2
Trabalhos Relacionados

A possibilidade de efetuar adaptações de forma dinâmica em sistemas de software tem sido discutida sob diversas perspectivas em trabalhos recentes. Em [26], uma série de trabalhos são analisados no intuito de identificar a tecnologias que dão suporte à adaptação dinâmica, classificar as diversas abordagens e identificar os principais desafios que ainda precisam ser endereçados para que a adaptação dinâmica alcance seu potencial integral. Três dimensões são avaliadas na classificação das abordagens. A primeira é *como* a adaptação é implementada, isto é, quais mecanismos específicos são utilizados para permitir a adaptação composicional. Uma série de técnicas são citadas, entre elas o uso dos padrões *proxy* e *wrapper*, o protocolo de meta-objetos e a programação orientada a aspectos. O segundo critério de avaliação é *quando* as adaptações são aplicadas. Nesse quesito, as abordagens são classificadas como dinâmicas, caso ocorram em tempo de execução, ou estáticas, caso ocorram antes do tempo de execução. As adaptações estáticas são subdivididas em *hardwired*, caso ocorram em tempo de desenvolvimento, *customizáveis* (*customizable*), caso ocorram em tempo de compilação ou de linkedição, ou *configuráveis* (*configurable*), caso ocorram em tempo de carga. Já as adaptações dinâmicas são subdivididas em *ajustáveis* (*tunable*), caso não envolvam alterações no código funcional, ou *mutáveis* (*mutable*), caso contrário. O último critério de avaliação é *aonde* o código adaptativo é inserido. Nesse critério, as possibilidades são a camada de sistema operacional, a camada de infra-estrutura do *middleware*, as camadas de distribuição (comunicação distribuída) e de serviços do *middleware* e a própria camada da aplicação. Na camada de infra-estrutura foram identificadas soluções que criaram uma camada de comunicação adaptável e outras que estenderam a máquina virtual para interceptar e redirecionar as interações no código funcional. Na camada de distribuição, a abordagem típica é a interceptação e processamento de mensagens remotas. No nível da aplicação, duas técnicas foram identificadas: o uso de linguagens com suporte inerente à reconfiguração e a mesclagem de código adaptativo e funcional no tempo de

compilação ou de carga.

Dentre as diversas abordagens existentes para a questão de adaptação dinâmica, selecionamos as de maior importância no contexto do trabalho desenvolvido para um estudo aprofundado. Os sistemas *DynamicTAO* e *OpenCOM* foram escolhidos por serem sistemas pioneiros na adaptação dinâmica de aplicações baseadas em componentes de software. Eles permitem apenas adaptação através da reconexão de componentes e o código adaptativo se situa na camada de distribuição do *middleware*. O sistema de *middleware Comet* foi selecionado por estender a capacidade de adaptação, permitindo alterações no comportamento interno do componente, e por introduzir abstrações que permitem controlar alterações de comportamento dos componentes de forma estruturada. Já o TRAP/J é uma ferramenta que embora implementada em Java, apresenta uma proposta radicalmente diferente da adotada neste trabalho. Ele suporta adaptações de ponto pequeno diretamente no nível da aplicação, introduzindo código adaptativo junto ao código funcional das aplicações em tempo de compilação. Por fim, analisamos o *LuaCCM* [23], um sistema desenvolvido em nosso grupo de estudos que mereceu atenção especial pois serviu de referência para a implementação descrita nesta dissertação.

2.1 DynamicTAO

Em [27], é apresentado o *DynamicTAO*, um *middleware* reflexivo criado para suportar o sistema operacional distribuído 2K. A motivação por trás do desenvolvimento do 2K, e, consequentemente do *DynamicTAO*, foi a identificação da necessidade crescente de sistemas modernos de se adaptar a variações na disponibilidade de recursos em ambientes de execução distribuídos. Enquanto sistemas de *middleware* tradicionais são otimizados para arquiteturas e configurações específicas, esses sistemas oferecem a possibilidade de ter sua configuração adaptada dinamicamente sempre que uma mudança no ambiente for identificada.

O *DynamicTAO* é uma extensão de um ORB convencional, o *TAO ORB* [28], que permite a reconfiguração dinâmica do estado interno do ORB e das aplicações rodando sobre ele. Ele armazena as dependências entre os componentes do ORB e da aplicação em entidades chamadas de *ComponentConfigurators*, de modo a gerar um grafo de dependências direcionado. Quando é necessário substituir algum componente, o *middleware* analisa as dependências

do componente substituído utilizando o *ComponentConfigurator* associado a ele para garantir a consistência do estado interno do ORB. Caso seja necessário, os *ComponentConfigurators* podem ser estendidos pelos desenvolvedores de componentes para oferecer tratamento adequado a diferentes tipos de componentes.

As funcionalidades de adição e remoção de módulos ao sistema e inspeção e alteração do estado de configuração do ORB no *DynamicTAO* são oferecidos sob a forma de meta-interfaces, que atendem a: (1) desenvolvedores, para a realização de testes e debug, (2) administradores de sistemas, para realização de tarefas de manutenção e (3) componentes de software, que inspecionam e reconfiguram o sistema baseados em informações colhidas de fontes diversas. O *DynamicTAO* oferece ainda suporte ao uso de interceptadores para a interposição de código entre chamadas remotas. Esse recurso é baseado no padrão definido pela OMG [29] para a implementação de interceptadores portáteis.

2.2 OpenCOM

Outro trabalho bastante relevante no contexto desta dissertação é o *OpenCOM* ([30] e [31]), uma tecnologia de construção de sistemas de software baseados em componentes independente de linguagem de programação que tem como requisitos o suporte à reconfiguração dinâmica em tempo de execução, a aplicabilidade em um conjunto abrangente de ambientes de execução e um impacto mínimo sobre o desempenho das aplicações. Para atender esses requisitos, o *OpenCOM* define um modelo de componentes de tempo de execução (*run-time component model*) genérico como base e introduz as noções arquiteturais de arcabouços de componentes (*component frameworks* - CFs) e *meta-modelos reflexivos*.

O modelo de componentes do *OpenCOM* foi construído tendo como base um subconjunto das facilidades do modelo COM [32], da Microsoft. Foram ignoradas todas as características de alto nível de COM, tais como distribuição, persistência, segurança, transações e reflexão, e aproveitadas as seguintes facilidades: (1) o padrão básico de interoperabilidade em nível binário (i.e. a estrutura de dados *vtable*), (2) a linguagem de definição de interfaces MS IDL, (3) os identificadores globais únicos (GUIDs), e (4) a interface de descoberta e contagem de referências *IUnknow*.

Os conceitos fundamentais do modelo de programação do *OpenCOM* são as *interfaces*, *receptáculos* e *conexões*. Enquanto uma *interface* representa uma unidade de provisão de serviço, um *receptáculo* representa um unidade de requisição de serviço, e é usada para expressar de forma explícita a dependência de um componente por uma interface externa. Uma *conexão* representa a ligação entre uma *interface* e um *receptáculo* de componentes distintos. É importante notar que, para dar suporte à reconfiguração de componentes em tempo de execução, é fundamental manter referências explícitas às dependências de cada componente, pois de outra forma seria impossível determinar as implicações de remover ou repor um componente.

O *OpenCOM* disponibiliza um ambiente de execução padrão (interface *IOpenCOM*) em todo o espaço de endereçamento em que é instalado, que tem como objetivos: (1) gerenciar um repositório de tipos de componentes, (2) permitir a criação e eliminação de instâncias, e (3) servir como ponto central para requisições de conexões entre os componentes desse espaço. Esse ambiente mantém ainda uma representação do estado de conexão de seus componentes sob a forma de um grafo, o que possibilita a ele responder consultas sobre os receptáculos e interfaces que fazem parte de uma dada conexão e fornecer detalhes a respeito dos componentes envolvidos nela.

Todos os componentes *OpenCOM* devem implementar as interfaces de gerência *IReceptacles* e *ILifeCycle*, utilizadas pelo ambiente de execução para realizar suas tarefas. A interface *IReceptacles* é utilizada para gerenciar as conexões estabelecidas nos receptáculos do componente que a implementa. Ela oferece métodos para conectar e desconectar interfaces externas aos receptáculos. Já a interface *ILifeCycle* define métodos de inicialização (*startup*) e destruição (*shutdown*) para o componente, que são invocados pelo ambiente de execução logo após a instanciação e imediatamente antes da remoção da instância do componente, respectivamente.

Além disto, os componentes *OpenCOM* contam com sub-componentes embutidos que implementam as facilidades reflexivas do sistema através de três interfaces exportadas. A interface *IMetaInterception* permite adicionar e remover interceptadores a interfaces oferecidas pelo componente em tempo de execução. Os interceptadores são implementados sob a forma de componentes com interfaces que oferecem métodos que são invocados antes ou depois dos métodos da interface interceptada. A segunda interface exportada, *IMetaArchitecture*, permite ao programador obter identificadores das conexões estabelecidas nos receptáculos do componente. Esses identificadores podem ser posteriormente utilizados junto à interface *IOpenCOM* para obter informações

a respeito das entidades envolvidas na conexão. A última interface reflexiva oferecida é a *IMetaInterface*, que permite a inspeção dos tipos das interfaces e receptáculos oferecidos pelo componente. A figura 2.1 ilustra as entidades do *OpenCOM* descritas acima.

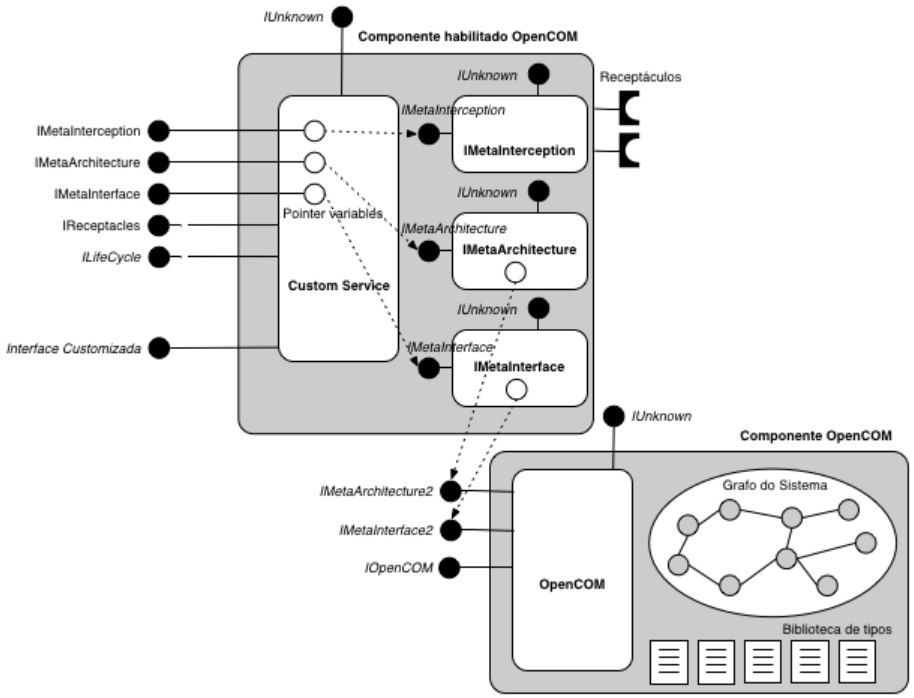

Figura 2.1: Estrutura de componentes *OpenCOM*

Duas importantes abstrações foram incorporadas ao *OpenCOM* em sua segunda versão [31]: *component frameworks* (CFs) e *meta-modelos reflexivos*. CFs foram definidos originalmente em [33] como "coleções de regras e interfaces que governam a interação de um conjunto de componentes", e no *OpenCOM* são representados sob a forma de componentes compostos que aceitam componentes *plug-ins* que adicionam ou modificam o seu comportamento. Sobre o ponto de vista arquitetural, os CFs incorporam políticas, restrições, serviços e facilidades que fazem sentido em um ambiente de execução ou aplicação específica. Os *meta-modelos reflexivos* são representações causalmente conectadas de aspectos selecionados de um sistema alvo, cuja função é permitir a inspeção e adaptação do aspecto selecionado, mantendo uma separação clara entre as questões de construção e de gerenciamento, configuração e adaptação do sistema.

2.3 Comet

O sistema de *middleware Comet* ([9] e [34]) é uma plataforma de construção de sistemas baseados em componentes que tem como principal objetivo o desacoplamento estrutural e em nível de controle dos componentes de suas aplicações. Por desacoplamento estrutural, entendemos que os componentes são localizados de forma transparente e não podem referenciar diretamente outros componentes. A relação de acoplamento é expressa através de conexões tipadas estabelecidas em tempo de execução. O desacoplamento em nível de controle é alcançado através do uso exclusivo de comunicação assíncrona entre os componentes. Desta forma, quando um componente emite um evento não há nenhum impacto no seu fluxo de controle local.

O *Comet* propõe um modelo de comunicação baseado em eventos semanticamente similar ao de sistemas baseados em atores, onde os componentes se comunicam através da troca de eventos tipados. Para tanto, os componentes declaram os tipos de eventos que estão aptos a receber e a enviar, e, em tempo de execução, conexões dinâmicas são estabelecidas entre eles. Naturalmente, uma conexão entre dois componentes só pode ser estabelecida se os tipos de evento envolvidos forem compatíveis. Para tornar este mecanismo mais flexível, foi introduzida uma relação de sub-tipagem entre os eventos: um subtipo de evento sempre pode ser utilizado em um contexto aonde um supertipo é esperado.

Outra característica peculiar do *Comet* é a estrutura arquitetural interna de seus componentes, chamada de *nível de meta-comportamento*. De acordo com essa estrutura, todas as operações envolvidas no processamento de eventos dentro do componente são materializadas em meta-objetos, como ilustra a figura 2.2. Inicialmente, todos os eventos recebidos são enfileirados pelos meta-objetos *MetaReceive* e *MetaInQueue*, de forma a garantir a assincronicidade da entrada de eventos. Paralelamente, os eventos são removidos da fila pelo meta-objeto *MetaInFetch* e encaminhados ao *MetaExec*, que tem a tarefa de associa-los à execução de uma dada tarefa. Eventos de saída também são enfileirados para, posteriormente serem retirados da fila e enviados pelos meta-objetos *MetaOutQueue*, *MetaOutFetch* e *MetaSend*. É importante ressaltar que o comportamento interno padrão dos componentes pode ser alterado através da extensão ou inserção de meta-objetos.

A adaptação dinâmica no *Comet* é garantida pela possibilidade de

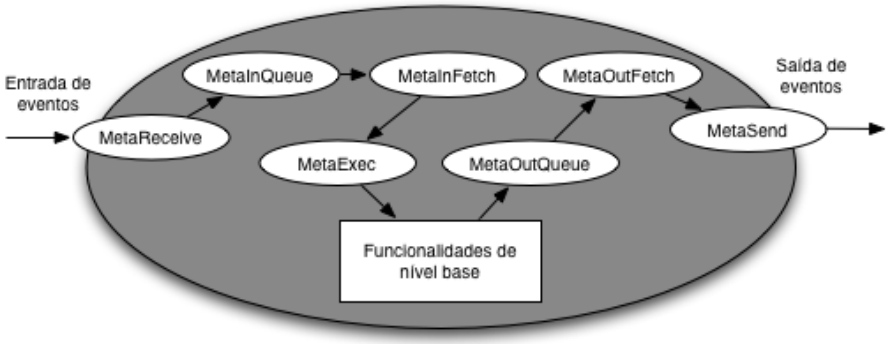

Figura 2.2: Arquitetura interna de componentes *Comet*

alterar a composição das aplicações em tempo de execução, adicionando, removendo e reconectando os seus componentes. Como a arquitetura interna dos componentes no *Comet* é bastante similar à arquitetura externa das suas aplicações - os sub-componentes são acoplados explicitamente e se comunicam através de eventos - o estado interno dos componentes também pode ser adaptado dinamicamente, permitindo adaptações dinâmicas de granularidade mais fina. Para representar as adaptações efetuadas sobre os componentes, são propostas duas abstrações: *papéis* e *protocolos*. Um *protocolo* é a descrição de uma adaptação de uma aplicação, e é composta de *papéis* e funcionalidades, assim como de um estado interno opcional. As funcionalidades descrevem as propriedades do protocolo compartilhadas entre os componentes participantes. Essas podem ser vistas como métodos ou scripts que explicam como usar o protocolo. Os *papéis* descrevem as convenções locais que cada componente participante deve seguir para que o protocolo seja posto em prática. Eles capturam a essência do protocolo, descrevendo o que efetivamente irá mudar no sistema em execução quando ele for aplicado.

2.4 TRAP/J

O *TRAP/J* [35] é uma ferramenta implementada em Java que utiliza mecanismos de reflexão computacional em conjunto com programação orientada a aspectos para gerar versões adaptáveis de programas existentes de maneira transparente. Reflexão computacional é a habilidade de um programa avaliar e possivelmente alterar o seu comportamento durante sua execução. Um sistema reflexivo é composto de objetos em nível base, que implementam suas funcionalidades, e meta objetos, que oferecem um representação externa do

estado dos objetos base. Os meta objetos são conectados aos objetos base de maneira causal, isto é, alterações efetuadas no primeiro grupo serão refletidas no segundo e vice versa. A programação orientada a aspectos é um paradigma de programação que permite a modularização de funcionalidades ortogonais de sistemas, tais como segurança, *logging* ou qualidade de serviço. Essas funcionalidades são implementadas em aspectos que posteriormente, em tempo de compilação ou de execução, são mesclados com a implementação da aplicação para formar um programa completo. Um aspecto é composto de *advices*, trechos de código contendo a implementação do aspecto, e *pointcuts*, pontos de junção na aplicação aonde os *advices* serão executados.

O modelo de execução do *TRAP/J*, ilustrado na figura 2.3, é composto por quatro camadas. A primeira camada, chamada de camada base, contém as classes e objetos originais da aplicação. Para cada classe que se deseja tornar adaptável na camada base é criada uma classe equivalente na camada superior, a camada *wrapper*. Essas classes encapsulam as classes originais e são responsáveis por decidir se as chamadas recebidas pelos objetos serão tratadas pela implementação original ou por alguma nova implementação introduzida em tempo de execução. A camada *meta* contém as meta classes utilizadas por agentes externos para efetuar adaptações nos objetos da aplicação. As adaptações são efetuadas através da adição ou remoção de classes delegadas, que podem sobrescrever um conjunto arbitrário de métodos das classes base selecionadas para adaptação. As classes delegadas ficam na quarta e última camada, a camada delegada. A pré-seleção de classes que devem se tornar adaptáveis foi adotada para minimizar a sobrecarga no desempenho da aplicação, uma vez que a interceptação e redirecionamento das chamadas serão introduzidos apenas nos pontos onde são realmente necessários.

O *TRAP/J* opera em duas fases. Na primeira fase, que ocorre em tempo de compilação, a aplicação existente é convertida em uma aplicação pronta para adaptação. Nesta fase, as classes compiladas da aplicação juntamente com uma lista contendo o nome das classes que se tornarão adaptáveis servem de entrada para o gerador de aspectos e classes reflexivas do *TRAP/J*. Para cada classe candidata a adaptações é gerado um aspecto, uma classe *wrapper* e uma meta classe. O aspecto define *pointcuts* e *advices* de inicialização para os construtores da classe adaptável que instanciam a classe *wrapper* ao invés dela. A classe *wrapper* estende a classe original, sobrescrevendo seus métodos públicos para caso exista um meta objeto associado, passar a ele uma mensagem com informações a respeito do contexto em que o método foi invocado. Caso não exista nenhum meta objeto associado, a invocação é

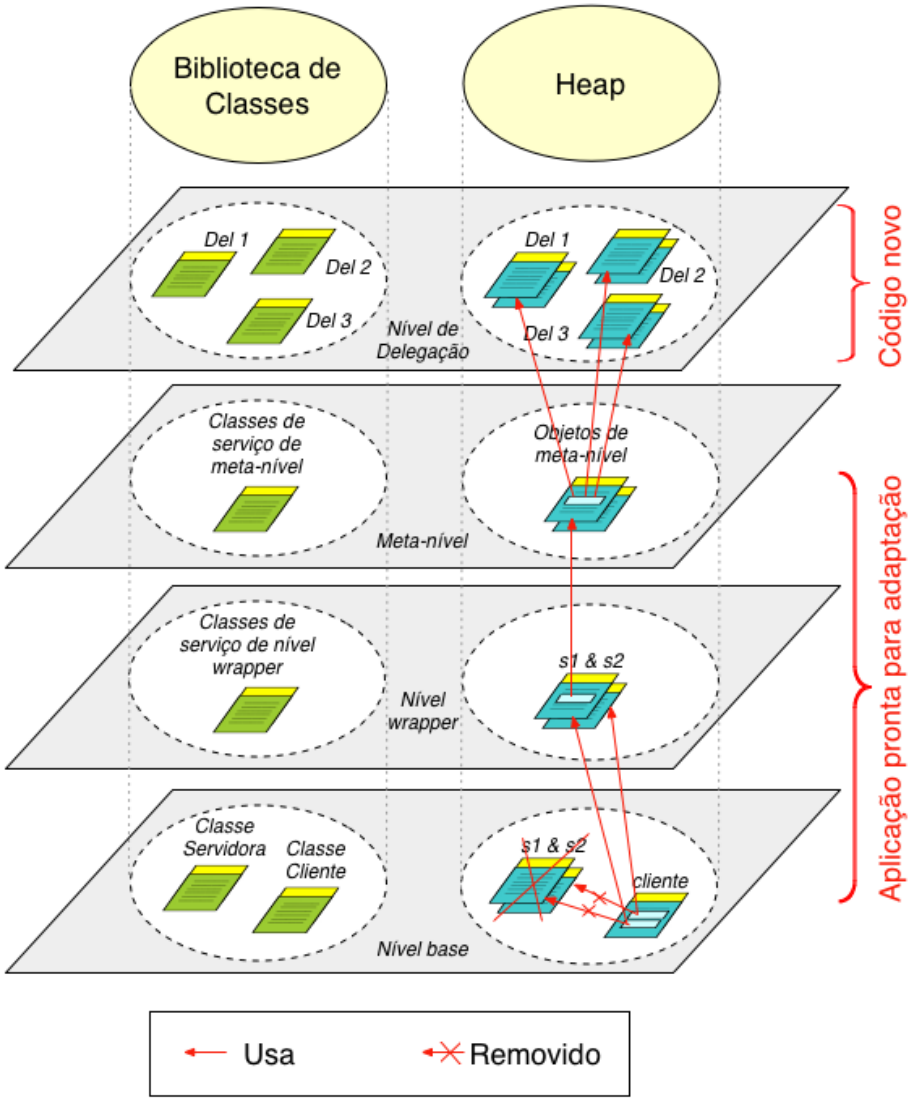

Figura 2.3: Modelo de execução TRAP/J

repassada à implementação base do método na superclasse. A meta classe mantém um conjunto de classes delegadas e, ao receber uma chamada da classe *wrapper*, verifica se o método invocado é implementado por alguma delas. Em caso afirmativo, a chamada é encaminhada dinamicamente ao objeto delegado adequado. Caso contrário, a mensagem é devolvida à implementação original do método na classe base.

A segunda fase da operação do *TRAP/J* ocorre em tempo de execução, quando agentes externos denominados de *composers* utilizam os meta objetos para alterar o comportamento dos objetos adaptáveis durante a sua execução. Para isso, as meta classes geradas implementam a interface *DelegateManagement*, que define métodos de adição e remoção de classes delegadas. Uma classe delegada implementa um conjunto arbitrário de métodos das classes se-

lecionadas para adaptação e pode ser compartilhada por vários meta objetos. A interface de gerência de classes delegadas estende a interface RMI *Remote*, portanto pode ser acessada remotamente utilizando essa tecnologia.

2.5 LuaCCM

A referência que utilizamos para desenvolver os mecanismos de adaptação dinâmica descritos neste trabalho foi o *LuaCCM* [23], um trabalho que pertence à última geração de sistemas de adaptação dinâmica desenvolvidos por nosso grupo de pesquisa. O *LuaCCM* se baseia no modelo de componentes CORBA (CCM - *CORBA Component Model*) [24], uma definição adicional ao padrão CORBA que oferece abstrações para a representação de componentes com interfaces bem definidas. Conforme veremos na seção seguinte, o CCM é um modelo bastante abrangente, que atende os principais requisitos de modularização das aplicações às quais nosso trabalho se destina. Algumas abstrações do CCM, como por exemplo o contêiner de componentes, são especialmente úteis na implementação dos mecanismos de adaptação dinâmica.

O *LuaCCM* é uma implementação em Lua do modelo de componentes CCM adaptada para permitir a construção de componentes dinamicamente adaptáveis através de alterações na estrutura e implementação desses componentes. Ele oferece ainda um conjunto de abstrações que formalizam o processo de adaptação de um sistema, agrupando as adaptações em seus componentes. A implementação do *LuaCCM* é baseada no *LuaOrb* [4, 36], uma ferramenta de programação CORBA para a linguagem Lua que adota um *binding* dinâmico entre as duas tecnologias.

2.5.1 O modelo CCM

O modelo de componentes CORBA define uma série de abstrações que permitem projetar sistemas isolando suas funcionalidades em unidades reutilizáveis, de acordo com o paradigma de programação orientada a componentes.

Neste modelo, componentes são as unidades básicas de construção das aplicações. Eles têm seu comportamento encapsulado e expõem suas funciona-

lidades e dependências através de interfaces bem definidas. Uma vez selecionados os componentes, um sistema é inteiramente construído através da conexão das interfaces de suas instâncias.

Uma conexão entre dois componentes pode ser orientada a interface ou a evento. Nas conexões orientadas a interface, o componente cliente realiza chamadas a operações definidas em uma interface disponibilizada pelo componente servidor. Já nas conexões orientadas a eventos, o componente consumidor se cadastra para receber eventos de um componente produtor.

Os pontos de conexão dos componentes são chamados de portas. O modelo do CCM define quatro tipos de portas, que determinam o tipo de conexão e o papel que o componente desempenha nela. Os quatro tipos de portas definidas no CCM são os seguintes:

- Faceta: Representa uma interface oferecida por um componente servidor em uma conexão orientada a interface. Uma faceta é composta de operações e atributos.
- Receptáculo: Representa uma dependência de um componente cliente de uma faceta em uma conexão orientada a interface. Os receptáculos podem aceitar apenas uma conexão ou conexões múltiplas.
- Fonte de Evento: Representa um canal de geração de eventos disponibilizado por um componente produtor em uma conexão orientada a eventos. As fontes de eventos podem ser de dois tipos: *emissoras de eventos*, que enviam eventos a um único receptor, e *publicadoras de eventos*, que enviam os eventos a um ou mais receptores.
- Receptor de eventos: Representa um canal de recebimento de eventos, que pode ser conectado a uma fonte de eventos de um componente produtor em uma conexão orientada a eventos.

A figura 2.4 apresenta uma representação gráfica de um componente com todos os tipos possíveis de portas e em seguida ilustra uma composição de três de componentes, realizada através da conexão de suas portas.

Outro conceito importante introduzido pelo modelo CCM é o de contêineres de componentes, que são abstrações que definem ambientes de execução protegidos onde as implementações de componentes são instaladas, criadas e executadas. Os contêineres intermedeiam a interação entre a implementação dos componentes e o mundo externo nas duas direções: chamadas

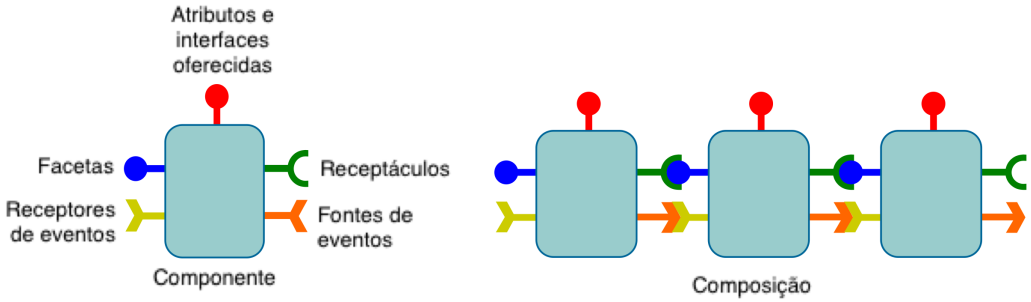

Figura 2.4: Componentes CCM

de componentes externos direcionadas ao componente em questão são recebidas pelo contêiner e encaminhadas a sua implementação. Já as chamadas da implementação do componente destinadas a componentes externos conectados a ele, são realizadas sobre objetos locais disponibilizados pelo contêiner e traduzidas em chamadas remotas. O contêiner de componentes pode ainda oferecer serviços diversos (segurança, persistência, transações, etc.) à implementação dos componentes sob a forma de APIs internas, disponibilizadas em seu ambiente de execução.

O uso de contêineres de componentes minimiza o esforço de implementação do desenvolvedor de componentes, que deixa de ser responsável pelo código de conexão entre componentes, e padroniza o desenvolvimento e a implantação dos componentes. A figura 2.5 ilustra o conceito de contêineres de componentes, mostrando um contêiner que oferece facetas ao mundo externo e delega as chamadas recebidas a uma interface de *callback* disponibilizada pelo objeto que contém a sua implementação. O contêiner oferece ainda um conjunto de interfaces internas, através das quais ele disponibiliza seus serviços para as implementações que ele hospeda.

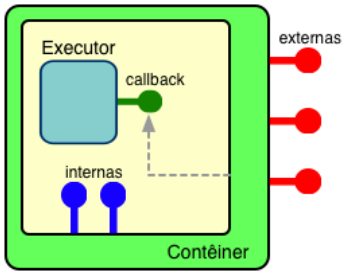

Figura 2.5: Contêiner de Componentes

2.5.2
Adaptação dinâmica no LuaCCM

O modelo do CCM por si só já permite um certo grau de adaptabilidade, uma vez que os componentes das aplicações podem ser conectados e desconectados em tempo de execução. É possível introduzir novos componentes e até mesmo substituir componentes antigos desde que a estrutura das portas definida no projeto da aplicação seja respeitada.

Para que este tipo de adaptação ocorra, porém, é necessário que os pontos de adaptação sejam previstos na fase do projeto da aplicação. As funcionalidades que poderão vir a ser substituídas devem ser isoladas em componentes a parte, os componentes do sistema que terão seu comportamento estendido devem definir portas adequadas para dar suporte a novas conexões e a estrutura das portas do sistema deve ser genérica o suficiente para acomodar o comportamento de novos componentes. Infelizmente, sabemos que muitas das situações que demandam algum tipo de adaptação nos sistemas surge quando estes entram em produção e não podem ser previstas com antecedência.

O *LuaCCM* estende a capacidade inicial de adaptação do CCM, adicionando a ele uma série de mecanismos e abstrações. Podemos dividir os mecanismos introduzidos em três grupos: manipulação da estrutura de um componente, alterações em sua implementação e utilização de interceptadores. Todos esses mecanismos são disponibilizados através da interface *LuaCCM::Adaptable* (listagem 2.1).

O mecanismo de manipulação da estrutura de um componente permite adicionar, remover e substituir as funcionalidades oferecidas por um componente através de alterações em seu conjunto de portas. A interface *LuaCCM::Adaptable* oferece métodos para adicionar portas dos cinco tipos (*addfacet*, *addreceptacle*, *addemitter*, *addpublisher* e *addconsumer*) e um método para remover uma porta qualquer a partir de seu nome (*removeport*). Para adicionar qualquer tipo de porta, é necessário informar um nome (parâmetro *name*), que será utilizado para identificar a porta no componente onde ela será instalada. No caso de uma faceta, é necessário fornecer também a definição de sua interface em IDL (parâmetro *iface*) e o código Lua com a sua implementação (parâmetro *code*). Para receptáculos, também deve ser fornecida a definição da interface e um indicador que sinaliza se serão aceitas conexões múltiplas (parâmetro *ismultiple*). Para portas emissoras e consumidoras de eventos, é necessário informar a definição do tipo de evento trafegado

(parâmetro *eventtype*), sendo que para as portas consumidoras é necessário informar também o código Lua com a sua implementação (parâmetro *code*). Todos os métodos de adição de portas lançam a exceção *PortNameAlreadyExists*, no caso da porta informada já existir. As operações de adição de facetas e de consumidores lançam também a exceção *LuaCodeError*, caso o código fornecido contenha erros de compilação. O método de remoção de portas lança a exceção *InvalidPortName* no caso de não existir nenhuma porta com o nome fornecido.

A possibilidade de efetuar alterações na implementação de um componente através da substituição de seu código fonte é um recurso extremamente poderoso, que permite modificar o comportamento de um componente sem alterar a sua estrutura, evitando assim a necessidade de alterações nos componentes que interagem com o componente alterado. Este recurso deve ser utilizado com cautela, pois não existem mecanismos que permitam reverter uma adaptação caso ela contenha algum erro que comprometa o funcionamento da aplicação. A funcionalidade de alteração de implementação é oferecida pelo *LuaCCM* através do método *setimplementation*, que recebe o nome de uma porta e o código Lua que deve ser usado para substituir a sua implementação. Este método lança exceções caso não existia nenhuma porta com o nome fornecido (exceção *InvalidPortName*) ou caso o código fornecido contenha erros de compilação (exceção *LuaCodeError*).

O mecanismo de interceptadores oferece a possibilidade de introduzir processamento logo antes e imediatamente depois das interações realizadas pelas portas dos componentes. No caso de facetas e receptáculos, esse processamento é realizado antes da invocação de uma operação e logo depois de seu retorno. Já no caso de fontes e receptores de eventos, ele é realizado antes e depois da emissão e do recebimento dos eventos. No *LuaCCM*, a implementação de um interceptador deve conter dois métodos: *before*, que, como o nome sugere, é executado antes da chamada na porta interceptada, e *after*, executado depois da chamada. Cada porta pode ter um interceptador, que, no caso de facetas e receptáculos, é invocado para todas as suas operações. Para adicionar um interceptador a uma porta, é utilizada a operação *intercept*, que recebe como parâmetro o nome da porta a interceptar (parâmetro *point*) e o código Lua da implementação do interceptador (parâmetro *code*). Para remover um interceptador, basta invocar a operação *unintercept*, passando como parâmetro o nome da porta (parâmetro *point*) que contém o interceptador que deve ser removido. Ambas as operações lançam uma exceção (exceção *InvalidPortName*) caso a porta informada não exista. A operação de adição de interceptadores lança

ainda exceções nos casos do interceptador não seguir o padrão descrito acima (exceção *InvalidInterceptor*) e do código Lua fornecido para sua implementação conter erros de compilação (exceção *LuaCodeError*). A operação de remoção de interceptadores lança uma exceção caso não exista nenhum interceptador cadastrado na porta informada (exceção *NoInterceptor*).

Interceptadores são especialmente úteis em dois casos específicos. O primeiro ocorre quando é necessário efetuar pequenas alterações temporárias em um componente. Como os interceptadores podem ser adicionados e removidos com facilidade, fica bem mais fácil introduzir pequenas mudanças desta forma do que substituindo toda a implementação da porta. O segundo caso ocorre quando surge a necessidade de adicionar ao componente funcionalidades ortogonais à sua finalidade original. A introdução de mecanismos de segurança ou de *log*, por exemplo, que tradicionalmente envolveria alterações em diversos pontos da implementação do componente, poderia ser muito simplificada se implementada utilizando interceptadores. Os trabalhos [37], [38] e [39] mostram como o uso de programação orientada a aspectos, um paradigma baseado no conceito de interceptadores, contribui para a implementação de funcionalidades ortogonais em sistemas.

```
1  #include<CCM.idl>
2
3  module LuaCCM {
4          typedef string LuaCode;
5
6          exception PortNameAlreadyExists {};
7          exception InvalidComponent {};
8          exception InvalidPortName {};
9          exception InvalidInterceptor {};
10         exception NoInterceptor {};
11         exception LuaCodeError {
12                 string message;
13                 string stack;
14         };
15
16         interface Adaptable {
17                 void addfacet(in string name,
18                         in string iface,
19                         in LuaCode code)
20                         raises(PortNameAlreadyExists,
21                                 LuaCodeError);
22
23                 void addreceptacle(in string name,
24                         in string iface,
25                         in boolean ismultiple)
```

```
26                          raises(PortNameAlreadyExists);
27
28                  void addemitter(in string name,
29                          in string eventtype)
30                          raises(PortNameAlreadyExists);
31
32                  void addpublisher(in string name,
33                          in string eventtype)
34                          raises(PortNameAlreadyExists);
35
36                  void addconsumer(in string name,
37                          in string eventtype,
38                          in LuaCode code)
39                          raises(PortNameAlreadyExists,
40                                  LuaCodeError);
41
42                  void removeport(in string name)
43                          raises(InvalidPortName);
44
45                  void setbase(in string componentname)
46                          raises(InvalidComponent);
47
48                  void setimplementation(in string name,
49                          in LuaCode code)
50                          raises(InvalidPortName,
51                                  LuaCodeError);
52
53                  void intercept(in string point,
54                          in LuaCode code)
55                          raises(InvalidPortName,
56                                  InvalidInterceptor,
57                                  LuaCodeError);
58
59                  void unintercept(in string point)
60                          raises(InvalidPortName,
61                                  NoInterceptor);
62          };
63  ...
```

Listing 2.1: Interface LuaCCM::Adaptable

Além dos mecanismos apontados acima, o *LuaCCM* utiliza duas abstrações, introduzidas originalmente em [9], que permitem realizar adaptações a sistemas de uma forma mais simples e estruturada. A primeira abstração é o *papel*, que sintetiza um determinado comportamento de um componente. Um *papel* define um conjunto de alterações as quais um componente deve ser sub-

metido para que possa oferecer um novo comportamento (ou "desempenhar um novo papel"). No modelo CCM essa alteração é caracterizada pela definição de novas portas do componente ou alteração da semântica (i.e., implementação) das portas existentes, de forma que novas interações possam ser feitas através delas.

Outra abstração utilizada pelo *LuaCCM* é o *protocolo*. Um *protocolo* nada mais é do que um conjunto de papéis que devem ser aplicados a componentes de um sistema para que este seja adaptado a uma nova configuração. A aplicação de um *protocolo* no *LuaCCM* é feita através de *scripts* que estabelecem novas conexões entre os componentes (e.g., utilizando as características impostas por um papel) e disparam eventos ou requisições.

2.6 Considerações Finais

Em relação à localização do código adaptativo, podemos dividir os trabalhos analisados basicamente em dois grupos. O primeiro grupo, do qual fazem parte *DynamicTAO*, *OpenCOM*, *Comet* e *LuaCCM*, é o grupo dos sistemas que viabilizam a adaptação das aplicações através de uma camada de *middleware* que implementa internamente os mecanismos de adaptação e os disponibiliza de forma transparente para o desenvolvedor. O segundo grupo, do qual faz parte o *TRAP/J*, adota a abordagem de acrescentar ao código fonte da aplicação funcionalidades de interceptação e redirecionamento de forma que, após a compilação, é gerada uma versão adaptável da aplicação original. Em relação a granularidade das adaptações, enquanto o *DynamicTAO* e o *OpenCOM* permitem apenas alterações através de reconexões entre os componentes, o *LuaCCM* permite também a substituição da sua implementação. Já o *Comet* e o *TRAP/J* permitem adaptações na estrutura interna dos componentes. O *TRAP/J* é o único dos sistemas que necessita que os pontos de adaptação sejam definidos em tempo de compilação, no entanto é o único que não depende de plataformas de *middleware* para funcionar.

Como o foco do sistema desenvolvido neste trabalho são aplicações distribuídas de alta disponibilidade, assumimos que o melhor local para implementar os mecanismos de adaptação é a camada de distribuição do *middleware*. Uma vez que boa parte das situações que demandam adaptações surgem de erros identificados somente em tempo de execução, acreditamos que não podemos contar com a definição dos pontos de adaptação em tempo de compilação.

Por fim, adaptações efetuadas através da reconexão de componentes são insuficientes para efetuar alterações em pontos específicos dos componentes. A possibilidade de alterar a implementação de funcionalidades específicas dos componentes em tempo de execução é importante para permitir que os componentes evoluam sem que seja necessário interromper completamente o seu funcionamento.

Uma ponto de extrema importância que ainda não é tratado de maneira adequada em grande parte dos sistemas avaliados é a questão da consistência das aplicações durante a adaptação. Sistemas dinamicamente adaptáveis devem garantir que suas aplicações continuem executando de maneira segura, ou ao menos sinalizem um erro adequado durante a aplicação de uma adaptação. Idealmente, as adaptações devem ser tratadas de maneira atômica, de forma que qualquer requisição recebida durante a sua aplicação seja tratada pela versão anterior do sistema ou, se possível, enfileirada para ser tratada após a conclusão da adaptação. Esse problema pode se tornar especialmente complexo em alguns casos, como por exemplo quando ocorrem mudanças nas interfaces externas da aplicação ou quando são aplicadas adaptações complexas compostas por adaptações menores dependentes entre si. O sistema LuaCCM em particular ainda não trata esta questão, porém esforços no sentido de implementar adaptações de forma atômica estão previstos entre seus trabalhos futuros.