, o refinamento *dispatch* tem como função

transformar uma invocação local em remota, e na linha 11, o refinamento *loadidl* introduz no AO-OiL um procedimento que realiza a carga das interfaces *IDL*. Finalmente, a interface em questão é associada ao aspecto recém criado, linha 22. A Tabela 5.5 apresenta uma breve descrição da motivação da criação de cada ponto de junção dessa interface.

Ponto de junção abstrato	**Descrição**
new	Criação dos componentes CORBA
init	Inicialização dos componentes Corba do lado do servidor
run	Inicia o processamento de todas as requisições continuamente
newservant	Criação de *servant*
referenceto	Criação de referências remotas
tostring	Criação de referências textuais
dispatch	Transformação de invocações locais em remotas
loadidl	Carga de interfaces IDL Corba
loadidlfile	Carga de interfaces IDL Corba a partir de arquivos
getIR	Acesso ao repositório de interfaces remoto
getLIR	Acesso ao objeto que implementa o repositório de interfaces remoto
setIR	Configuração do repositório de interfaces remoto
newencoder	Criação de novos codificadores de tipos Corba
newdecoder	Criação de novos decodificadores de tipos Corba
newexcept	Criação de novas exceções Corba

Tabela 5.5: Descrição dos pontos de junção abstratos da interface *icorba_server*

Por sua vez, a interface *icorba_client* oferece um conjunto de pontos de junção abstratos que permitem adaptar o processo de criação dos *proxies* e o processo de conversão de referências textuais em referências para *servants*. A Listagem 5.2 apresenta a implementação da interface *icorba_client*. Na primeira linha, o aspecto *CorbaClient* é criado. Em seguida, nas linhas 2 a 10, é definido o conjunto de refinamentos que forma a interface. Finalmente, na linha 11, a interface em questão é associada ao aspecto *CorbaClient*.

A Tabela 5.6 resume a funcionalidade de cada ponto de junção abstrato definido por essa interface.

Ponto de junção abstrato	**Descrição**
init	Inicialização da arquitetura cliente
fromstring	Transforma uma referência textual em uma referência para o *servant*
proxy	Criação de *proxies*

Tabela 5.6: Descrição dos pontos de junção abstratos da interface *icorba_client*

```
local corba_client = Aspect:new('CorbaClient')
local iclient = AspectInterface:new(
    {name = 'icorba_client'},
    {
        {refine = 'init',action = CorbaClient.init},
        {refine = 'fromstring',action = CorbaClient.fromstring},
        {refine = 'proxy',action = CorbaClient.proxy},
        {refine = 'aspects',action = CorbaClient.load_aspects},
    }
)
corba_client:interface(iclient)
```

Listagem 5.2: Definição da interface *icorba_client*

O AO-OiL permite a carga dos aspectos de duas formas: estática ou dinâmica. A carga estática ocorre durante o processo de criação do middleware. A carga dinâmica permite carregar o aspecto em qualquer momento durante o ciclo de vida da aplicação através da invocação da função *load_module* da API do AO-OiL. A Figura 5.6 ilustra esse processo. Ao ser invocada, a função *load_module* carrega toda a infra-estrutura relacionada ao protocolo CORBA.

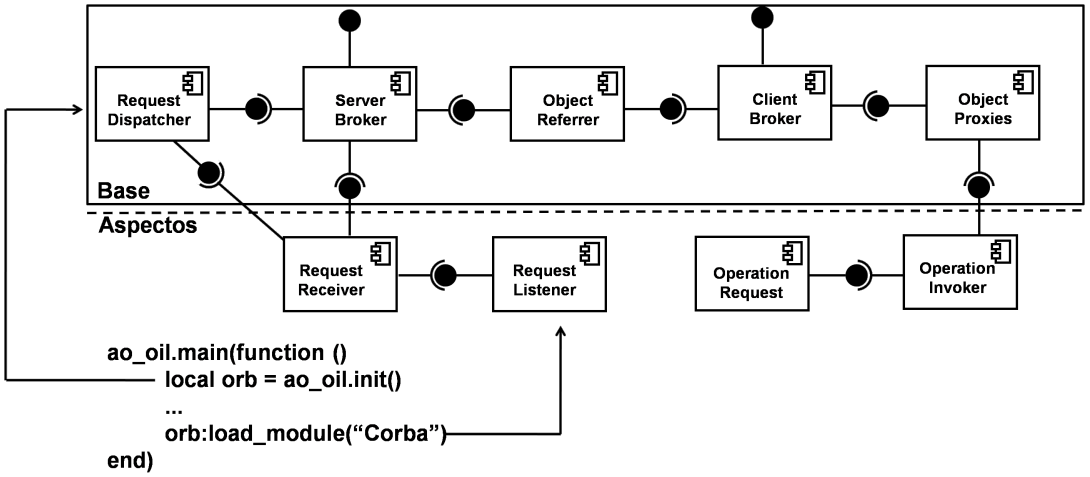


Figura 5.6: Carga dinâmica de módulos do AO-OiL

Caso o desenvolvedor opte por uma carga estática, a definição do aspecto é carregada antes da instanciação dos componentes que formam a infra-estrutura básica do middleware, ou seja, apenas os elementos da classe **KERNEL** estão definidos nesse momento. Logo, ou define-se um ponto de junção abstrato, que atua após a criação dos componentes básicos do AO-OiL, ou

define-se um ponto de junção antecipado. Este trabalho optou por definir um ponto de junção abstrato, denominado *aspects*, em detrimento de um ponto de junção antecipado, pois esse tipo de ponto de junção envolve a criação de monitor que teria uma vida muito curta, pois o ciclo de vida de um monitor termina no momento em que o elemento monitorado é definido. A função monitorada no caso é *ao_oil.init*, que é responsável pela criação e inicialização dos componentes básicos AO-OiL. Porém, acontece que essa função é a primeira a ser invocada para que se possa utilizar o middleware.

A Listagem 5.3 exibe como ocorre o processo de carga estática do AO-OiL. Primeiramente é carregado o módulo *ao_oil* que representa a plataforma. Na segunda linha é carregado o aspecto Corba. Em seguida, é definida a função *ao_oil.main* que contém o código da aplicação. A linha 4 invoca a função *load_aspect* que instancia o ponto de junção abstrato *aspects* definido na interface *icorba_server*. Esse ponto de junção atua após a invocação da função *ao_oil.init*, carregando os elementos relacionados ao aspecto de distribuição.

```
1 require "ao_oil"
2 local remote = require "ao_oil.aspects.Corba"
3 ao_oil.main(function()
4     remote.load_aspect()
5     local orb = ao_oil.init()
6 end)
```

Listagem 5.3: Processo de carga estática

O objetivo das interfaces providas pelo aspecto **Corba** é permitir que os desenvolvedores possam adaptar o aspecto às suas necessidades. Contudo, o aspecto **Corba** não deixa de fornecer um conjunto de pontos de junção concretos que anexam a infra-estrutura de distribuição de CORBA ao AO-OiL. A Listagem 5.4 apresenta a instanciação dos principais pontos de junção abstratos do aspecto **Corba**.

As linhas de 3 a 10 da Listagem 5.4 definem pontos de junção que introduzem novas funcionalidades no AO-OiL, representado pela classe *KERNEL* dentro do módulo ao_oil. O *designator introduction* denota que um *advice* será introduzido como um método em um objeto. Dessa maneira, as linhas de 3 a 10 adicionam o seguinte conjunto de funções: *loadidl* - carga de interfaces IDL Corba, *loadidlfile* – carga de interfaces IDL Corba a partir de um arquivo, *getLIR* - acesso ao repositório de interfaces remoto, *getIR* - acesso ao objeto que implementa o repositório de interfaces remoto, *setIR* - configuração do repositório de interfaces remoto, *newexcept* - criação de novas exceções Corba, *newencoder* - criação de novos codificadores de tipos Corba e *newdecoder* - criação de novos decodificadores de tipos Corba.

```
1  corba_server.icorba_server["tostring"]= {expression='call(ao_oil.KERNEL.
       tostring)',type='around'}
2  corba_server.icorba_server["run"]= {expression='call(ao_oil.KERNEL.run)',
       type='around'}
3  corba_server.icorba_server["loadidl"]= {expression='introduction(ao_oil.
       KERNEL.loadidl)'}
4  corba_server.icorba_server["loadidlfile"]= {expression='introduction(ao_oil.
       KERNEL.loadidlfile)'}
5  corba_server.icorba_server["getLIR"]= {expression='introduction(ao_oil.
       KERNEL.getLIR)'}
6  corba_server.icorba_server["getIR"]= {expression='introduction(ao_oil.KERNEL
       .getIR)'}
7  corba_server.icorba_server["setIR"]= {expression='introduction(ao_oil.KERNEL
       .setIR)'}
8  corba_server.icorba_server["newexcept"]= {expression='introduction(ao_oil.
       KERNEL.newexcept)'}
9  corba_server.icorba_server["newencoder"]= {expression='introduction(ao_oil.
       KERNEL.newencoder)'}
10 corba_server.icorba_server["newdecoder"]= {expression='introduction(ao_oil.
       KERNEL.newdecoder)'}
11 corba_server.icorba_server["newservant"]= {expression='call(ao_oil.kernel.
       Server.object)',type='before'}
12 corba_server.icorba_server["dispatch"]= {expression='call(ao_oil.kernel.
       Dispatcher.dispatch)',type='before'}
13 corba_server.icorba_server["referenceto"]= {expression='call(ao_oil.kernel.
       Referrer.referenceto)',type='around'}
14
15 corba_client.icorba_client["proxy"] = {expression= 'call(ao_oil.kernel.
       Client.proxy)', type = 'around'}
16 corba_client.icorba_client["load"] = {expression= 'call(ao_oil.KERNEL.load)'
       , type = 'around'}
```

Listagem 5.4: Instanciação do aspecto **Corba**

As demais linhas interceptam chamadas de métodos nas diversas classes do sistema adicionando novos comportamentos ou substituindo um comportamento existente. As linhas de 2 a 13, definem os aspectos que atuam do lado do servidor, enquanto que da linha 14 a 15 definem aspecto que atuam do lado do cliente. A linha 2 intercepta invocações, denotado pelo *designator call*, ao método *run* da classe *ao_oil.KERNEL*, o tipo de *advice around* indica que um novo comportamento será definido para o elemento interceptado. A linha 11 define que um *advice* irá

atuar antes, *advice* do tipo *before*, das invocações, *designator call*, ao método *object* da classe *Server* do módulo *ao_oil.kernel*.

A linha 12 define um aspecto que atua antes, *advice* do tipo *before*, das invocações, *designator call*, ao método *dispatch* do objeto *ao_oil.kernel.Dispatcher.dispatch*. Em seguida, na linhas 13, o refinamento *referenceto* intercepta as invocações ao método *ao_oil.kernel.Referrer.referenceto* redefinindo o seu comportamento. Trata-se, portanto, de um *advice* do tipo *around*. A linha 15 define um aspecto que define um novo comportamento para o método *proxy* da classe *ao_oil.kernel.Client*. Finalmente, na linha 16, o refinamento *load* define um aspecto que intercepta o carregamento de componentes do AO-OiL, função *ao_oil.kernel.load*.

5.4.1.2 LuDo

LuDO (*Lua Distributed Objects*) [Maia et al., 2005] é uma tecnologia de RMI especificamente projetada para a linguagem Lua. O objetivo do LuDo é ser o mais simples possível, de maneira que os desenvolvedores possam facilmente utilizá-lo e estendê-lo. Uma das principais características do LuDO é que ele não oferece suporte a interfaces ou assinatura de métodos, ele confia na tipagem dinâmica para realizar as invocações remotas. Dessa maneira, os métodos remotos podem ser invocados com qualquer número de parâmetros e podem retornar qualquer número de valores. Outra característica do protocolo LuDo que vale ressaltar é o suporte a *multithreading*, através de co-rotinas.

O aspecto **LuDo** é formado pelas interfaces *iludo_server* e *iludo_client*, e pelos aspectos, *ludo_server* e *ludo_client* que respectivamente implementam essas interfaces. A primeira interface contém uma lista de refinamentos voltados para eventos do lado do servidor, enquanto que a segunda interface contém refinamentos que estão relacionados a eventos do cliente.

```
1  local iludo_server = AspectInterface:new(
2      {name = 'iludo_server'},
3      {
4          {refine = 'new', action = LudoServer.new },
5          {refine = 'init', action = LudoServer.init },
6          {refine = 'run', action = LudoServer.run },
7      }
8  )
9  local ludo_server = Aspect:new('LudoServer')
10 ludo_server:interface(iludo_server)
```

Listagem 5.5: Definição da interface *iludo_server*

A Listagem 5.5 apresenta a definição da interface *iludo_server* e do aspecto *ludo_server* que a implementa. As oito primeiras linhas definem o conjunto de pontos de junção abstratos fornecidos pela interface em questão. A linha 9 cria o aspecto *iludo_server*, e em seguida, na linha 10, ele é associado à interface *iludo_server*. A interface *iludo_server* fornece três pontos de junção abstratos: *new*, *init* e *run*. O primeiro permite personalizar a rotina de criação dos componentes que formam a infra-estrutura do LuDo. O segundo permite configurar o processo de inicialização de tais componentes. Por fim, o ponto de junção abstrato *run* implementa o procedimento de iniciar o processamento de requisições do middleware.

Do mesmo modo, a Listagem 5.6 apresenta a definição da interface *iludo_client* e do aspecto *ludo_client* que a implementa. Nas sete primeiras linhas, é definido o conjunto de pontos de junção abstratos fornecidos pela interface. Tal conjunto é formado pelos seguintes elementos: *new* e *proxy*. O primeiro permite personalizar a rotina de criação dos componentes que formam a infra-estrutura do LuDo do lado do cliente. O segundo implementa o procedimento para a criação de *proxies*. Em seguida, nas linhas 9 e 10, é criado o aspecto *ludo_client* e a associação entre *iludo_client* e *ludo_client* é estabelecida.

```
1 local iludo_client = AspectInterface:new(
2     {name = 'iludo_client'},
3     {
4         {refine = 'new', action = LudoClient.new },
5         {refine = 'proxy', action = LudoClient.proxy },
6     }
7 )
8 ludo_client = Aspect:new('LudoClient')
9 ludo_client:interface(iludo_client)
```

Listagem 5.6: Definição da interface *iludo_client*

Assim como o aspecto **corba**, o aspecto **LuDo** fornece um conjunto de pontos de junção concretos que permitem anexar o protocolo LuDo no AO-OiL. A Listagem 5.7 apresenta esse conjunto de pontos. Os pontos de junção que atuam no lado do servidor são definidos nas linhas 1 a 3, interface *iludo_server*. Na primeira linha é definido o refinamento *new*, o *designator callone* indica que o *advice* relacionado a esse aspecto atua apenas na primeira chamada à função *ao_oil.KERNEL.init*. O campo *type* indica o momento em que o *advice* atua, no caso o *advice* atua antes da função de inicialização do AO-OiL. Na linha 2 é definido o refinamento *init* que também atua antes da função *ao_oil.KERNEL.init*. Entretanto o *designator call* indica que o aspecto atua em toda invocação à referida função, ao invés de atuar apenas na primeira chamada como ocorre com o *designator callone*. Finalizando os aspectos do lado do servidor, o refina-

mento *run* atua em todas as invocações ao método *ao_oil.KERNEL.run*. O *designator around* indica que um novo comportamento é atribuído a função em questão.

Do lado do cliente, interface *iludo_client*, dois refinamentos são definidos: *new* e *proxy*. O primeiro intercepta na primeira invocação, *designator callone*, à função *ao_oil.KERNEL.init* para redefinir o seu comportamento, *advice* do tipo *around*. O segundo refinamento atua durante as invocações ao método *ao_oil.KERNEL.new_proxy*, *designator call*, redefinindo a maneira como os *proxies* são criados.

```
1 ludo_server.ludo["new"]= {expression='callone(ao_oil.KERNEL.init)',type='
    before'}
2 ludo_server.ludo["init"]= {expression='call(ao_oil.KERNEL.init)',type='
    before'}
3 ludo_server.ludo["run"]= {expression='callone(ao_oil.KERNEL.run)',type='
    around'}
4 --client
5 ludo_client.ludo["new"]= {expression='callone(ao_oil.KERNEL.init)',type='
    around'}
6 ludo_client.ludo["proxy"]= {expression='call(ao_oil.KERNEL.load)',type='
    around'}
```

Listagem 5.7: Instanciação do aspecto *LuDo*

5.4.2 Coordenação

Como foi dito no capítulo 4, o modelo de coordenação adotado pelo AO-OiL é o *Publisher/-Subscriber*. Esse modelo requer um sistema de evento que permita o registro dinâmico de objetos e a entrega eficiente de notificações aos objetos que estão registrados no sistema.

CORBA especifica um serviço de eventos baseado na filosofia do *Publisher/Subscriber* para oferecer suporte a aplicações que necessitam de comunicação assíncrona. E como o AO-OiL oferece suporte a CORBA, esse trabalho optou por aspectizar o serviço de eventos de CORBA em detrimento a uma modelagem de um sistema de eventos inteiramente novo.

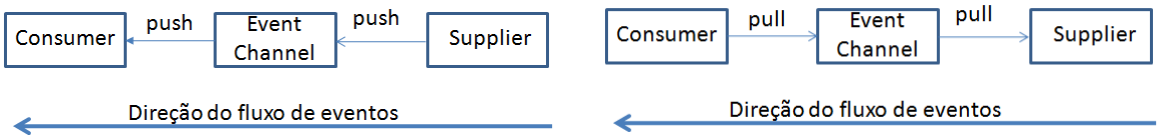
O serviço de eventos de CORBA define dois papeis para os objetos: Os fornecedores, denominado de *Suppliers*, que produzem os eventos e os consumidores, *Consumers*, que processam os eventos. A comunicação entre os fornecedores e os consumidores acontece via um elemento intermediário chamado de canal de eventos (*event channel*). O canal de eventos viabiliza a troca de eventos entre dois ou mais objetos, sem que seja necessário que os objetos conheçam-se mutuamente. Dois modelos de comunicação são disponibilizados: *Push* e *Pull*.

No primeiro modelo, o *Supplier* é o elemento responsável por iniciar a comunicação, em outras palavras, ele é o objetivo ativo. O *Supplier* envia os eventos para o canal de eventos, através do método *push* do canal, e por sua vez, o canal de eventos envia os eventos para os *Consumers* utilizando o método *push* do *Consumers*. Para registrar o seu interesse em algum evento em específico no canal de eventos, um *Consumer* invoca o método *add_push_consumer*. A criação de um *Supplier* do tipo *push* é disponibilizada através da função *add_push_supplier*. Para parar de receber notificações sobre a ocorrência de eventos em canal específico, o *Consumer* utiliza o método *disconnect_push_consumer*. Por fim, o método *disconnect_supplier* é utilizado para desconectar um *Supplier* do canal de eventos.

Por outro lado, no modelo *Pull*, o elemento ativo é o *Consumer*, uma vez que o *Consumer* tem a responsabilidade de verificar se existem eventos a serem processados no canal de eventos. Dessa maneira, o *Consumer* invoca o método *pull* do canal de eventos para solicitar os eventos gerados pelos *Suppliers*. O canal de eventos então invoca o método *pull* nos fornecedores, *Suppliers*, do evento em questão. Esse segundo modelo também fornece primitivas que permitem adicionar e retirar novos *Suppliers*. Para registrar a sua referência e oferecer os seus serviços um *Supplier* utiliza o método *add_pull_supplier*. Enquanto que, para retirar a sua referência do canal de eventos e conseqüentemente parar de receber solicitações de serviços, um *Supplier* invoca o método *disconnect_pull_supplier*. Uma característica importante desse modelo, é o fato de que o método *pull* é bloqueante, ou seja, quem o invoca fica bloqueado esperando a chegada de um evento. Caso a aplicação não deseje esse comportamento, o modelo oferece a primitiva *try_pull* que retorna imediatamente um booleano indicando se existe algum evento.

Uma característica do serviço de eventos de CORBA é que um evento é representando por uma invocação. Uma invocação é direcionada a um tipo de objeto específico, e tem uma semântica bem definida. O serviço de eventos de CORBA garante que todos os consumidores, no modelo *Push*, ou todos os fornecedores, no modelo *Pull*, possuam uma interface compatível para processar a invocação. Em outras palavras, quando um elemento invoca uma operação no canal de eventos, seja ela um *push* ou um *pull*, é garantido que o canal tem a mesma interface dos atuais consumidores ou fornecedores. A Figura 5.7 ilustra o fluxo de evento nos dois modelos citados. A Figura 5.7(a) ilustra o modelo *Push*. Nesse modelo o *Supplier* faz o papel de servidor, enquanto que o *Consumer* faz o papel de cliente. Já no modelo *Pull*, os papéis se invertem e o *Supplier* faz o papel de cliente e o *Consumer* o de servidor como mostra a Figura 5.7(b).

O Primeiro objeto a ser criado é o canal de eventos, pois a sua função é desacoplar os fornecedores e os consumidores. A partir do canal são criados uma instância dos elementos *SupplierAdmin* e *ConsumerAdmin*. Um *SupplierAdmin* é uma fábrica de *proxies* para componentes



(a) Modelo *Push* (b) Modelo *Pull*

Figura 5.7: Modelos de comunicação do serviços de eventos CORBA

Supplier, de modo equivalente, um *ConsumerAdmin* é uma fábrica para *proxies* do tipo *Consumer*. Para obter uma referência a um elemento *SupplierAdmin*, os *Suppliers* invocam a função *for_supplier* do canal de eventos. Analogamente, os elementos *Consumers* invocam a função *for_consumer* do canal de eventos para obter uma referência para o elemento *ConsumerAdmin*.

A estratégia de aspectização adotada para o serviço de eventos é bem simples: o aspecto tem por responsabilidade instanciar os elementos que forneçam suporte ao serviço de eventos quando a aplicação necessitar. Sendo assim, a API do serviço de eventos não é carregada até que a aplicação faça uso da mesma. Porém, como a API não é carregada, os elementos que a compõem não são declarados no programa. Logo para manter a disponibilidade da API do serviço de eventos, é necessário que o aspecto seja capaz de interceptar invocações a esses elementos não declarados.

Nesse contexto, o uso dos pontos de junção antecipados de RE-AspectLua é primordial, pois esse recurso além de tornar possível a interceptação de elementos não declarados, permite aplicar a eles uma ação, que no caso é realizar a carga da biblioteca responsável pelo serviço de eventos. A Listagem 5.8 mostra a definição do aspecto *EventService* que é que é responsável por instanciar o serviço de eventos no AO-OiL.

Primeiramente, são definidas as seguintes funções: *init_for_consumers*, *init_for_suppliers* e *lazy_load*. Tais funções têm um objetivo em comum: realizar a carga preguiçosa, *lazying load*, de componentes que formam a arquitetura do serviço de eventos. A função *init_for_consumers*, definida nas linhas 1 a 6, tem como objetivo carregar o componente *ConsumerAdmin* e gerenciar uma lista de *proxies* do tipo *PushConsumer* criados a partir do referido componente. A biblioteca que contém a definição do componente *ConsumerAdmin* é carregado na linha 2 através da função *loader*. Na linha 3, um contador é criado para gerenciar o número de *proxies* criados. Em seguida, uma instância do *ConsumerAdmin* é criada e retornada para o utilizador, linhas 4 e 5. A função *init_for_suppliers*, linhas 7 a 12, efetua a mesma tarefa; a única diferença é que essa função é responsável por carregar o componente *SupplierAdmin* e gerencia a lista de *proxies* do tipo *PullSupplier*. Por último, a função *lazy_load*, definida nas linhas 13 a 17, realiza a carga

do canal de eventos. Após a carga do canal, *lazy_load* invoca a função *load_aspects* para criar dois pontos de junção antecipados que atuam respectivamente na primeira chamada aos métodos *for_consumers* e *for_suppliers*. Dessa maneira, os elementos *ConsumerAdmin* e *SupplierAdmin* também são carregados sob-demanda.

```
1  function init_for_consumers(self)
2      local ConsumerAdmin = loader("ConsumerAdmin")
3      self.push_consumer_count = 0
4      self.consumer_admin = ConsumerAdmin(self)
5      return self.consumer_admin
6  end
7  function init_for_suppliers(self)
8      local SupplierAdmin = loader("SupplierAdmin")
9      self.push_supplier_count = 0
10     self.supplier_admin = SupplierAdmin(self)
11     return self.supplier_admin
12 end
13 function lazy_load()
14     local event = require "ao_oil.corba.services.event"
15     load_aspects()
16     return event
17 end
18 local event_aspect = Aspect:new('EventService')
19 local ievent_aspect = AspectInterface:new(
20     {name = 'event'},
21     {
22         {refine = 'lazy_load', action = lazy_load },
23         {refine = 'init_for_consumers', action = init_for_consumers },
24         {refine = 'init_for_suppliers', action = init_for_suppliers },
25     }
26 )
27 event_aspect:interface(ievent_aspect)
28 event_aspect.event["lazy_load"] = {expression='callone(ao_oil.corba.services
       .event.*)',type='before'}
29 function load_aspects()
30     event_aspect.event["init_for_consumers"] = {expression='callone(ao_oil.
           corba.services.event.EventChannel.for_consumers)',type='around'}
31     event_aspect.event["init_for_suppliers"] = {expression='callone(ao_oil.
           corba.services.event.EventChannel.for_suppliers)',type='around'}
32 end
```

Listagem 5.8: Definição do aspecto *EventService*

A definição do aspecto *EventService* é feita na linha 18. E a definição da interface *event* aparece nas linhas 19 a 26. Essa interface contém três pontos de junção abstratos. O primeiro ponto de junção abstrato, *lazy_load*, é o responsável por carregar o canal de eventos de CORBA, logo está associado a função *lazy_load*. O segundo e o terceiro pontos de junção abstratos são responsáveis por instanciar respectivamente os elementos *ConsumerAdmin* e o *SupplierAdmin* e estão associados as funções de mesmo nome apresentadas anteriormente. Na linha 28, é criado um o ponto de junção antecipado para monitorar a primeira chamada de qualquer elemento da API do serviço de eventos. Quando uma invocação a algum desses elementos acontece, é realizado a carga do canal de eventos, através da função *lazy_load*.

```
1  function invoke(self, reference, operation, ...)
2      local context = self.context
3      local requester = context.requester
4      local result, except = requester:getchannel(reference)
5      if result then
6          local channel = result
7          local mutex = context.mutex
8          mutex:locksend(channel)
9          result,except = self:do_invoke(requester,channel,reference,operation
               ,...)
10         mutex:freesend(channel)
11         if result then
12             result[InvokerKey] = self
13             result[ChannelKey] = channel
14             result = Request(result)
15         end
16     end
17     return result, except
18 end
19 function do_invoke(self,requester,channel,reference,operation,...)
20     local result, except = requester:newrequest(channel, reference,
           operation, ...)
21     return result,except
22 end
```

Listagem 5.9: Componente *Invoker* com suporte a concorrência do OiL.

5.4.3 Concorrência

O OiL oferece suporte à programação concorrente através do módulo *cooperative*. O módulo *cooperative* oferece um escalonador, utilizado para gerenciar a execução das requisições, e um mecanismo de sincronização baseado em *mutexes* para realizar a exclusão mútua. A análise da arquitetura concorrente do OiL constatou que o código responsável por implementar a programação concorrente estava entrelaçado e espalhado pelos demais componentes do sistema. Em especial, o código destinado a realizar a exclusão mútua. Sendo assim, trata-se de mais uma característica transversal do OiL.

A Listagem 5.9 ilustra o processo de entrelaçamento do código de sincronização em um componente de distribuição, o *Invoker*. Esse código tem por responsabilidade realizar uma invocação remota. Porém, as linhas 8 e 10 da Listagem 5.9 contém código que vai além da responsabilidade da classe.

As duas primeiras linhas da função *invoke* apenas criam referências para os atributos *context* e *context.requester*. O primeiro passo para realizar uma invocação remota é obter um canal de comunicação, esse passo é implementado pela função *getchannel* localizada na linha 4. Caso um canal de comunicação possa ser disponibilizado para a aplicação, o próximo passo é obter um *mutex* para controlar o acesso a região crítica. Na linha 7 é criada uma referência para a faceta *mutex*. Ela oferece acesso as primitivas de sincronização do OiL. A linhas 8 invoca a função *locksend* para solicitar o acesso a região crítica. Caso algum *thread* esteja nessa região crítica, a *thread* atual fica bloqueado esperando que a região crítica seja liberada. Após entrar na região crítica, a função *invoke* chama a função *do_invoker* para realizar a chamada remota. Uma vez concluída a chamada remota, *invoke* libera a região crítica através da função *freesend*, linha 10. Finalmente, o resultado é retornado a aplicação.

Para avaliar o grau de entrelaçamento desse conceito no código do OiL, foi utilizada a métrica CDLOC (*Concern Diffusion over LOC*). Essa métrica conta o número de pontos de transição para cada conceito do começo ao fim das linhas de código [Freitas, 2008]. O processo de coleta e análise dessa métrica foi realizado em parceria com Tássia Aparecida Vieira de Freitas, aluna de mestrado deste mesmo departamento. A Tabela 5.7 apresenta os resultados dessa análise.

Módulo	**CDLOC**
Kernel	16
Distribuição	12

Tabela 5.7: Análise do entrelaçamento do conceito de concorrência no OiL

A primeira linha da referida tabela apresenta o valor de CDLOC para os componentes do

kernel do OiL. Por sua vez, a segunda linha apresenta o valor de CDLOC para os componentes que implementam o conceito de distribuição no OiL. Esses resultados indicam que o conceito de concorrência está espalhado e entrelaçado com outros conceitos do OiL.

```
1  local icooperative = AspectInterface:new(
2      {name = 'thread'},
3      {
4          {refine = 'new', action = Cooperative.new },
5          {refine = 'init', action = Cooperative.init },
6          {refine = 'before_newrequest', action = Cooperative.
               before_newrequest },
7          {refine = 'after_newrequest', action = Cooperative.after_newrequest
               },
8          {refine = 'try_getreply', action = Cooperative.try_getreply},
9          {refine = 'do_process_all', action = Cooperative.do_process_all},
10         {refine = 'halt', action = Cooperative.halt},
11         {refine = 'pcall', action = Cooperative.pcall},
12     }
13 )
14 local cooperative_aspect =  Aspect:new('Cooperative')
15 cooperative_aspect:interface(icooperative)
16
17 function load_aspects()
18     cooperative_aspect.thread["new"]= {expression='callone(ao_oil.KERNEL.
           init)',type='after'}
19     cooperative_aspect.thread["init"]= {expression='call(ao_oil.KERNEL.init)
           ',type='after'}
20     cooperative_aspect.thread["before_newrequest"]= {expression='call(ao_oil
           .distribution.Invoker.do_invoke)',type='before'}
21     cooperative_aspect.thread["after_newrequest"]= {expression='call(ao_oil.
           distribution.Invoker.do_invoke)',type='after'}
22     cooperative_aspect.thread["try_getreply"]= {expression='call(ao_oil.
           distribution.Invoker.try_getreply)',type='around'}
23     cooperative_aspect.thread["do_process_all"]= {expression='call(ao_oil.
           distribution.Receiver.do_process_all)',type='around'}
24     cooperative_aspect.thread["halt"]= {expression='call(ao_oil.distribution
           .Receiver.halt_now)',type='around'}
25 end
```

Listagem 5.10: Definição do aspecto *Cooperative*

A implementação do conceito de concorrência como uma aspecto emerge como uma solução

natural para o problema citado. A modelagem do conceito de concorrência como aspecto é bastante direita: o aspecto deve interceptar as áreas que precisam ser protegidas pelos mecanismos de sincronização e injetar o código necessário.

O aspecto *Cooperative*, apresentado na Listagem 5.10, é formado por uma única interface, denominada *icooperative*. Essa interface define os pontos de atuação do aspecto. Note que essa interface intercepta elementos que fazem parte da infra-estrutura de distribuição mas não são específicos de nenhuma tecnologia. Logo, o aspecto *Cooperative* é independente da infra-estrutura de distribuição utilizada.

Os pontos de junção abstratos *new* e *init* são responsáveis por criar e inicializar os componentes necessários pelo módulo *cooperative*. O aspecto deve ser capaz de garantir a exclusão mútua nos canais de comunicação, logo ele deve atuar no envio e na recepção de requisições e respostas. Para garantir a exclusão mútua no envio de requisições, o aspecto define os ponto de junção abstratos: *before_newrequest* e *after_newrequest*. O primeiro intercepta as invocações de serviços antes delas ocorrem, o objetivo desse ponto de junção é obter um *mutex*, enquanto que o segundo atua após o envio de uma requisição para liberar o *mutex* obtido antes do envio da requisição. O ponto de junção abstrato *try_getrepley* garante a exclusão mútua durante o acesso ao canal para obter respostas de requisições realizadas.

É possível observar que os pontos de junção definidos até agora atuam no lado do cliente, ou seja, eles atuam no envio de requisições e na recepção de respostas . Entretanto, é necessário proteger a recepção das requisições e o envio de respostas, operações que ocorrem do lado do servidor. Para proteger esse grupo de operações, o aspecto *cooperative* define o ponto de junção *do_process_all*.

E por último, o ponto de junção abstrato *halt* realiza o processo de desalocação dos recursos empregados, como *threads*, *mutexes* e canais de comunicação.

5.4.4 Flexibilidade na Composição de Aspectos e Elementos Base no AO-OIL

Durante a implementação do AO-OiL, observou-se que existem vários cenários aonde o processo de composição dos aspectos com o código base da aplicação é sensível ao contexto. Nesses casos, os aspectos precisam ter acesso às informações contextuais para decidir se devem atuar ou não.

Como anteriormente citado na seção 4.5.1, diferentes tipos de *proxies* podem conviver em uma mesma aplicação. Logo, se faz necessário que os aspectos que implementam o conceito de distribuição sejam capaz de identificar os tipos de *proxies* que eles devem atuar. A Listagem 5.11

apresenta um exemplo onde ocorre tal situação exposta. As duas primeiras linhas criam dois aspectos abstratos:*corba_client* e *ludo_client*. Na linha 4, o aspecto *corba_client* é associado a interface *iclient*. De modo, equivalente, na linha 5 o aspecto *ludo_client* é associado a interface *iclient_ludo*. A instanciação dos aspectos ocorre nas linhas 7 e 8. É possível notar que ambos os aspectos implementam comportamentos diferentes, interfaces diferentes, porém atuam sobre um mesmo ponto de junção, que no caso é o método *Proxy* do componente *Client*. Todavia, como RE-AspectLua é usada como linguagem de configuração que direciona a atuação dos aspectos, a ambigüidade citada pode ser resolvida adicionando uma propriedade contextual a todo *proxy* criado que identifique qual aspecto de distribuição atua sobre o mesmo.

```
1 --Cria   o dos aspectos abstratos
2 local corba_client = Aspect:new("corba_client")
3 local ludo_client = Aspect:new("ludo_client")
4 corba_client:interface(iclient)
5 ludo_client:interface(iclient_ludo)
6 --Instancia   o dos aspectos
7 corba_client.icorba_client["proxy"] = {expression= 'call(ao_oil.kernel.
    Client.proxy)',  type = 'around'}
8 ludo_client.iludo_client["proxy"] = {expression= 'call(ao_oil.kernel.Client.
    proxy)',   type = 'around'}
```

Listagem 5.11: Pontos de junção coincidentes

A Listagem 5.12 mostra como definir em RE-AspectLua um conjunto de propriedades de contexto a cada elemento *proxy*. As propriedades de contexto são definidas via classe *Context* que atribui uma propriedade contextual a um elemento ou define uma propriedade de contexto global. Na linha 1, a propriedade "*distribution*" de valor "*corba*" é associada aos *proxies* do tipo *proxyA*. Similarmente, as linhas 2 e 3 definem novos valores para essa propriedade de acordo com o tipo de *Proxy* em questão. Os aspectos inspecionam o valor dessa propriedade e definem que protocolo atua sobre um dado *proxy* em questão.

```
1 Context:setProperty("proxyA","distribution", "corba)
2 Context:setProperty("proxyB","distribution", "Ludo")
3 Context:setProperty("proxyC","distribution",nil)
```

Listagem 5.12: Definindo propriedades contextuais

A Listagem 5.13 mostra como um adendo pode recuperar a propriedade contextual definida na Listagem 5.12. A primeira linha recupera o meta-objeto que representa o ponto de junção passado como parâmetro para a função *search_instance*, no caso *ao_oil.kernel.Client.proxy*, e acessa o campo *jp* que contém as informações contextuais desses ponto de junção. Em seguida,

na linha 2, recupera-se o valor da propriedade contextual *distribution*. O aspecto pode então decidir se ele deve atuar ou se for um aspecto de coordenação, como o da Figura 13(b), ele repassa a invocação para o componente encarregado de processá-la. As linhas de 3 a 8 da Listagem 5.13 mostram as expressões condicionais que inspeciona o valor da propriedade de contexto *distribution* e determinam qual protocolo de comunicação será usado.

```
1 context = LuaMOP:search_instance("ao_oil.kernel.client.proxy").jp.context
2 local distribution = context:getProperty("distribution")
3 if(distribution == "corba") then
4     corba:process(...)
5 else
6     if(distribution == "ludo")  then
7         ludo:process(...)
8     end
9 end
```

Listagem 5.13: Passagem de Informações contextuais

Essa técnica pode ser aplicada em vários cenários envolvendo diversos tipos de componentes que precisam ser sensíveis ao contexto. Adicionalmente, RE-AspectLua permite a criação de *scripts* de configuração de políticas podem ser definidos, através das propriedades de contexto globais, o que permite a habilitação os aspectos adequados ao contexto por instâncias do middleware.

5.5 Processo de desenvolvimento de aplicações

O desenvolvimento de aplicações com o AO-OiL requer que o desenvolvedor siga um conjunto de passos durante a construção de uma aplicação distribuída. O processo de desenvolvimento estabelece dois tipos de papeis para as aplicações: servidores e clientes. Evidentemente, que uma aplicação pode atuar tanto como servidor como cliente.

Os passos do processo de desenvolvimento de uma aplicação que atua como servidor são os seguintes:

1. Implementação dos componentes que atuarão como *servants*.
2. Instanciação e inicialização da arquitetura base do AO-OiL. Ou seja, invocação à função *ao_oil.init*.
3. Carregamento dos módulos necessários para a execução dos serviços.

4. Registro e instalação dos *servants* no AO-OiL.

```
1  local Server = { data = { a_number = 1234 } }
2
3  function Server:read(tag)
4      local value = Server.data[tag]
5      if value == nil then
6          error(orb:newexcept{"Control::AccessError",
7              tagname=tag,
8              reason="unknown tag name",
9          })
10     end
11     return value
12 end
13
14 function Server:write(tag, value)
15     local old = Server.data[tag]
16     if type(old) ~= type(value) then
17         error(orb:newexcept{"Control::AccessError",
18             tagname=tag,
19             reason="invalid value for tag",
20         })
21     end
22     Server.data[tag] = value
23 end
24
25 require "ao_oil"
26
27 ao_oil.main(function()
28     orb = ao_oil.init()
29     orb:load_module("Corba")
30
31     orb:loadidlfile("control.idl")
32     local server = orb:new_servant(Server, nil, "Control::Server")
33     ao_oil.writeto("ref.ior", orb:tostring(server))
34     orb:run()
35 end)
```

Listagem 5.14: Código de uma aplicação servidora no AO-OiL

A Listagem 5.14 ilustra como criar uma aplicação servidora no AO-OiL. As linhas de 1 a 24 definem o objeto servidor *Server*. Ele expõem os métodos *read* e *write* que respectivamente lê e

escreve uma *tag*, ambos disparam a exceção *AcesseError*. Logo essas linhas realizam o primeiro passo do processo. A linha 28 invoca a função *ao_oil.init* realizando, portanto, o segundo passo do procedimento definido acima. O terceiro passo é efetuado pela função *load_module* na linha 29 que carrega o módulo de suporte a CORBA. Em seguida, temos o carregamento das IDL's do objeto *Server*, requisitado pelo protocolo CORBA. Finalmente, o registro e a instalação do *servants*, quarto passo, é efetuado na linha 32. A variável *Server*, definida em tal linha, contém uma referência para o *servant* instalado. A linha 33 transforma a referência para o *servant* em uma referência textual e a grava em um arquivo. Por fim, é invocado o método *run* para dar inicio ao processo de espera por requisições.

Já os passos do processo de desenvolvimento de uma aplicação que atua como cliente são os seguintes:

1. Instanciação e inicialização da arquitetura base do AO-OiL. Ou seja, invocação à função *ao_oil.init*.

2. Carregamento dos módulos necessários para o funcionamento do cliente.

3. Instanciação de *proxies* para os *servants* que fornecem os recursos requisitados pelo cliente.

4. Realização de requisições sobre os *proxies*.

A Listagem 5.15 ilustra como criar uma aplicação cliente no AO-OiL. As linhas de 1 a 6 definem um procedimento para tratar as exceções que ocorrem nos *proxies*, a função *returnexception*. Essa função tem como parâmetros uma referência para o *proxy* que realiza a operação, a exceção levantada e o descritor da operação que levantou a exceção. Ela é registrada no AO-OiL como uma função para tratar exceções através da função *setexcatch* na linha 23.

A linha 9 é formada por uma invocação a função *ao_oil.init* como determina o primeiro passo do procedimento. Em seguida, na linha 10, o módulo Corba é carregado como determinar o segundo passo da aplicação. O terceiro passo ocorre na linha 13. Essa linha lê uma referência textual de um arquivo, através da função *ao_oil.readfrom* e a passa para a função *load* que gera um *proxy* a partir da referência textual passada. A partir desse momento, o cliente pode realizar requisições ao *servant* representado pelo *proxy*, como ocorre na linhas 14, 15, 25 e 29.

Vale salientar que o motivo pelo qual o código referente aos passos 3 e 4 estão encapsulados pela função *pcall*. Isso ocorre porque Lua determina que quando uma exceção precisa ser tratada, o programador dever utilizar a função *pcall* para encapsular o código que levanta a exceção.

```
1  local function returnexception(load, exception, operation)
2      if operation.name == "read" and
3          exception[1] == "IDL:Control/AccessError:1.0"
4      then return nil, exception.reason
5      end
6      error(exception)
7  end
8  ao_oil.main(function()
9      local orb = ao_oil.init()
10     orb:load_module("Corba")
11     local Server
12     local success, exception = ao_oil.pcall(function()
13         Server = orb:load(ao_oil.readfrom("ref.ior"))
14         print("Value of 'a_number' is ", Server:read("a_number")._anyval)
15         Server:write("a_number", "bad value")
16     end)
17     if not success then
18         if exception[1] == "IDL:Control/AccessError:1.0"
19             then print(string.format("Got error: %s '%s'", exception.reason,
                   exception.tagname))
20             else print("Got unkown exception:", exception[1])
21         end
22     end
23     orb:setexcatch(returnexception, "Control::Server")
24     local success, exception = ao_oil.pcall(function()
25         local value, errmsg = Server:read("unknown")
26         if value then print("Value of 'unknown' is ", value._anyval)
27         else print("Error on 'unknown' access:", errmsg)
28         end
29         Server:write("unknown", 1234)
30     end)
31     if not success then
32         if exception[1] == "IDL:Control/AccessError:1.0" then
33             print(string.format("Got error: %s '%s'", exception.reason,
                   exception.tagname))
34         else print("Got unkown exception:", exception[1])
35         end
36     end
37 end)
```

Listagem 5.15: Código de uma aplicação cliente no AO-OiL

Capítulo 6

Avaliação

O processo de avaliação pode ser classificado em dois tipos: análise interna e análise externa. A análise interna (Seção 6.1) busca mensurar o consumo de memória da arquitetura e o desempenho individual das operações básicas do AO-OiL em relação ao OiL. Além disso, também foi estudado o tempo de carga médio de alguns módulos do AO-OiL, bem como o tempo médio de carga e de inicialização de ambos os sistemas de middleware. O objetivo é quantificar o *overhead* que a refatoração do OiL introduziu e verificar o tempo gasto no processo de adaptação do middleware. Por outro lado, análise externa realiza uma comparação do desempenho do middleware no nível de aplicação. Para isso, foi desenvolvido um estudo de caso na área do petróleo e gás natural: um sistema de monitoramento remoto para poços petrolíferos com elevação artificial. A análise externa e seu estudo de caso são apresentados em detalhes na Seção 6.2.

6.1 Análise Interna

O processo de avaliação buscou mensurar o desempenho do AO-OiL nos seguintes pontos: (i) o consumo de memória; (ii) o desempenho das operações básicas do AO-OiL em relação ao OiL; (iii) o tempo médio de carga de alguns módulos do AO-OiL; (iv) o tempo médio de carga e de inicialização de ambos os sistemas de *middleware*. O objetivo é quantificar o *overhead* que a refatoração orientada a aspectos do OiL introduziu e verificar o tempo gasto no processo de adaptação do middleware. Para avaliar o *overhead* introduzido na refatoração do OiL foram analisados os desempenhos de duas operações comuns nas aplicações: a criação de novos *servants* e a requisição de serviços aos *servants*, invocações remotas.

Todos os testes foram realizados em um computador equipado com um processador Core 2 Duo com 4GB de RAM. O sistema operacional utilizado foi o Gentoo Linux com *kernel* 2.6.26.

A versão do interpretador Lua utilizado foi a 5.1.

A Figura 6.1 exibe o *overhead* introduzido pelo AO-OiL na criação de *servants*. Pelo gráfico é possível concluir que o *overhead* é pequeno já que para criar 300 *servants*, o AO-OiL demorou 16,369 microsegundos a mais do que o OiL, valor pouco significativo diante da latência da rede e de outras operações de entrada e saída.

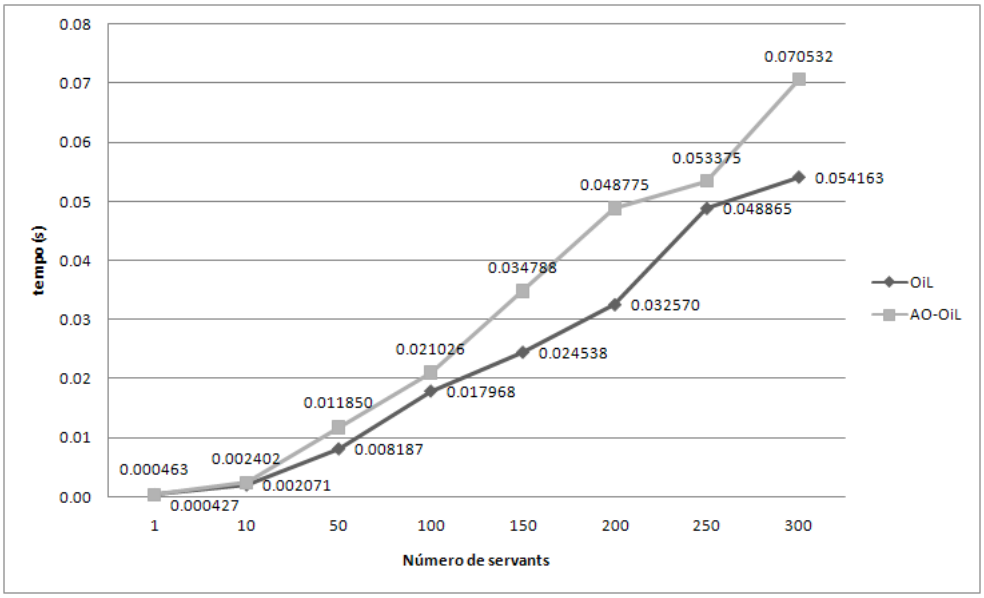


Figura 6.1: *Overhead* na criação de *servants*

A Figura 6.2 apresenta o resultado do teste comparativo que mede o desempenho dos sistemas de *middleware* na realização de uma invocação remota. Para que a latência da rede não influenciasse nos resultados, a medição foi feita localmente. Os dados dessa figura também indicam que o *overhead* introduzido pelo AO-OiL é aceitável.

Os *overheads* apresentados nas Figuras 6.1 e 6.2 ocorrem devido ao fato de que toda criação de novos *servants*, bem como toda invocação remota, são interceptadas pelos aspectos de distribuição do AO-OiL. Os aspectos definem um novo comportamento para essas operações de acordo com a plataforma de distribuição utilizada. Além disso, RE-AspectLua realiza a avaliação dos pontos de junção em tempo de execução, ou seja, toda interceptação feita por um aspecto provoca a avaliação da expressão de ponto de junção que o envolve. Essa característica de RE-AspectLua a torna mais custosa tanto em termos de desempenho quanto em termos de consumo de memória do que as abordagens estáticas. Todavia, pode-se afirmar que os custos são aceitáveis, pois o AO-OiL possui uma maior capacidade de personalização da infra-estrutura básica do que o OiL.

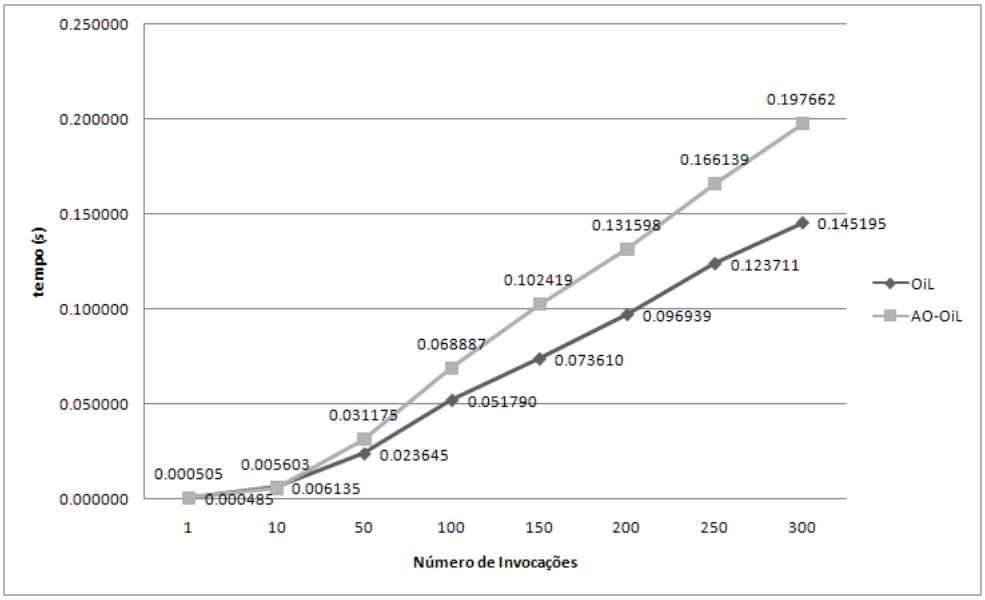


Figura 6.2: *Overhead* nas requisições de serviços

O consumo de memória foi avaliado através da comparação de três instâncias de ambos os sistemas de *middleware* com diferentes configurações e do consumo total de uma aplicação simples do tipo cliente-servidor com todos os módulos disponíveis carregados. As configurações usadas para cada middleware foram: (i) configuração padrão, (ii) configuração apenas com módulo distribuição e (iii) configuração com módulos de distribuição e concorrência. A Figura 6.3 exibe o consumo em *megabytes* dos cenários citados. Na primeira configuração (padrão) o AO-OiL carrega apenas o seu *microkernel*, ao passo que o OiL obrigatoriamente carrega o módulo de distribuição. Na segunda configuração, o AO-OiL carrega o módulo de distribuição, porém, o seu consumo continua ligeiramente inferior, pois o código do conceito de concorrência que estava entrelaçado dentro da arquitetura de distribuição do OiL foram retirados da versão aspectizada. Os demais serviços providos pela infra-estrutura de distribuição são carregados sob-demanda, ao passo que no OiL isso não acontece, sendo assim, o OiL carrega todos os serviços. Entretanto, o consumo do AO-OiL é maior quando todos os módulos estão carregados, como mostram os dois últimos cenários de tal figura. O consumo de memória do AO-OiL é maior nessa situação porque, além dos módulos, é carregado um conjunto de aspectos que atuam sobre esses módulos, o que eleva o consumo de memória. Em particular, o consumo de memória máximo em uma aplicação foi em torno de um *megabyte*. A maior parte desse consumo adicional de memoria advém da plataforma de orientação a aspectos utilizada. Vale a pena relembrar que o RE-AspectLua implementa a orientação a aspecto através de mecanismos reflexivos, o que implica na criação

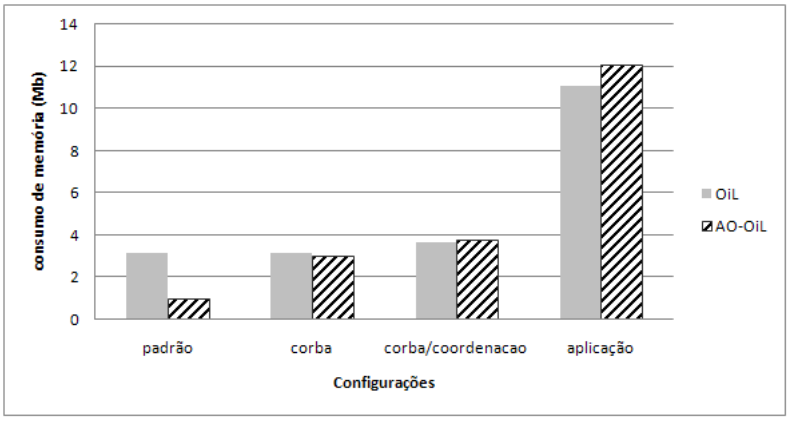


Figura 6.3: Consumo de memória

de ao menos um meta-objeto para cada elemento base afetado por um aspecto. Além disso, RE-AspectLua carrega um *parser* para avaliar as expressões de ponto de junção. Dessa maneira, um *megabyte* é um valor razoável, já que esse dado é soma da memória consumida pelo cliente e pelo servidor da aplicação.

A fim de avaliar o impacto da composição dinâmica em relação ao tempo de resposta da aplicação foi avaliado o tempo médio de carga de cada módulo. A Tabela 6.1 apresenta os resultados dessas medições. Com base nos resultados dessa tabela pode-se concluir que o tempo de carregamento dos módulos é insignificante, principalmente quando comparado a quantidade de memória poupada. Conclui-se que é mais vantajoso carregar e descarregar os módulos dinamicamente do que mantê-los em memória, já que dessa forma consegue-se diminuir a quantidade de memória sem comprometer o desempenho.

Módulo	**Tempo de carga** (ms)
Corba	61,357
Concorrência	0,69
Ludo	0,114

Tabela 6.1: Tempo de carga dos módulos

Finalmente, a Tabela 6.2 mostra o tempo necessário para carregar cada middleware. Como era de se esperar, o tempo de carga do AO-OiL é bem menor do que o tempo de carga do OiL, pois o AO-OiL carrega apenas o seu *microkernel*. Logo, pode-se afirmar que o AO-OiL tende a ter um tempo de resposta menor do que o do OiL.

Além do consumo de memória e do desempenho, é necessário verificar se a refatoração conduzida foi satisfatória. Em outras palavras, é necessário verificar se a introdução da dimensão

Middleware	**Tempo de carga** (ms)
OiL	94,5
AO-OiL	3,8

Tabela 6.2: Tempo de carga dos sistemas de middleware

de decomposição horizontal obtida com a separação dos conceitos transversais promovida pelo AO-OiL efetivamente melhora a modularidade do sistema. Em [Freitas, 2008] é apresentado um *framework* de métricas para a avaliação de sistemas de middleware orientadas a aspectos. Para avaliar a separação de conceitos, esse trabalho utilizou as seguintes métricas: CDC (*Concern Diffusion over Components*), CDO (*Concern Diffusion over Operations*) e CDLOC (*Concern Diffusion over LOC*). Os resultados dessas métricas podem ser vistos na Tabela 6.3.

Métricas	**OiL**	**AO-OiL**
CDC	26	27
CDO	242	265
CDLOC	77	27

Tabela 6.3: Métricas para separação de conceitos

As métricas CDC e CDO obtiveram resultados ligeiramente favoráveis ao OiL, explicitando que os interesses do AO-OiL estão implementados em um maior número de módulos e operações. Trata-se de um efeito decorrente da separação de conceitos promovida durante o processo de refatoração, o qual criou novas entidades e novas operações para implementar os interesses transversais do middleware. Em contrapartida, a métrica CDLOC apresentou uma diminuição significativa no AO-OiL, indicando que os conceitos transversais estão menos entrelaçados com o código base do middleware, evidenciando que os conceitos transversais estão melhor modularizados no AO-OiL e que há uma menor alternância entre código-base e código dos conceitos transversais dentro dos módulos. A melhor modularidade do AO-OiL facilita a sua manutenção, evolução, adaptação e extensão. Além disso, a refatoração tornou o código do AO-OiL mais reutilizável.

Os resultados obtidos na avaliação estiveram dentro do esperado e demonstraram que o emprego da OA na refatoração do OiL gerou uma arquitetura de middleware mais flexível sem impactar negativamente no desempenho do sistema gerado. O emprego conjunto das técnicas de decomposição vertical (componentização) e horizontal (aspectos) permitiu a criação de um núcleo mínimo contendo apenas as funcionalidades essenciais, aplicadas a qualquer sistema de middleware e, ao mesmo tempo, preparado para ser estendido de acordo com os requisitos da aplicação de uma forma simples e elegante. Portanto, pode-se afirmar que o uso da OA potenci-

alizou a capacidade de adaptação e personalização do OiL.

6.2 Análise Externa

A análise externa visa avaliar o impacto total da refatoração no desenvolvimento de aplicações distribuídas sobre o AO-OiL. Portanto, essa etapa da avaliação exige uma aplicação que utilize todos os recursos oferecidos pelo AO-OiL. Desse objetivo principal emergiu os requisitos para a aplicação que será utilizada como estudo de caso, logo a aplicação deve ser: distribuída, concorrente e orientada a eventos.

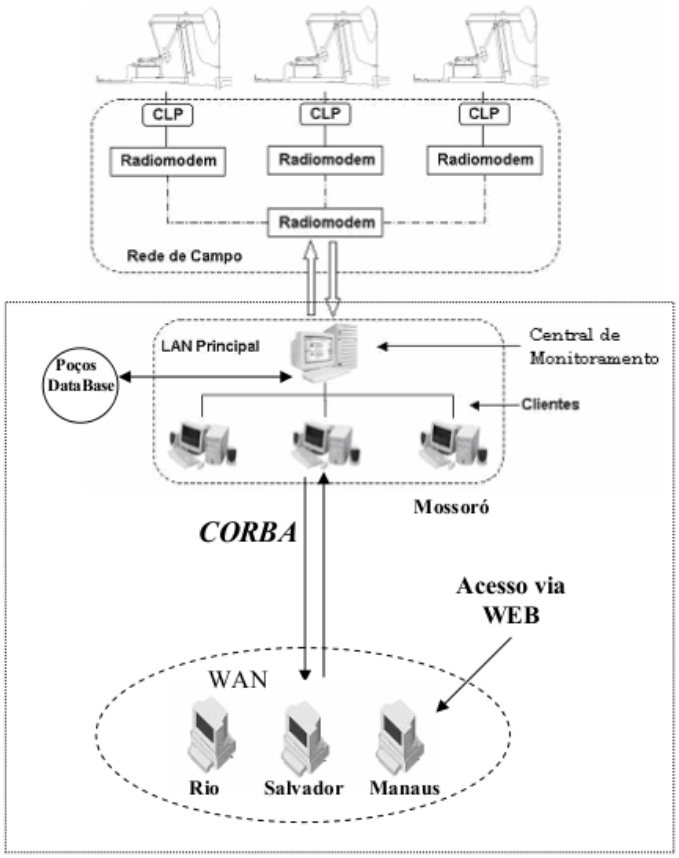


Figura 6.4: Estrutura do sistema de monitoramento

Esse trabalho utilizou como estudo de caso um sistema distribuído para monitoramento de poços com elevação artificial. Trata-se, portanto, de uma aplicação de tempo-real da indústria do petróleo e gás natural que é distribuída, concorrente e orientada a eventos. Essa aplicação

foi baseada no trabalho desenvolvido por Diogo Salim em [Salim, 2004] e foi desenvolvida em parceria com a Tássia Vieira de Freitas, aluna do Programa de Pós-graduação em Sistemas e Computação da UFRN.

O principal objetivo da aplicação desenvolvida nesse trabalho é monitorar as características individuais de um conjunto de reservatórios de petróleo, fornecer descrições estruturais e comportamentais dos poços monitorados, permitindo a realização de análises da produção de cada poço. Além disso, o sistema permite automatizar a tomada de decisões com base nos dados fornecidos pelo sistema.

A estrutura do sistema de monitoramento definido em [Salim, 2004] é apresentado na Figura 6.4. Através de uma avaliação dessa figura é possível identificar as partes que formam o modelo proposto: a rede de campos de petróleo, a central de monitoramento e os clientes. A cada poço está associado um Controlador Lógico Programável (CLP), que é responsável por enviar os dados relativos àquele poço. A central de monitoramento é um maquina que recebe, armazena e supervisiona os dados provenientes dos campos de petróleo. Finalmente, os clientes são estações remotas interessadas em serem notificadas sobre os eventos que ocorrem nos campos.

O gerenciamento da comunicação entre a central de monitoramento e os poços está fora do escopo do trabalho de [Salim, 2004]. Tal trabalho foca-se apenas na estruturação da comunicação entre a central de monitoramento e os clientes. Para implementar essa comunicação, foi utilizado uma arquitetura cliente-servidor. O servidor, instalado na mesma estação da central de monitoramento, é responsável por quatro tarefas primordiais: capturar as informações dos campos de petróleo, notificar os clientes interessados, disponibilizar um conjunto de serviços que permita a gerência dos campos petrolíferos e executar requisições oriundas dos clientes remotos. Por sua vez, os clientes apenas recebem notificações sobre os eventos ocorridos e com base nessas notificações podem tomar alguma ação através dos serviços disponibilizados pelo servidor.

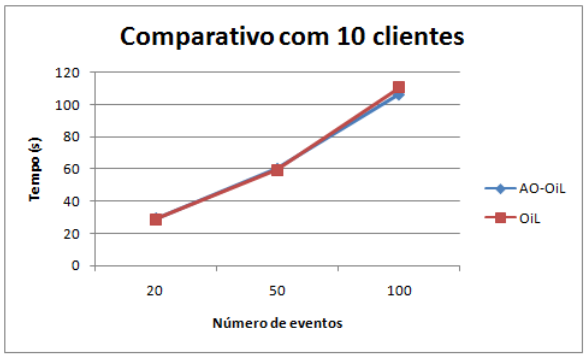
O processo de interação entre um cliente e o servidor é bastante simples. Primeiramente, o cliente deve buscar uma referência para o servidor no serviço de nomes. Em seguida, ele deve solicitar a sua inscrição no servidor para estar apto a ser notificado pelo servidor quando um evento ocorrer. Após a inscrição em um servidor, o cliente passa a receber os eventos que ocorrem nos poços e pode utilizar os serviços disponibilizados pela central de monitoramento caso necessite consultar ou até mesmo alterar algum parâmetro dos poços.

A aplicação foi desenvolvida em duas versões: uma versão para OiL e uma versão para o AO-OiL. Para mensurar o desempenho foram criados um conjunto de 12 cenários através da variação do número de clientes e do número de eventos gerados. A Figura 6.5 apresenta os resultados dos testes. Os resultados indicam que o *overhead* introduzido na aplicação foi mínimo, uma vez que

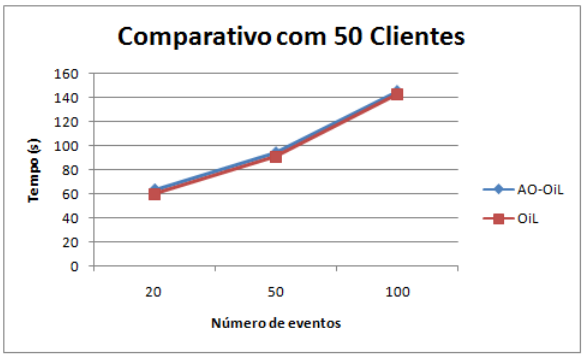
as curvas de desempenho são muito próximas. A ausência de penalidades significativas demostra que a programação orientada a aspectos pode ser utilizada sem necessariamente ter uma queda de desempenho na aplicação.

(a) Comparação com 1 cliente

(b) Comparação com 5 clientes



(c) Comparação com 10 clientes



(d) Comparação com 50 clientes

Figura 6.5: Teste de desempenho do sistema de monitoramento no AO-OiL e no OiL

Capítulo 7

Trabalhos Relacionados

Nesse capítulo é apresentado um resumo comparativo das principais abordagens de personalização dos sistemas de middleware encontradas na literatura em relação à abordagem proposta nesse trabalho. Pode-se dividir o conjunto de plataforma de middleware pesquisadas em dois: os sistemas de middleware orientados a aspectos, apresentados na seção 7.1 e os sistemas de middleware orientados a aspectos baseados na arquitetura de referência, descritos na seção 7.2

7.1 Sistemas de Middleware Orientados a Aspectos

7.1.1 Aspect-OpenORB

O Aspect-OpenORB [Cacho. et al., 2006] é um middleware OA, reflexivo, que consiste em uma refatoração do OpenORB [Blair et al., 1998]. Ele utiliza a programação orientada a aspectos para personalizar a infra-estrutura básica e a reflexão para personalizar a infra-estrutura da aplicação e a infra-estrutura de componentes.

Uma aplicação desenvolvida sobre o Aspect-OpenORB é composta por três elementos: o código base da aplicação, os aspectos relativos a configuração do ORB e os aspectos desenvolvidos para a própria aplicação. Estes elementos são combinados através do AspectLua.

No Aspect-OpenORB, todos os mecanismos internos do ORB são implementados como aspectos e o middleware inicia seu funcionamento com o aspecto que recebe uma invocação e, com base em tal invocação, seleciona o próximo aspecto a ser carregado dinamicamente para tratar a invocação.

A idéia do Aspect OpenORB é permitir que o código da aplicação defina as funcionalidades do middleware. Para que isso seja possível, o middleware deve verificar as dependências dos

métodos invocados a fim de carregá-las quando necessário. Para realizar esse processo, o Aspect-OpenORB utiliza os denominados *anticipated join points*. Eles são pontos de interceptação para elementos que ainda não foram declarados no programa, assim é possível interceptar métodos não declarados e aplicar ações a esses, tal como, uma carga preguiçosa.

Dessa maneira, o Aspect OpenORB suporta a personalização dinâmica incremental e decremental. Além disso, os testes de desempenho realizados por [Cacho. et al., 2006] indicam que o middleware não possui problema de desempenho.

Diferentemente do AO-OIL, a arquitetura desse middleware não segue a arquitetura de referência, portanto não há uma política clara para definição do *microkernel* e dos aspectos. O middleware define todos os mecanismos internos como aspectos o que pode se considerar um uso muito amplo de aspectos, sem respeitar a idéia da DSOA de que aspectos representam interesses transversais.

7.1.2 Lasagne

O Lasagne [Truyen et al., 2001] é um middleware orientado a aspectos para sistemas distribuídos que necessitam de personalização dinâmica sensível ao contexto. Essa sensibilidade ao contexto pode ser descrita como a capacidade de um cliente de personalizar seus serviços para uso no seu próprio contexto sem interferir no comportamento do serviço que é oferecido para os outros clientes. O Lasagne é atualmente desenvolvido pelo grupo *Distributed Systems and Computer Networks* (*DistriNet*) que é parte do departamento de Ciências da Computação da K.U. Leuven university.

O Lasagne possui como características principais a introdução dinâmica de aspectos em tempo de execução de maneira não intrusiva e a seleção sensível ao contexto dos aspectos que irão atuar no sistema. Ele define o seu próprio modelo de programação de componentes e de aspectos que não segue a nenhum dos padrões existentes. A modelagem de um sistema distribuído no Lasagne consiste de um núcleo funcional mínimo, implementado através do modelo de componentes, e de um conjunto de potenciais colaborações ou extensões, os aspectos, que podem ser seletivamente integradas com o núcleo funcional.

Um componente é uma unidade coerente de uma ou mais classes que possui um conjunto de interfaces providas e requeridas. Todas as interações entre componentes ocorrem unicamente através da troca de mensagens entre suas interfaces. O modelo de componentes do Lasagne suporta estes modelos de troca de mensagens: síncrono, assíncrono e invocação implícita.

Os aspectos são descritos através de colaborações (*collaborations*). Essas colaborações são constituídas por um conjunto de um ou mais *wrappers* ou *meta-wrappers*. Um componente

wrapper pode especializar um dado componente de núcleo ou outra colaboração através da adição de novos estados ou através da redefinição de métodos existentes em tal componente ou colaboração. Por sua vez, um componente *meta-wrapper* pode definir métodos especializados que serão reusados através de várias assinaturas de métodos ou em vários componentes. Além disso, ambos podem introduzir novas funcionalidades nos componentes.

No Lasagne, os *wrappers* operam no nível de instância, dessa maneira, um *wrapper* é a implementação por instância de um aspecto que é aplicada a cada componente. Isso permite a adaptação do comportamento do sistema em tempo de execução sem a necessidade de reiniciar o sistema.

Durante a instalação da aplicação, o descritor de componente especifica quais os *wrappers* devem ser aplicados a quais componentes de núcleo e a ordem de precedência entre os *wrappers*. O processo de *weaving* é feito em tempo de execução utilizando interceptadores que atuam sobre as mensagens que trafegam pelo sistema.

Entretanto, em comparação ao AO-OiL, esse middleware utiliza um modelo de programação próprio que não segue nenhum padrão existente e o suporte a criação aspectos reutilizáveis é limitado, pois os aspectos referenciam a elementos do código base da aplicação. O AO-OiL também oferece suporte a seleção sensível ao contexto dos aspectos através da camada intermediária de RE-AspectLua. O Lasagne utiliza uma abordagem dirigida a meta-dados para implementar composição aspectual sensível ao contexto, enquanto que o AO-OiL utiliza uma abordagem em uma linguagem de *script*. A abordagem utilizada pelo AO-OiL é mais abrangente pois além de permitir a criação de meta-dados, permite que o programador tenha maior poder de expressividade.

7.1.3 JBoss-AOP

O JBoss AOP é um *framework* orientado a aspecto programado em Java que pode ser utilizado em qualquer ambiente de programação ou integrado com o servidor de aplicações JBoss AS [Fleury and Reverbel, 2003]. Neste *framework*, todas as construções utilizadas na programação orientada a aspecto são definidas através de classes Java normais e associadas ao código da aplicação através de arquivos XML ou anotações. Ou seja, o JBoss AOP reutiliza o modelo de componentes da plataforma Java e define apenas o seu próprio modelo de programação de aspectos.

O modelo de aspectos do JBoss AOP oferece ao programador dois tipos distintos de aspectos: o aspecto e o interceptador. O primeiro codifica uma categoria de aspecto que possui um número arbitrário de *advices*, enquanto que os interceptadores são aspectos que possuem um

único *advice*.

O modelo de pontos de junção do JBoss AOP é constituído principalmente por chamadas a métodos e construtores, execução de métodos e de construtores e acesso a campos (leitura e escrita). O modelo de ponto de junção definido pelo JBoss AOP é intrusivo. Isso significa que o *framework* pode interceptar pontos da execução do programa independentemente do fato deles serem públicos, privados ou protegidos. Os pontos de junção são definidos através de expressões regulares que podem utilizar expressões booleanas e curingas.

Além dos aspectos, interceptadores e os *pointcuts*, o JBoss AOP possui estas duas importantes estruturas: *Introduction* e *Mixin*. A primeira construção permite a introdução de uma interface em uma classe Java existente em tempo de carga. E a segunda construção é utilizada quando interface introduzida adiciona novos métodos a uma classe Java existente. Neste caso, tal classe deve implementar tais métodos para que ela continue sendo uma classe Java válida. O *framework* JBoss AOP permite que se defina uma classe Java que implementa essa interface através da construção *mixin*.

O JBoss AOP suporta a programação orientada a aspectos dinâmica através dos seguintes mecanismos: *Hot Deployment*, *Hot Swap* e *Per Instance AOP*. O mecanismo de *Hot Deployment* permite adicionar e remover aspectos em tempo de carga em pontos instrumentados. O *Hot Swap* realiza a troca de *bytecodes* em tempo de execução das classes afetas por operações de POA dinâmica em pontos previamente não instrumentados. Finalmente, O mecanismo *Per Instance AOP* permite aplicar interceptadores em instâncias específicas de um objeto de uma dada classe, ao invés de aplicar em todo objeto da classe em questão. Esse mecanismo é muito útil quando instâncias de um objeto precisam ter comportamentos diferentes em algumas circunstâncias.

O JBoss AOP promove o reúso de aspectos através da separação dos pontos de junção do corpo do aspecto. Porém, diferentemente da presente proposta, o JBoss AOP não suporta a aspectização da infra-estrutura básica, não segue a arquitetura de referência e não possui a flexibilidade de ter uma camada intermediária que liga aspectos a elementos base.

7.2 Sistemas de Middleware Orientados a Aspectos baseados na Arquitetura de Referência

7.2.1 OpenCOM AOP

O OpenCOM AOP é a versão orientada a aspectos do OpenCOM [Coulson et al., 2002b] [Coulson et al., 2002a] que permite o desenvolvimento de serviços do middleware baseado em

componentes e em aspectos, tornando mais flexível o processo de configuração e reconfiguração dinâmica [Greenwood et al., 2007].

O OpenCOM é um middleware baseado em componentes leve, reflexível e eficiente, construído sobre a plataforma COM da Microsoft. Ele adota as seguintes características do COM: i) interoperabilidade binária, ii) a linguagem de definição de interface - IDL, iii) identificador global único, e iv) a interface *IUnknown*, útil para contagem de referência e descoberta. Ele é construído utilizando o padrão de projeto *microkernel*. Todo elemento do OpenCOM é um componente descrito através de suas interfaces e de seus receptáculos. As conexões entre interfaces e receptáculos são explicitamente estabelecidas e são representadas pelos elementos *Connections*.

A introspecção e a adaptação são providas através de meta-interfaces que todo componente pertencente ao middleware deve implementar. Além disso, o OpenCOM mantém uma grafo das estruturas dos componentes e gerencia o ciclo de vida dos mesmos.

O OpenCOM AOP segue a mesma abordagem do OpenCOM para o modelo de componentes. Os aspectos são definidos sobre o topo dos meta-níveis do OpenCOM. O nível aspectual pode ser carregado e descarregado dinamicamente assim como os outros meta-níveis. A Figura 7.1 mostra os meta modelos do OpenCOM AOP.

O nível aspectual é composto por um *Aspect Manager* que a partir de um ponto de junção e a especificação do aspecto fornece os mecanismos necessários para se comunicar com os meta-níveis necessários para que o middleware reflita o comportamento descrito pelo aspecto.

A introdução dos aspectos na arquitetura implica a necessidade de mecanismos para composição aspectual, definição de um conjunto de pontos de junção a serem interceptados pelos aspectos e uma linguagem de definição de pontos de junção.

O conjunto de pontos de junção do OpenCOM AOP é formado pelas interfaces e operações implementadas por cada interface. Por sua vez, a linguagem de ponto de junção do middleware é baseada na APL da arquitetura de referência e admite dois tipos de propriedades: estática e dinâmica. As propriedades estáticas são definidas através de expressões regulares que atuam sobre componentes e interfaces. As operações podem ser referenciadas através de suas assinaturas. Enquanto que, as propriedades dinâmicas são expressas através de pares <chave,valor> anexadas a componentes e a interfaces.

A principal diferença entre o OpenCOM AOP e o AO-OiL é fato de que o OpenCOM AOP não é construído utilizando aspectos. Ele apenas fornece um *framework* para o desenvolvimento de aplicações orientadas a aspectos baseadas em componentes. Ou seja, o OpenCOM AOP não utiliza aspecto na construção da sua infra-estrutura e dos serviços oferecidos pelo middleware.

A principal desvantagem dessa abordagem OpenCOM AOP é que herda os problemas do

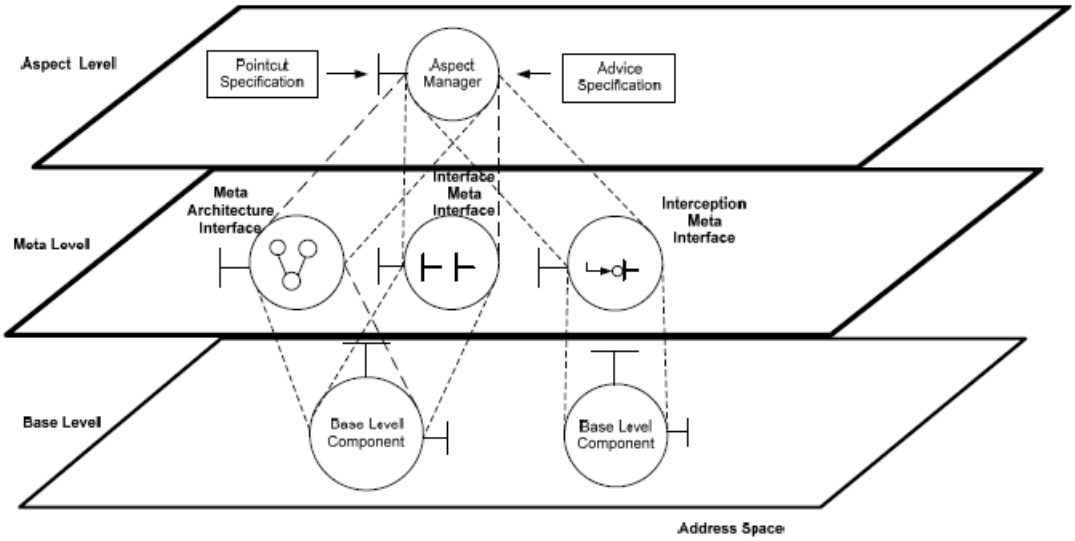


Figura 7.1: OpenCOM AOP

OpenCOM. O OpenCOM exige um maior esforço para personalizar o middleware em comparação ao AO-OiL. É difícil desenvolver novas funcionalidades como fábricas de *bindings*, protocolos de distribuição dentre outros serviços. Além disso, o processo de modificar as funcionalidades existentes é muito suscetível a erros, uma vez que os meta-modelos disponibilizam primitivas de baixo para modificar a infra-estrutura. No AO-OiL, o desenvolvedor utiliza-se da programação orientada a aspectos para adicionar, modificar e substituir novas funcionalidades, logo ele não fica totalmente dependente das primitivas oferecidas pelo middleware.

O OpenCOM AOP ainda não tem uma implementação disponível. Todas as informações sobre a sua arquitetura foram retiradas de artigos que limitam-se a trazer uma especificação da arquitetura.

Capítulo 8

Conclusão

Este trabalho apresentou o AO-OiL, uma plataforma de middleware orientada a aspectos dinamicamente adaptável baseada na arquitetura de referência para middlewares OA definida por Loughran et al [Loughran et al., 2005]. A utilização da programação orientada a aspectos na construção de plataformas de middleware permite a criação de middlewares com núcleo mínimo cujas funcionalidades adicionais são ditadas pelo código da aplicação, o que permite ao usuário do middleware adequá-lo às suas necessidades. Portanto, a arquitetura base de um middleware orientado a aspectos tende a ser formada por funcionalidades básicas como instanciação e ativação de componentes, carga de componentes em contêineres, e estabelecimento de associações entre componentes. As demais funcionalidades, como suporte a programação concorrente, mecanismos de coordenação, protocolos de distribuição, dentre outros conceitos transversais envolvidos na construção de plataformas de middleware, são inseridas segundo os requisitos da aplicação, tratando-se, portanto, de um processo de adaptação incremental do middleware. O alto grau de personalização da arquitetura proposta advém da combinação do AO-OiL, um middleware dinâmico, com RE-AspectLua, uma linguagem OA dinâmica.

Muitos trabalhos utilizam o paradigma orientado a aspectos no desenvolvimento de plataforma de middlewares, contudo cada projeto utiliza um conjunto de soluções específicas para tratar um mesmo problema: diminuir a complexidade dos middlewares. Evidentemente, que comparações entre as várias tecnologias de middleware é difícil nesse cenário, pois não havia uma arquitetura de referência que servisse como modelo base para o desenvolvimento de middleware OA.

Por esse motivo, esse trabalho aplicou as técnicas da programação orientada a aspectos aliadas aos conceitos definidos na arquitetura de referência. O objetivo foi validar a arquitetura de referência dado que, por ser uma proposta recente, não há relato na literatura de middlewares

construídos com base em tal arquitetura. Isso pode ser considerado um passo em direção ao objetivo final definido pelos autores da arquitetura de referência: formular um modelo de referência que reúna as melhores práticas para a construção de middlewares orientados a aspectos.

A implementação do AO-OiL revelou que a arquitetura de referência explora demasiadamente da idéia de aspecto. Em particular, a classificação do conceito de distribuição como um conceito transversal é bastante discutível, pois embora seja um conceito complexo, trata-se de uma funcionalidade básica das plataformas de middleware e não um conceito transversal. Além disso, a distribuição é um conceito bem modularizado pelos paradigmas convencionais de desenvolvimento. Portanto, os mesmos benefícios, ou praticamente os mesmos, alcançados através da orientação a aspectos, podem ser conseguidos através da programação baseada em componentes. O contrário ocorre com o conceito de concorrência, é possível notar claramente que esse conceito não é adequadamente modularizado pelas abordagens tradicionais.

Os resultados obtidos com o desenvolvimento desse trabalho indicam que a utilização mais restrita dos aspectos pode ser uma abordagem mais adequada. A arquitetura de referência propõe o uso de aspectos para modularizar distribuição, coordenação, mobilidade e outros conceitos, no entanto, o uso mais restrito, como por exemplo, a modulaziração via aspectos apenas da coordenação e da concorrência pode gerar resultados mais satisfatórios. Uma vez que, evita-se o *overhead* introduzido pela aplicação dos aspectos nesse conceito, como indicou os testes de desempenho da Seção 6, e mantém a boa modularização do código do middleware. Essa nova configuração gera um middleware mais eficiente porque não é mais necessário a interceptação de toda requisição de serviço feita pelo cliente. Outra saída, seria utilizar um *proxy colocated* via aspecto. Nesse esquema, o aspecto descobriria se a requisição seria remota ou não e criaria o *proxy* de acordo com o contexto.

Dessa maneira, a ampla utilização dos aspectos pode não trazer, na prática, benefícios significativos que justifiquem a sua aplicação. Por conseguinte, pode-se afirmar que uma análise mais aprofundada sobre a melhor maneira de implementar cada conceito se faz necessária.

Em termos de avaliação, foram realizados vários experimentos que buscaram avaliar o desempenho do middleware proposto em relação ao OiL. Esses experimentos avaliaram o consumo de memória da arquitetura, o desempenho individual das principais operações, o tempo médio de carga e de inicialização de ambos os middlewares. Completando o ciclo de análises, foi realizado um estudo do desempenho do AO-OiL em nível de aplicação. Para isso, utilizou-se uma aplicação do setor de petróleo e gás natural, mais precisamente um software destinado a monitoração remota de poços de petróleo.

Desses experimentos, conclui-se que o alto grau de personalização do AO-OiL permitiu eco-

nomizar uma quantidade significativa de memória sem comprometer o desempenho das aplicações, pois os testes de desempenho mostraram que o *overhead* introduzido é pequeno. Além disso, a melhor modularização da arquitetura proposta torna mais fácil a adaptação dinâmica do middleware, a evolução e a extensão da plataforma.

8.1 Contribuições

Resumidamente, as contribuições desse trabalho foram:

- Definição de uma arquitetura de middleware OA extensível, baseada na arquitetura de referência para middleware AO, com suporte a adaptação dinâmica.
- Extensão da linguagem de orientação a aspectos RE-AspectLua para expressar as expressões de pontos de corte definidas na arquitetura de referência.
- Implementação dessa arquitetura em Lua e RE-AspectLua.
- Validação da arquitetura de referência dado que, por ser uma proposta recente, não há relato na literatura de middleware AO construído com base em tal arquitetura.

8.2 Trabalhos Futuros

Alguns trabalhos futuros podem ser derivados do presente trabalho.

O primeiro e mais evidente é a introdução de outros conceitos transversais. Uma gama maior de conceitos transversais permitirá a utilização do middleware proposto em vários domínios de aplicação, além de permitir um análise da escalabilidade da plataforma. Outra linha de pesquisa nessa direção é definição de cenários de evolução para analisar a capacidade de personalização da plataforma.

Além desses estudos, pode-se citar uma linha de pesquisa correlata que é a implementação de políticas de adaptação. As políticas de adaptação fornecem aos usuários do AO-OiL uma maior facilidade na escolha de configurações adequadas ao contexto e domínio de suas aplicações.

Outro trabalho interessante é aplicar o conceito de linha de produto de software (SPL) ao AO-OiL, para criar um middleware com um núcleo mínimo (*microkernel*) e serviços opcionais representando *features* especificas. Isso permitiria o desenvolvimento de famílias de middleware, especialmente interessante para sistemas embarcados ou para a computação ubíqua, pois esses

sistemas têm muitas especificidades e exigem uma plataforma de middleware com as características do AO-OiL, ou seja, um middleware altamente personalizável, flexível e configurável.

A melhor forma de expressar a concorrência, é uma interessante questão não tratada no escopo desse trabalho. Sendo assim, um profundo estudo comparativo entre as abordagens orientada a aspectos e orientada a objetos tem grande valia para a comunidade acadêmica. Nessa mesma linha, um estudo comparativo entre o AO-OiL e outros middlewares baseados em *microkernel* também teria grande valia para a comunidade, pois forneceria maiores subsídio para analisar a arquitetura e a API do *microkernel* proposto.

O desenvolvimento de outras aplicações que demandem vários conceitos transversais é interessante para validar o middleware de forma mais ampla e verificar a sua capacidade de adaptação. Enriquecendo o conjunto de estudados de casos da arquitetura proposta.

O aumento do número de conceitos transversais logicamente implica em um aumento do número de pontos de junção definidos por RE-AspectLua. Esse contexto leva a seguinte questão: RE-AspectLua é escalável? Naturalmente, podemos citar a analise de escalabilidade de RE-AspectLua como um trabalho futuro.

Finalmente, uma análise de desempenho mais precisa é bastante interessante, pois o cruzamento entre esses dados e os dados apresentados nesse trabalho podem mensurar a influência de processos adjacentes como o coletor de lixo de Lua.

Referências Bibliográficas

[Arbab, 2005] Arbab, F. (2005). Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52.

[Batista and Vieira, 2007] Batista, T. and Vieira, M. (2007). RE-AspectLua-Achieving Reuse in AspectLua. *Journal of Universal Computer Science*, 13(6):786–805.

[Bernstein, 1996] Bernstein, P. (1996). Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98.

[Blair et al., 1998] 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.

[Cacho et al., 2005] Cacho, N., Batista, T., and Fernandes, F. (2005). AspectLua-A Dynamic AOP Approach. *Journal of Universal Computer Society (J. UCS)*, 11(7):1177–1197.

[Cacho. et al., 2006] Cacho., N., Batista, T., Garcia, A., Sant'Anna, C., and Blair, G. (2006). Improving modularity of reflective middleware with aspect-oriented programming. *Proceedings of the 6th international workshop on Software engineering and middleware*, pages 31–38.

[Chavez et al., 2006] Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., and Lucena, C. (2006). Taming Heterogeneous Aspects with Crosscutting Interfaces. *Journal of the Brazilian Computer Society*, 12(1).

[CORBA, 1999] CORBA, O. (1999). The Common Object Request Broker: Architecture and Specification. *Object Management Group disponível em < http://www. corba. org/>. Acesso em: 12/05/2008*.

[Coulson et al., 2002a] Coulson, G., Blair, G., Clarke, M., and Parlavantzas, N. (2002a). The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126.

[Coulson et al., 2002b] Coulson, G., Blair, G. S., Clarke, M., and Parlavantzas, N. (2002b). The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126.

[Cunha et al., 2006] Cunha, C. A., Sobral, Jo a. L., and Monteiro, M. P. (2006). Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 134–145, New York, NY, USA. ACM.

[Dósea et al., 2007] Dósea, M., Neto, A. C., Borba, P., and Soares, S. (2007). Specifying Design Rules in Aspect-Oriented Systems. In *I Latin American Workshop on Aspect-Oriented Software Development - LA-WASP'2007, affiliated with SBES'07*, pages 67–78, João Pessoa-PB,Brazil.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

[Fernandes et al., 2004] Fernandes, F., Cacho, N., and Batista, T. (2004). Luamop – a meta-object protocol for dynamic weaving. In *First Brazilian Workshop on Aspect-Oriented Software Development (WASP'04), 18 th Brazilian Symposium on Software Engineering (SBES)*.

[Fleury and Reverbel, 2003] Fleury, M. and Reverbel, F. (2003). The jboss extensible server. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 344–373, New York, NY, USA. Springer-Verlag New York, Inc.

[Freitas, 2008] Freitas, T. A. V. (2008). Métricas para avaliação de sistemas de middleware orientado a aspectos e aplicação em um sistema de monitoramento de poços de petróleo. Tese de mestrado, Universidade Federal do Rio Grande do Norte.

[Greenwood et al., 2007] Greenwood, P., Loughran, N., Surajbali, B., Coulson, G., Seinturier, A. R. L., andRenaud Pawlak, N. P., Truyen, E., Sanen, F., Lagaisse, B., Gregoire, J., Bynens, M., Jackson, W. J. A., Clarke, S., Hatton, N., Pinto, M., Fuentes, L., andTal Cohen, M. A., Colyer, A., and Schwanninger, C. (2007). Validation of the reference architecture. Technical Report AOSD-Europe Deliverable D68, AOSD-Europe-ULANC-26, Lancaster University.

[Greenwood et al., 2008] Greenwood, P., Surajbali, B., Coulson, G., and Rashid, A. (2008). Reference architecture v3.0. Technical Report AOSD-Europe Deliverable D103, AOSD-Europe-ULANC-37, Lancaster University.

[Henning and Vinoski, 1999] Henning, M. and Vinoski, S. (1999). *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Ierusalimschy et al., 2007] Ierusalimschy, R., de Figueiredo, L., and Celes, W. (2007). The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1. ACM New York, NY, USA.

[Jacobsen, 2001] Jacobsen, H. (2001). Middleware architecture design based on aspects, the open implementation metaphor and modularity. *Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, UK, August*.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York.

[Kon et al., 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. (2002). The Case for Reflexive Middleware. *Communications of the ACM*, 45(6):33–38.

[Loughran et al., 2005] Loughran, N., Coulson, G., Seinturier, L., Pawlak, R., Truyen, E., Sanen, F., Bynens, M., Joosen, W., Jackson, A., Clarke, S., Hatton, N., Pinto, M., Fuentes, L., Amor, M., Cohen, T., Colyer, A., and Schwanninger, C. (2005). Requirements and definition of aspect-oriented middleware reference architecture. Technical Report AOSD-Europe Deliverable D21, AOSD-Europe-ULANC-15, Lancaster University.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155.

[Maia, 2004] Maia, R. (2004). Lua object-oriented programming. Disponível em <http://loop.luaforge.net>. Acesso em: 02/03/2008.

[Maia et al., 2006] Maia, R., Cerqueira, R., and Cosme, R. (2006). OiL: An Object Request Broker in The Lua Language. In *Proc. 5th Tools Session of the Brazilian Simposium on Computer Networks (SBRC2006)*, Curitiba, Brazil.

[Maia et al., 2005] Maia, R., Cerqueira, R., and Kon, F. (2005). A middleware for experimentation on dynamic adaptation. *Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, pages 1–6.

[Nehab, 2004] Nehab, D. (2004). "Luasocket: Network support for the lua language.". Object Management Group disponível em < http://www.tecgraf.puc-rio.br/ diego/professional/luasocket/>. Acesso em: 12/05/2008.

[Putrycz and Bernard, 2002] Putrycz, E. and Bernard, G. (2002). Using aspect oriented programming to build a portable load balancing service. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 473–478.

[Rashid and Chitchyan, 2003] Rashid, A. and Chitchyan, R. (2003). Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA. ACM.

[Salim, 2004] Salim, D. C. (2004). Um sistema distribuído para monitoramento de poços de petróleo com elevação artificial. Master's thesis, Universidade Federal do Rio Grande do Norte.

[Schmidt et al., 2000] Schmidt, D., Rohnert, H., Stal, M., and Schultz, D. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc. New York, NY, USA.

[Stankovic and Ramamritham, 1994] Stankovic, J. and Ramamritham, K. (1994). A reflective architecture for real-time operating systems. J.A. Stankovic and K. Ramamritham, A reflective architecture for real-time operating systems, in Advances in Real-Time Systems, Sang Son, Ed. Prentice-Hall, 1994.

[Truyen et al., 2001] Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., and Joergensen, B. (2001). A Dynamic Customization Model for Distributed Component-Based Applications. *Proceedings of DDMA*, 2001:147–152.

[Van Deursen et al., 2000] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo