4

Infra-Estrutura de Execução para o OpenBus

Para dar suporte à implantação e execução de componentes de software em um ambiente distribuído, não só é necessário prover serviços que permitam realizar estas tarefas de forma simples, mas também é mais que desejável permitir a independência de linguagens. Nosso modelo permite tal facilidade, devido ao uso da tecnologia CORBA, bastante madura e utilizada. Embora muitos trabalhos enveredem por diferentes tecnologias como Serviços Web ou tecnologias baseadas em Java e existam críticas a CORBA [45], para os nossos intuitos a tecnologia se adequa muito bem. É possível construir aplicações de forma simples, principalmente em linguagens com gerenciamento automático de memória como Java e Lua, e ao mesmo tempo manter um bom desempenho.

Durante o estudo de outros trabalhos, identificamos que muitos contam com uma forte curva de aprendizado, refletida na grande quantidade de regras, documentação, tamanho de código e quantidade de serviços. Por isso, tentamos minimizar estes problemas ao especificar nossa infra-estrutura de execução.

Nosso maior objetivo é prover facilidades para a execução de aplicações distribuídas, que sejam representadas por conjuntos de componentes. Esses componentes não precisam estar necessariamente na mesma máquina. A composição em componentes ajuda a manter a simplicidade no desenvolvimento através de uma melhor divisão de tarefas, e dá a possibilidade de um melhor aproveitamento dos recursos distribuídos.

O serviço mais básico da nossa infra-estrutura de execução de componentes chama-se Nó de Execução. Este serviço representa um nó do ambiente distribuído, conta com seu próprio processo e permite a criação de contêineres de componentes, que por sua vez serão os verdadeiros responsáveis pela carga e instanciação dos componentes. Nós de Execução podem regular o acesso à implantação de componentes de aplicações, através de um serviço externo como o Serviço de Controle de Acesso oferecido pelo OpenBus.

O Contêiner de Componentes, que representa outro serviço da infra-estrutura, também conta com seu próprio processo. Desta forma, atua como um ambiente fechado para a carga e execução dos componentes de aplicação desenvolvidos por usuários da infra-estrutura. Em um contêiner podem ser carregados diferentes componentes, funcionalmente acoplados ou não, mas componentes fortemente acoplados podem se aproveitar de otimizações específicas ao serem carregados em um mesmo contêiner.

Como mencionado anteriormente, neste trabalho consideramos o processo de implantação como as ações necessárias para que um componente ou serviço seja instalado em um nó, possa ser carregado e possa ter suas dependências externas supridas. Assim, precisamos ainda de um serviço que permita a instalação de implementações de componentes. O serviço que provê esta facilidade chama-se Repositório de Componentes, e é utilizado por contêineres para que componentes ainda não carregados possam ser obtidos. A utilização de um repositório torna desnecessário manter implementações de componentes em todas as máquinas, simplificando o processo de instalação. Nossa infra-estrutura ainda não fornece facilidades para a carga automática de dependências externas, mas isso é um importante trabalho futuro, como mencionamos na Seção 6.1.

A infra-estrutura de execução fornece bibliotecas para o suporte ao desenvolvimento de componentes em Lua, Java e C++, mas em geral as implementações dos serviços básicos da infra-estrutura foram feitas em Lua. Desses serviços, apenas o Contêiner de Componentes demanda implementações em cada linguagem, pois conta com tratamentos e otimizações diferentes para carregar componentes de sua linguagem nativa. No entanto, este trabalho trata apenas das implementações Lua. Na Figura 4.1 podemos ter uma visão geral dos serviços, que serão melhor apresentados ao longo das seções deste capítulo, e suas relações.

Todos os serviços da infra-estrutura de execução são também componentes que aderem ao modelo especificado na Seção 3.1. Isto permite que a própria infra-estrutura tire proveito das facilidades providas por este modelo, como a reflexão computacional ou a (re)configuração de suas dependências em tempo de execução.

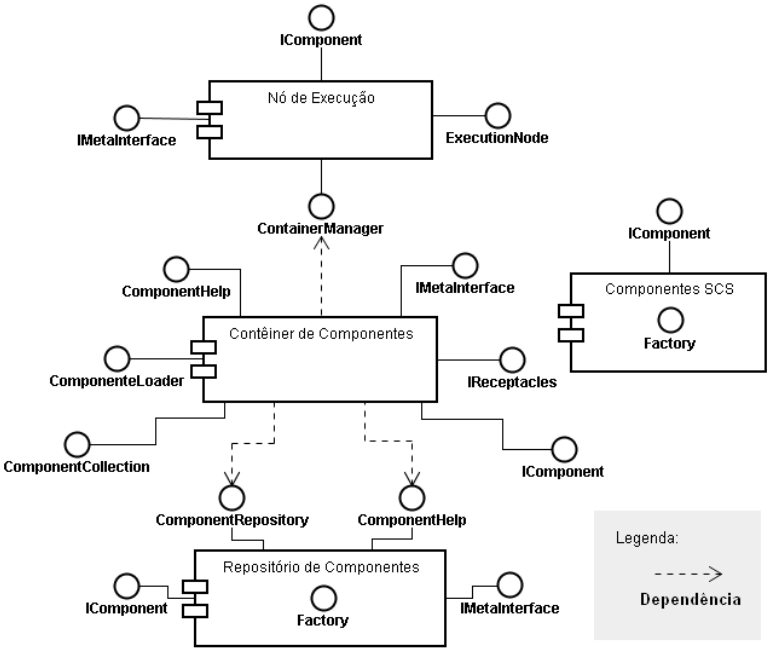

Figura 4.1: Serviços da Infra-Estrutura de Execução e suas Relações

4.1
Nó de Execução

Sendo nossa arquitetura inerentemente distribuída, é necessário criar uma representação para os nós da rede. Para isto, criamos um serviço chamado Nó de Execução. Este serviço atua como a porta de entrada para uma máquina específica do domínio distribuído e usualmente há apenas um por máquina, com uma porta TCP fixa. Na arquitetura atual, sua função principal é a de criar contêineres de componentes (os quais abordaremos na próxima seção) e gerenciá-los. Estes contêineres executam em um novo processo, permitindo que sejam escritos em diversas linguagens. Um Nó de Execução pode ser reiniciado sem perder as referências para seus contêineres, mas ao receber uma chamada para ser finalizado, tenta finalizar todos os seus contêineres.

Em nosso trabalho, optamos por não permitir a criação de novos processos pelo Nó de Execução que não sejam processos contêiner, pois a utilização e avaliação de contêineres era um de nossos objetivos. Além disso, o projeto SGA, mencionado na Seção 5.5, já implementa essa outra abordagem em nosso grupo.

Todo Nó de Execução tem um conjunto de propriedades inicial, que representa suas configurações de inicialização. Este conjunto é padronizado

e com algumas propriedades obrigatórias. No entanto, nós de execução contam com uma faceta (as facetas serão descritas mais adiante nesta mesma seção) que permite que propriedades sejam adicionadas e/ou alteradas em tempo de execução. Estas propriedades podem também ser definidas como somente-leitura. Segue uma lista das propriedades obrigatórias e seus significados:

- *platform*: Informa a plataforma ou sistema operacional no qual o Nó de Execução está funcionando. Atualmente são suportadas as plataformas Windows e UNIX, com valores "WIN", "WINDOWS", "WINNT", "WINDOWSNT", "WINDOWS_NT", "LINUX"ou "UNIX". Esta propriedade é definida como somente-leitura.
- *lua_name*: Nome do executável de lua que deverá ser utilizado como padrão para a criação de contêineres lua. Exemplos: "lua", "lua5.1.3", "lua.exe", "luajit.exe". Esta propriedade é definida como somente-leitura.

Segue uma lista de propriedades opcionais que podem ser utilizadas pelo próprio Nó de Execução:

- *host*: Nome ou endereço IP da máquina hospedeira do serviço na rede. Esta propriedade é definida como somente-leitura.
- *port*: Porta desejada para o processo do Nó de Execução. Esta propriedade é definida como somente-leitura.
- *timeout*: Quantidade de tempo máxima, em segundos, que o Nó de Execução deve aguardar pelo registro de um contêiner recentemente criado e inicializado. Esta propriedade também é utilizada para a espera realizada no método *shutdown*(), relativa à espera pelos contêineres sendo finalizados. Caso um valor não seja especificado, será utilizado o valor de 20 segundos.
- *machines*: Lista de nomes (separados por espaços) de executáveis que poderão ser aceitos para a execução de contêineres, não necessariamente executáveis lua. O uso desta propriedade será melhor descrito na Seção 4.1.3. Exemplo: "lua lua5.1 luajit java".
- *restart*: Quando ativada, esta propriedade faz com que o Nó de Execução não termine os processos contêiner iniciados por este nó, ao receber uma chamada *shutdown*. O uso desta propriedade será melhor descrito na Seção 4.1.2. Valores: "true"ou "false".

- *initfile*: Informa o nome de um arquivo Lua a ser carregado e executado durante a chamada ao método *startup*(). Este arquivo pode ser utilizado para configurações adicionais pós-inicialização, por exemplo. Esta propriedade é definida como somente-leitura.
- *use_core_services*: Indica se outros serviços OpenBus, como o Serviço de Controle de Acesso e o Serviço de Registro, devem ser utilizados ou não. Valores: "true"ou "false". O valor padrão é falso. Esta propriedade é definida como somente-leitura.
- *accessControlServerHostName*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o nome ou endereço IP da máquina hospedeira do Serviço de Controle de Acesso do barramento OpenBus. Esta propriedade é definida como somente-leitura.
- *accessControlServerHostPort*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa a porta utilizada pelo processo do Serviço de Controle de Acesso do barramento OpenBus. Esta propriedade é definida como somente-leitura.
- *privateKeyFile*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o caminho, incluindo o nome do arquivo, para a chave de acesso ao barramento OpenBus. Esta propriedade é definida como somente-leitura.
- *accessControlServiceCertificateFile*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o caminho, incluindo o nome do arquivo, para o certificado a ser utilizado no barramento OpenBus. Esta propriedade é definida como somente-leitura.

Nós de execução são capazes de criar contêineres de componentes em diferentes linguagens. Uma funcionalidade interessante que escolhemos dar suporte foi a possibilidade de utilização de diferentes máquinas virtuais, podendo ser especificadas para cada processo contêiner. Esta funcionalidade, e outras características e funcionalidades principais do Nó de Execução, serão descritas a seguir.

4.1.1
Interação com Outros Serviços OpenBus

Aos nós de execução é permitido trabalhar em conjunto com os serviços já existentes do barramento OpenBus, como o Serviço de Controle de Acesso ou o Serviço de Registro, de forma opcional. Caso esta opção seja utilizada, todos os contêineres criados são informados por seu Nó de Execução, para que participem também do barramento formado. Isto é importante pois com a utilização do Serviço de Controle de Acesso, a comunicação tanto do contêiner como de seus componentes com quaisquer objetos remotos será bloqueada, caso o contêiner não tenha acesso ao barramento.

4.1.2
Reinicialização

Em todo sistema complexo e em seus estágios iniciais de evolução, é mais do que razoável assumir que seus administradores terão problemas eventuais, levando à necessidade de manutenção do sistema. Considerando isto, vimos que seria relevante e útil fornecer uma forma de reiniciar um Nó de Execução, sem que fosse necessário reiniciar os contêineres e seus componentes. O Nó de Execução pode ser reiniciado remotamente, com os seguintes passos:

1. Ativar a propriedade *restart*;
2. Realizar uma chamada ao método *shutdown* da interface IComponent;

A reinicialização de um Nó de Execução não precisa necessariamente ter qualquer efeito sobre seus contêineres, a não ser por um breve período de indisponibilidade do serviço Nó de Execução. De acordo com nosso modelo, todo contêiner se registra em seu nó ao ser inicializado, enviando uma referência remota para sua interface IComponent. Esta referência é persistida em um arquivo, e são persistidas também outras informações sobre cada contêiner, como suas identificações.

Caso a propriedade *restart* esteja ativada, um novo processo Nó de Execução é automaticamente iniciado ao fim da execução de seu método *shutdown*. O novo nó procura por referências persistidas e, caso as encontre, tenta reativá-las. Obtendo sucesso com uma referência, o processo de registro deste contêiner é novamente realizado, e o processo é repetido para todas as

referências ainda válidas. Ao final, todas as referências inválidas são removidas. Com isto, contêineres podem continuar funcionando enquanto um Nó de Execução é reiniciado, sendo recuperados logo em seguida.

4.1.3
Máquinas Virtuais para Contêineres

Nem sempre é possível que a máquina virtual padrão, Lua ou Java, atenda a todos os requisitos de uma aplicação. Por exemplo, pode ser necessário utilizar uma representação númerica diferente das oferecidas pela máquina padrão, como "long double"em Lua. Um outro exemplo é o uso de uma máquina virtual que atue de forma diferenciada como a máquina LuaJIT, que permite utilizar compilação *Just-In-Time* em Lua. Por isso, opcionalmente pode-se fornecer o nome da máquina virtual desejada ao se iniciar um contêiner. Obviamente, todos os componentes deste contêiner serão afetados pela escolha.

O Nó de Execução só utiliza, entretanto, máquinas registradas em sua configuração. Como mencionado anteriormente, existe a propriedade *machines* para definí-las. É necessário que o caminho para a máquina esteja registrado na variável de ambiente "*path*"do sistema operacional, ou que o caminho completo seja fornecido na propriedade. O acesso à faceta de modificação das propriedades pode ser restringido a administradores, através do uso do Serviço de Controle de Acesso do OpenBus, por exemplo.

4.1.4
Facetas

No Apêndice B.2 estão listadas as facetas do Nó de Execução, com explicações sobre cada método. Não são listadas todas as facetas do modelo de componentes básico, pois funcionam da mesma forma que foram descritas na Seção 3.2. A única exceção é a faceta IComponent, pois esta teve métodos sobrescritos. Somente os métodos sobrescritos serão mencionados. O Nó de Execução provê todas as facetas básicas, IComponent, IReceptacles e IMetaInterface.

O Apêndice A inclui todas as operações das facetas exclusivas ao Nó de Execução no módulo *execution_node*, além de outras comuns a outros serviços no módulo *auxiliar*.

4.1.5
Exemplo de Uso do Nó de Execução

Para ilustrar o uso mais comum de um Nó de Execução, apresentaremos aqui um pequeno trecho de código. As chamadas serão todas síncronas e sem tratamento de exceções por motivo de clareza.

Caso esteja participando de um barramento OpenBus, o Nó de Execução tenta se registrar no Serviço de Registro, passando sua referência para a faceta IComponent. No exemplo abaixo, por motivo de simplificação, não utilizaremos o barramento OpenBus e assumiremos que o cliente remoto tem acesso a um arquivo IOR que represente uma referência para o Nó de Execução.

Agora podemos acessá-lo remotamente, com chamadas como:

Listing 4.1: Exemplo de uso de um Nó de Execução

```
1  -- A referência obtida remete à interface IComponent.
2  EN = oil.newproxy(assert(oil.readfrom("execution_node.ior")))
3
4  -- Com ela podemos obter qualquer outra faceta, como a
       ExecutionNode.
5  enFacet = EN:getFacet("scs::execution_node::ExecutionNode")
6  -- Precisamos encontrar a interface mais específica, pois
       recebemos um objeto do tipo Object nesta chamada, de
       acordo com a IDL.
7  enFacet = enFacet:_narrow()
8
9    -- Criação de um contêiner
10 local containerName = "ComponentContainer"
11 -- Aqui daremos apenas uma propriedade ao contêiner. Para
       nossa implementação do Nó de Execução, isto não seria
       necessário para contêineres lua com a máquina virtual
       padrão, mas colocamos aqui como exemplo.
12 local containerPropertySeq = { { name = "language", value =
       "lua", read_only = true } }
13
14 -- CC será a faceta IComponent do contêiner
15 CC = enFacet:startContainer(containerName,
       containerPropertySeq)
16
17 -- Destruiremos agora o contêiner a título de exemplo. O
       procedimento normal seria uma chamada shutdown() a ele ou
        ao nó.
18 enFacet:killContainer(containerName)
```

```
19
20    -- Por fim, pediremos ao nó que termine seu processo de forma
          controlada.
21    EN:shutdown()
```

4.2
Contêiner de Componentes

Contêineres atuam como encapsuladores de componentes de aplicação, e provêem um espaço de endereçamento próprio para estes. Portanto, são responsáveis principalmente por sua carga, descarga, e por manter o registro de componentes carregados. Desta forma, componentes em um mesmo contêiner são executados em um mesmo processo, que é o do contêiner. Obviamente, o mesmo não ocorre para componentes carregados em contêineres diferentes, ainda que estejam no mesmo Nó de Execução.

Nesta seção descreveremos apenas a implementação Lua do Contêiner de Componentes. Todo contêiner Lua mantém em memória e disco os componentes que conhece, ou seja, os componentes de sistema e qualquer outro que já tenha sido carregado por ele. Além disso, faz também uso de um componente chamado ComponentRepository que, como seu nome indica, atua como um repositório de componentes. Um receptáculo múltiplo existe para a conexão com repositórios, e toda vez que um componente a ser carregado não for encontrado localmente, o contêiner o procurará nos repositórios conectados. Contêineres fornecem também uma faceta para a obtenção de um texto de ajuda sobre os componentes conhecidos. Estes textos também são obtidos a partir de repositórios e em seguida armazenados localmente.

Componentes de aplicação, ou seja, componentes que não sejam um Nó de Execução ou um Contêiner de Componentes (pois estes têm seus próprios processos), devem implementar um objeto "Factory", que será responsável por criar uma nova instância (ou devolver sempre a mesma, caso o componente atue de acordo com o padrão *singleton*) do componente. Assim, o contêiner pode carregar o componente em memória sem instanciá-lo, fazendo-o apenas quando receber um pedido para tal. Além disso, a escolha de atuação como *singleton* ou não fica a cargo do próprio componente, e não do sistema.

Ao trabalhar com contêineres ganha-se efetivamente um nível de in-

direção, que pode ser aproveitado para fornecer uma gama de facilidades e serviços que afetem todos os seus componentes instanciados. Em contrapartida, pode também acarretar em perda de desempenho, o que será minimamente abordado no Capítulo 5 por fugir ao escopo deste trabalho. Esse nível de indireção pode ser aproveitado de diferentes formas, caso um contêiner conte com acesso restrito e contenha apenas componentes com funcionalidades fortemente acopladas, ou o contrário. Por exemplo, pode-se aplicar uma política de segurança diferente para contêineres públicos, através de interceptadores específicos para essa tarefa que tratam todas as comunicações de todos os componentes de um contêiner.

Outra vantagem do uso de contêiners que possibilitem a carga de diferentes componentes é a facilidade de troca de módulos não só remotos, mas também locais, em tempo de execução. Dependências locais podem ser representadas da mesma forma que as remotas, ou seja, como componentes conectados a receptáculos. No entanto, podem também ser carregadas diretamente por um componente (devemos notar que um contêiner pode limitar as funcionalidades da máquina virtual Lua disponíveis aos seus componentes, como funções de carga), geralmente em sua fábrica. Não há uma forma padronizada de se realizar remotamente a carga de dependências locais. Pode-se, por exemplo, fornecer uma faceta com estas funcionalidades. Pode-se também criar um componente configurador que realize a carga dos módulos locais necessários, carregado no mesmo contêiner para que compartilhem a mesma área de memória. Futuramente, pretendemos estender o modelo SCS e a infra-estrutura de execução para que seja possível a carga e conexão local com outras tecnologias que não CORBA e SCS, como bibliotecas dinâmicas em geral, componentes COM, etc.

Essa futura funcionalidade pode ser vista como uma (re)configuração do componente. Reconfigurações complexas como essa geralmente exigem que o componente não esteja realizando nem atendendo a nenhuma requisição remota, pois seu estado será modificado. Para que o processo possa ocorrer em tempo de execução, podemos novamente nos aproveitar do nível de indireção fornecido pelo contêiner e prover uma funcionalidade de suspensão das comunicações. Tal funcionalidade atualmente já está presente neste trabalho, e será descrita e analisada com mais detalhes ainda nesta seção.

Seguindo nosso objetivo de manter interfaces pequenas e simples, criamos algumas facetas que apresentam pequenos conjuntos de métodos, fáceis de utilizar mas ainda assim expressivos. Como esta tarefa não é sempre fácil, nos deparamos com algumas dificuldades, que serão descritas mais adiante nas subseções apropriadas deste capítulo.

Assim como os nós de execução, todo contêiner tem um conjunto de propriedades inicial, que é padronizado e com algumas propriedades obrigatórias. No entanto, contam com uma faceta (as facetas serão descritas mais adiante nesta mesma seção) que permite que propriedades sejam adicionadas e/ou alteradas em tempo de execução. Estas propriedades podem também ser definidas como somente-leitura. Segue uma lista das propriedades obrigatórias e seus significados:

- *platform*: Informa a plataforma ou sistema operacional no qual o contêiner está funcionando. Atualmente são suportadas as plataformas Windows e UNIX, com valores "WIN", "WINDOWS", "WINNT", "WINDOWSNT", "WINDOWS_NT", "LINUX"ou "UNIX". Esta propriedade é definida como somente-leitura.
- *enIOR*: Referência para a faceta IComponent do Nó de Execução no qual o contêiner deve se registrar, na forma de um arquivo IOR. Esta propriedade é definida como somente-leitura.

Segue uma lista de propriedades opcionais que podem ser utilizadas pelo próprio contêiner:

- *host*: Nome ou endereço IP da máquina hospedeira do serviço na rede. Esta propriedade é definida como somente-leitura.
- *port*: Porta desejada para o processo do contêiner. Esta propriedade é definida como somente-leitura.
- *initfile*: Propriedade específica da implementação Lua, que informa o nome de um arquivo Lua a ser carregado e executado durante a chamada ao método *startup*(). Este arquivo pode ser utilizado para configurações adicionais pós-inicialização, por exemplo. Esta propriedade é definida como somente-leitura.
- *use_core_services*: Indica se outros serviços OpenBus, como o Serviço de Controle de Acesso e o Serviço de Registro, devem ser utilizados ou não. Valores: "true"ou "false". Caso esta propriedade não seja especificada, ela será criada e terá atribuída a si o valor falso. Esta propriedade é definida como somente-leitura.
- *accessControlServerHostName*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o nome ou endereço IP da máquina hospedeira do Serviço de Controle de Acesso do barramento OpenBus. Esta propriedade é definida como somente-leitura.

– *accessControlServerHostPort*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa a porta utilizada pelo processo do Serviço de Controle de Acesso do barramento OpenBus. Esta propriedade é definida como somente-leitura.

– *privateKeyFile*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o caminho, incluindo o nome do arquivo, para a chave de acesso ao barramento OpenBus. Esta propriedade é definida como somente-leitura.

– *accessControlServiceCertificateFile*: Propriedade obrigatória somente se "use_core_services"for verdadeira. Informa o caminho, incluindo o nome do arquivo, para o certificado a ser utilizado no barramento OpenBus. Esta propriedade é definida como somente-leitura.

4.2.1
Interação com o OpenBus

Aos contêineres é permitido trabalhar em conjunto com alguns dos serviços já existentes do barramento OpenBus, como o Serviço de Controle de Acesso e o Serviço de Registro, de forma opcional. No entanto, caso seu Nó de Execução pai esteja participando de um barramento, o contêiner deve obrigatoriamente participar também.

4.2.2
Argumentos

Nossa implementação de contêineres Lua aceita dois argumentos, opcionais, utilizados pelos nós de execução:

– *-L*: Carrega um arquivo na inicialização, com propósito de configuração. Isto funciona de forma ligeiramente diferente da propriedade "initfile", pois o arquivo lá definido só é executado na chamada ao método startup().

– *-O*: Parâmetro que define o uso de outros serviços do barramento OpenBus, da mesma forma que a propriedade "use_core_services". Os parâmetros são avaliados após a leitura de propriedades, e assim sendo este tem prioridade sobre a propriedade "use_core_services". O Nó de

Execução, quando participa de um barramento OpenBus, se utiliza deste parâmetro ao criar um contêiner.

4.2.3 Interceptadores

Interceptadores são ferramentas extremamente úteis, principalmente em componentes distribuídos, pois não só podem ser utilizados para a implementação de aspectos relativos à aplicação, como também para inserir pequenas novas funcionalidades ao longo da evolução da mesma. O suporte à interceptação pode ser provido em diferentes níveis, como no contêiner como um todo ou em cada faceta de cada componente, inclusive as do próprio contêiner. Em nossa infra-estrutura, optamos por fornecer interceptadores ao nível do contêiner. A maior vantagem de se fornecer a interceptação neste nível é que pode-se utilizá-los para realizar tarefas globais, referentes a todos os componentes instanciados neste contêiner. É natural que componentes instanciados em um mesmo contêiner sejam fortemente acoplados funcionalmente, o que também colabora para que este nível seja o mais adequado para nossa arquitetura. Um exemplo é o uso do próprio Serviço de Controle de Acesso do OpenBus, descrito na Seção 3.3.1. Nossa infra-estrutura fornece um interceptador que cuida das validações de credenciais que precisa ser instalado apenas uma vez no contêiner, realizando esse tratamento para todas as comunicações de todos os componentes internos.

Interceptadores devem, como todo o resto da arquitetura, ser componentes SCS. Isto garante que tenham acesso a todas as facilidades da infra-estrutura, permitindo uma alta flexibilidade em termos de configuração, conexão e adaptação, como qualquer outro componente. Da mesma forma, interceptadores mais simples podem ser desenvolvidos, com apenas a faceta IComponent sendo implementada.

O ORB OiL permite a instalação de apenas um interceptador servidor, para chamadas recebidas, e um interceptador cliente, para chamadas enviadas. Fica a cargo da aplicação que o utilize dar o suporte a mais de um interceptador, caso necessite. Desta forma, desenvolvemos um gerenciador de interceptadores, que é instalado no ORB tanto como interceptador cliente como interceptador servidor, e se encarrega de lançar o método apropriado em cada interceptador cadastrado. Este gerenciador não é um componente, mas sim parte do Contêiner de Componentes e é chamado sempre que uma chamada é

feita ou recebida.

São mantidas duas filas pelo gerenciador, uma para cada tipo de interceptador (cliente ou servidor), e ambas são gerenciadas de forma robusta, impedindo erros derivados de acesso concorrente. Ambas as filas são ordenadas, para que certos interceptadores possam ter precedência sobre outros. Essa ordem pode ser modificada em tempo de execução através da faceta ComponentInterception de um contêiner(descrita na Seção 4.2.5). Ao receber uma chamada para a modificação da ordenação, o gerenciador espera que a fila apropriada esteja disponível para escrita, podendo lançar uma exceção após um tempo determinado pelas propriedades do contêiner. Ao obter o acesso à lista, esta fica travada para leitura e escrita até que o procedimento termine. Os acessos concorrentes serão mantidos em espera, podendo também receber exceções caso o tempo determinado nas propriedades seja excedido.

A faceta ComponentInterception permite também a carga e remoção de interceptadores. A carga de interceptadores basicamente consiste da carga normal de um componente (através da faceta ComponentLoader) seguida do registro no gerenciador, e um processo análogo é feito para a remoção.

Uma dificuldade que encontramos durante o desenvolvimento dos interceptadores foi a otimização do uso de referências locais, conhecida como *collocation*. Esta facilidade é provida automaticamente pelo ORB OiL, fazendo com que chamadas locais não precisem ser sobrecarregadas com os protocolos CORBA, e previamente era utilizada pelo contêiner para melhorar o desempenho de chamadas locais. Ao implementarmos interceptadores, percebemos que estes não estavam sendo utilizados em chamadas locais, pelo fato de o gerenciador de interceptadores ser um interceptador CORBA. Este problema ainda está sem uma boa solução, pois nossa solução inicial foi desabilitar completamente as otimizações de *collocation*. Como estamos em um ambiente distribuído, o tempo necessário para o tráfego de informações na rede em geral tende a diminuir a importância desta perda de desempenho, mas para aplicações centradas em apenas um contêiner, com pesadas comunicações entre seus componentes, isto torna-se um problema mais grave. Essas otimizações de *collocation* geraram alguns outros pequenos problemas, como o fato de *proxies* locais não conterem informações sobre as interfaces/facetas sendo chamadas, nem o método *_narrow*(). Uma solução alternativa é manter um envólucro para o objeto local com parte das informações CORBA, mesmo em *proxies* locais.

Um outro problema mais simples que ainda enfrentamos é o fato de interceptadores poderem sobrescrever o contexto das requisições, removendo

dados importantes para outros interceptadores. Para evitar isto, pode-se fornecer uma interface interna para a manipulação de contextos, que evite esse tipo de interferência entre interceptadores.

O contêiner conta com um interceptador de sistema, que é o interceptador relativo ao Serviço de Controle de Acesso do OpenBus. Este interceptador é automaticamente instalado se a propriedade *use_core_services* estiver definida na inicialização do contêiner, e se encarrega de inserir a credencial em chamadas sendo feitas, ou conferir a validade de credenciais contidas em chamadas recebidas. Caso a credencial seja inválida, a chamada é bloqueada. Este interceptador é sempre o primeiro a ser executado.

Cada interceptador deve implementar quatro funções, seguindo a especificação do ORB OiL:

- *sendrequest*: Esta função é lançada sempre que uma chamada externa for feita.
- *sendreply*: Esta função é lançada ao término de uma requisição recebida.
- *receiverequest*: Esta função é lançada quando uma requisição é recebida, antes de ser atendida.
- *receivereply*: Esta função é lançada sempre que uma resposta a uma chamada externa for recebida.

Apesar de não seguirmos exatamente as interfaces definidas pelo padrão CORBA para interceptadores (*Portable Interceptors*), adotamos essencialmente a mesma semântica definida pelo padrão. A principal exceção é que fornecemos a possibilidade de inserção e remoção dinâmica de interceptadores.

4.2.4 Suspensão

Todo sistema, por mais estável que seja, em algum momento de sua execução pode necessitar passar por alguma manutenção. Em certos casos, esta manutenção pode exigir algum tempo de inatividade. Este tempo, por sua vez, pode ser breve ou extenso.

Para dar suporte à possibilidade de realizar algumas tarefas de manutenção sem interromper o sistema, desenvolvemos uma faceta para o contêiner chamada ComponentSuspension (descrita na Seção 4.2.5). Ela permite que

toda a comunicação externa relativa aos componentes de um contêiner seja interrompida, e posteriormente reativada.

Ao se utilizar a faceta de suspensão, pode-se escolher entre três estados para o contêiner: normal, suspenso ou parado. O estado normal representa o estado de execução natural do contêiner. O estado suspenso, obtido através do método *suspend*(), é utilizado em pequenas pausas, como por exemplo pequenas reconfigurações de dependências internas ou externas. Nesse estado, todas as chamadas externas recebidas ou feitas por componentes são retidas e ficam aguardando liberação. Já o estado parado, obtido através do método *halt*(), é utilizado para modificações mais demoradas. Nesse estado, todas as chamadas recebidas e feitas são rejeitadas, recebendo uma exceção. Em todos os estados, os comportamentos respectivos afetam também quaisquer interceptadores OpenBus instalados. Há, ainda, um método para a consulta do estado corrente.

É necessário notar que até mesmo as chamadas entre componentes pertencentes a um mesmo contêiner devem ser bloqueadas. No entanto, nenhuma comunicação com o próprio contêiner deve sofrer bloqueio, pois o mesmo deve poder continuar exercendo todas as suas funções, principalmente as relativas à faceta ComponentSuspension. Além disto, respostas a requisições também não devem ser bloqueadas, pois podem haver requisições já em andamento no momento em que a suspensão for iniciada. Por fim, certos interceptadores de sistema também não devem ser bloqueados, como o interceptador relativo ao Serviço de Controle de Acesso OpenBus por exemplo, pois a credencial do contêiner deve ser renovada periodicamente.

Existem diversas formas de se implementar a funcionalidade de suspensão, sendo que cada uma levaria a resultados diferentes. Uma alternativa consiste em separar o contêiner de seus componentes através de POAs diferentes. Assim, seria simples bloquear as comunicações dos componentes enquanto as comunicações do contêiner seriam mantidas ativas. Já uma implementação através do uso de um interceptador consistiria em implementar os tratamentos necessários para a suspensão no interceptador CORBA que seria instalado no ORB. Deve-se notar que independente da solução adotada, ainda existiriam quaisquer outros interceptadores instalados no sistema, utilizando o mesmo POA que fosse utilizado para os componentes.

A princípio a opção sem interceptadores é mais simples e mais eficiente, porém ela levaria a alguns problemas de difícil solução. Por exemplo, ao suspender as comunicações de todo o POA dos componentes, não só suspenderíamos

a comunicação desses mas também a de todos os interceptadores instalados. Não teríamos acesso a um controle fino, necessário para permitir a execução de interceptadores que não devem ser bloqueados, como o responsável pela renovação da credencial no Serviço de Controle de Acesso OpenBus. Para isso, teríamos que ter um tratamento especial para cada caso, o que negaria a vantagem.

Com a escolha da implementação via interceptadores, nos deparamos com a necessidade de resolver os problemas de liberação das chamadas relacionadas ao próprio contêiner e a interceptadores de sistema. É fácil reconhecer o *servant* requisitado em chamadas recebidas; no entanto, ao interceptar uma chamada externa a ser realizada fica difícil descobrir quem está fazendo o pedido. Criamos então uma forma de reconhecer o *servant* que estava realizando a chamada. Isso resolveu o problema para chamadas derivadas de outras chamadas recebidas como é o caso de chamadas relacionadas às facetas do contêiner. No entanto, ainda não nos seria possível reconhecer chamadas realizadas a partir de *threads*[1] independentes, como a *thread* do interceptador que realiza a renovação da credencial no Serviço de Controle de Acesso OpenBus. Tivemos então que encontrar também uma solução para este problema, que consistiu em identificar as threads internas . Ambas as soluções foram possíveis devido ao ORB OiL, pois este é responsável também pelas *threads* em funcionamento.

Inicialmente o código da suspensão ficava separado em interceptadores de suspensão mas, devido à necessidade destes tratamentos, resolvemos integrar suas funcionalidades ao gerenciador de interceptadores. Isto efetivamente acabou com os interceptadores OpenBus relacionados à suspensão, mas ainda manteve a solução como um interceptador CORBA, que é o que o gerenciador de interceptadores é. Dada a relação intrínseca entre ambas as funcionalidades de gerenciamento de interceptadores e suspensão, isto fez muito sentido, e aumentou a eficácia do sistema. Algumas poucas situações específicas que existiam inicialmente, nas quais a suspensão poderia não funcionar da forma intencionada, foram impedidas. Por exemplo, caso a suspensão continuasse dependente de um interceptador OpenBus, medidas deveriam ser tomadas para que sua posição nas filas não fosse modificada. A adição ou remoção de interceptadores também seria complicada, dependendo de haver chamadas retidas ou não, pois caso um interceptador fosse executado ao receber uma chamada, não poderia ser removido ao menos até que a resposta fosse enviada. Com a

[1]Na verdade, estas são co-rotinas Lua gerenciadas pelo OiL. Co-rotinas Lua são o mesmo que *threads* cooperativas.

integração, ficou mais fácil restringir mudanças no estado dos interceptadores dependendo do estado de suspensão do contêiner. O maior controle sobre as interações entre chamadas interceptadas, sobre o controle das filas de interceptadores e sobre como a suspensão deve funcionar em cada caso, tornou a implementação bastante robusta.

4.2.5
Facetas

No Apêndice B.3 estão listadas as facetas do contêiner, com explicações sobre cada método. Não são listadas todas as facetas do modelo de componentes básico, pois funcionam da forma padrão e já foram mencionadas na seção 3.2. A única exceção é a faceta IComponent, pois esta teve métodos sobrescritos. Somente os métodos sobrescritos serão mencionados. O contêiner provê todas as facetas básicas, IComponent, IReceptacles e IMetaInterface.

O Apêndice A inclui todas as operações das facetas exclusivas ao contêiner no módulo *container*, além de outras comuns a outros serviços no módulo *auxiliar*, disponível também no Apêndice A.

4.2.6
Receptáculos

Aqui serão listados os receptáculos do contêiner, com breves explicações sobre seus propósitos. Cada nome remete ao nome de uma faceta ao qual se conecta, e não de um componente.

- Receptáculo *ContainerManager*: Este receptáculo guarda a referência para a faceta "ContainerManager"do Nó de Execução. Em sua inicialização, o contêiner obtém esta faceta a partir da faceta IComponent que recebe no formato IOR, para realizar seu registro no nó. A faceta "ContainerManager"é utilizada novamente no método shutdown(), por exemplo, quando o contêiner desfaz seu registro.
- Receptáculo-Lista *ComponentRepository*: Este receptáculo é múltiplo, pois o contêiner suporta a utilização de múltiplos repositórios. Ao receber um pedido de carga de componente que não exista em sua *cache* interna,

o contêiner itera sobre esta lista, procurando o componente em todos os repositórios conhecidos através de suas facetas "ComponentRepository".

– Receptáculo-Lista *ComponentHelpRecept*: Este receptáculo geralmente guarda o mesmo número de referências que o anterior, pois normalmente são referências à faceta "ComponentHelp"dos mesmos repositórios. Podem haver também referências a facetas "ComponentHelp"de outros contêineres, ou de outros componentes que a ofereçam.

4.2.7

Exemplo de Uso do Contêiner

Para ilustrar o uso mais comum de um contêiner, apresentaremos aqui um pequeno trecho de código. As chamadas serão todas síncronas e sem tratamento de exceções por motivo de clareza.

No exemplo abaixo, assumiremos que já existe uma variável CC referente à faceta IComponent do contêiner. O código para a obtenção desta referência se encontra na listagem 4.1. Assumiremos também que o contêiner já tem os componentes carregados em sua *cache* local, ou conhece repositórios que os forneçam.

Listing 4.2: Exemplo de uso de um contêiner

```
1  -- De posse da faceta IComponent, podemos obter as outras
       facetas
2  clFacet = CC:getFacet("scs::container::ComponentLoader"):
       _narrow()
3
4  -- Carga de uma instância do componente PingPong
5  componentId = { name = "PingPong", version = 1.0,
       platform_spec = "" }
6  arg = {"1"}
7  ppHandle = clFacet:load(componentId, arg)
8  ppHandle.cmp:startup()
9
10 -- Por fim, pediremos ao contêiner que termine seu processo
       de forma organizada.
11 CC:shutdown()
```

4.3
Repositório de Componentes

Durante o desenvolvimento da infra-estrutura de execução para o *middleware* OpenBus, criamos um componente para atuar como um repositório de componentes simples, chamado ComponentRepository. Este repositório é um componente de aplicação comum, e não é obrigatório em nossa arquitetura. No entanto, não há atualmente na infra-estrutura uma forma de se instalar em tempo de execução definições de componentes (o que inclui o código-fonte e binários) diretamente em um contêiner, o que torna o uso de ao menos um repositório quase inevitável. O propósito desta versão inicial do repositório é justamente simplificar o processo de instalação de aplicações distribuídas, ao diminuir as responsabilidades do contêiner e permitir a separação destas funções que não necessitam existir em todos os contêineres de um ambiente distribuído.

Repositórios permitem a instalação, desinstalação e obtenção de descrições, textos de ajuda e códigos-fonte e binários de um componente. Atualmente o OpenBus está preparado para trabalhar com componentes implementados nas linguagens Lua, C++ e Java, mas o repositório não faz distinção e pode aceitar normalmente a instalação de componentes em qualquer linguagem, que podem vir a ser suportadas por futuras implementações de contêiner.

Apenas um arquivo é aceito na instalação de um componente; assim, pode ser fornecida uma biblioteca dinâmica para C++, um arquivo *class* Java, ou um arquivo Lua comum. Para componentes que exijam múltiplos arquivos, podem ser usados arquivos-contêiner como *zips*, *rars*, *jars*, etc. A implementação atual do contêiner Lua também suporta *zips* para o empacotamento de diversos arquivos Lua.

Assim como os nós de execução e contêineres, o repositório tem um conjunto de propriedades. No entanto, por ser um componente comum e não contar com seu processo próprio, não precisa definir propriedades como *host* e *port*, por exemplo. Assim, se resume a apenas uma propriedade inicial, que além disto é opcional:

- *initfile*: Informa o nome de um arquivo Lua a ser carregado e executado durante a chamada ao método *startup*(). Este arquivo pode ser utilizado para configurações adicionais pós-inicialização, por exemplo.

4.3.1
Facetas

No Apêndice B.4 estão listadas as facetas do repositório, com explicações sobre cada método. Não são listadas todas as facetas do modelo de componentes básico, pois funcionam da forma padrão e já foram mencionadas na seção 3.2. A única exceção é a faceta IComponent, pois esta teve métodos sobrescritos. Somente os métodos sobrescritos serão mencionados. O repositório provê todas as facetas básicas, IComponent, IReceptacles e IMetaInterface.

O Apêndice A inclui todas as operações das facetas exclusivas ao repositório no módulo *repository*. Facetas comuns a outros serviços estão no módulo *auxiliar*, disponível também no Apêndice A.

4.3.2
Exemplo de Uso do Repositório

Para ilustrar o uso mais comum de um repositório, apresentaremos aqui um pequeno trecho de código. As chamadas serão todas síncronas e sem tratamento de exceções por motivo de clareza.

No exemplo abaixo, assumiremos que já existe uma variável CC referente à faceta IComponent do contêiner. O código para a obtenção desta referência se encontra na listagem 4.1. Mostraremos então os passos necessários para se carregar um repositório, conectá-lo ao contêiner e instalar um componente. Note que apenas um repositório é suficiente para diversos contêineres, mesmo em máquinas diferentes. Basta criá-lo em um contêiner e conectá-lo a todos os contêineres desejados.

Listing 4.3: Exemplo de uso de um repositório

```
1  -- De posse da faceta IComponent do contêiner, podemos obter
       as outras facetas
2  clFacet = CC:getFacet("scs::container::ComponentLoader"):
       _narrow()
3
4  -- Carga e inicialização de uma instância do componente
       ComponentRepository
5  componentId = { name = "ComponentRepository", version = 1.0,
       platform_spec = "" }
```

```
6   cpnRepoHandle = clFacet:load(componentId, {})
7   CR = cpnRepoHandle.cmp
8   CR:startup()
9
10  -- Obtenção das facetas ComponentRepository e ComponentHelp
11  crFacet = CR:getFacet("IDL:scs::repository::
        ComponentRepository"):_narrow()
12  helpFacet = CR:getFacet("IDL:scs::auxiliar::ComponentHelp"):
        _narrow()
13
14  -- Conexão do repositório aos receptáculos do container
        chamados ComponentRepository e ComponentHelpRecept
15  recepFacet = CC:getFacetByName("IReceptacles"):_narrow()
16  conId = recepFacet:connect("ComponentRepository", crFacet)
17  helpConId = recepFacet:connect("ComponentHelpRecept",
        helpFacet)
18
19  -- Instalação do componente PingPong
20  componentId = { name = "PingPong", version = 1.0,
        platform_spec = "" }
21  entry_point = "PingPong1.PingPong" -- {Name+Version}.Name
22  shared = true
23  ppFile = io.open("PingPong.lua", "r"):read("*a")
24  ppHelp_info = "This component pings and pongs!"
25  extension = "lua"
26
27  crFacet:install(componentId, entry_point, shared, ppFile,
        ppHelp_info, extension)
```