4
Instalação das Dependências Estáticas por Sistemas de Pacotes

Este capítulo apresenta uma solução para tratar dependências estáticas por linguagem e plataforma em componentes SCS. Na Seção 4.1 apresentamos alguns requisitos encontrados na literatura e outros inerentes ao cenário distribuído, multi-plataforma e multi-linguagem. Com base nesse estudo e na experiência prática, escolhemos um sistema de pacotes pré-existente para ser reusado e integrado ao SCS. Os principais recursos desse sistema são apresentados na Seção 4.2. Algumas modificações sobre esse sistema de pacotes ainda foram necessárias e são apresentadas na Seção 4.3. A integração desse sistema à infraestrutura de execução do SCS é detalhada na Seção 4.4 e, por fim, destacamos algumas observações finais na Seção 4.5.

4.1
Motivação e Requisitos

Através do amplo estudo apresentado no Capítulo 2 evidencia-se que a maioria das abordagens para componentes de software ou não suportam o conceito ou não possuem procedimentos padronizados para a instalação de dependências estáticas. Essa evidência é preocupante, pois não é prático sempre optar por dependências paramétricas [3].

Uma análise mais profunda sobre o suporte a dependências estáticas indica uma marginalização do processo de pós-desenvolvimento [47]. Dessa forma não estabelece-se como os desenvolvedores devem publicar seus componentes nem como os usuários devem obtê-los para então implantá-los. Geralmente, ou o usuário responsabiliza-se por localizar e obter todos os componentes necessários ou o desenvolvedor precisa publicar seus componentes de forma auto-contida, incluindo todas as dependências. Ambas alternativas mostram-se exaustivas e propensas a erros.

Hoek et al. definem esse problema como *Software Release Management* e entendem que ele é parte do processo de implantação [48]. Esses autores também desenvolveram uma arquitetura para tratar esse problema em cenários distribuídos [21] e revisaram os requisitos relacionados a esse tipo de sistema no

contexto das tecnologias baseadas em componentes de software [47]. A seguir destacamos alguns desses requisitos que contemplam os pontos de vista do desenvolvedor e do usuário[1]:

1. As dependências entre os componentes deveriam ser explícitas e facilmente registradas. É fundamental que o desenvolvedor seja capaz de documentar as dependências durante o processo de liberação (do Inglês, *release*). Uma vez registradas, essas dependências devem estar disponíveis para as ferramentas que automatizam seu tratamento.

2. As versões liberadas (do Inglês, *releases*) devem se manter consistentes. Para o desenvolvedor, em geral, não deve ser possível remover componentes que sejam dependentes de outros componentes. De forma similar, uma nova versão de um componente não deveria, necessariamente, causar a revogação de uma versão antiga pois ambas versões podem não ser funcionalmente compatíveis, precisando coexistir.

3. O escopo de uma liberação deve ser controlável. Um desenvolvedor deveria ser capaz de controlar quais usuários podem obter um componente. No mínimo, políticas de licenciamento ou controle de acesso deveriam ser providos.

4. O processo de liberação deve demandar esforço mínimo do desenvolvedor. Quando uma nova versão for lançada, o desenvolvedor deveria poder reusar as descrições anteriores para especificar apenas o que mudou.

5. As descrições dos componentes devem estar disponíveis. Baseado nos metadados, o usuário deve ser capaz de decidir quais componentes lhe interessam.

6. Deve-se prover transparência de localização. O usuário deve ter uma forma simples de obter componentes, sem precisar saber onde os componentes estão fisicamente armazenados nem de onde serão obtidos.

7. Um componente e suas dependências deveriam ser obtidos através de uma operação única. É comum ocorrer inconsistências e erros quando um usuário obtém um componente e suas dependências um por um, portanto o sistema deve permitir ao usuário obter o componente e suas dependências em um passo único.

[1]Nesse parágrafo, o termo *usuário* refere-se ao ator da instalação e não necessariamente ao usuário final, pois a instalação pode ser mediada por uma ferramenta de implantação.

Em paralelo aos estudos de Hoek et al., a popularização de sistemas operacionais baseados em GNU/Linux também ajudou a disseminar entre desenvolvedores e administradores o conceito de *sistemas de pacotes* (do Inglês, *Package Management Systems*). Entende-se por sistemas de pacotes um conjunto de descritores, ferramentas, formatos de arquivos e políticas de empacotamento cujo uso aplica-se às diversas etapas do processo de implantação como: liberação, instalação, atualização, remoção e revogação.

Atualmente, a maioria dos (senão todos) sistemas operacionais desse gênero usam algum sistema de pacotes para enfrentar os problemas observados por Hoek et al. Entre os mais populares estão o DEB [49] e RPM [50], que geralmente são usados em conjunto com meta-instaladores[2] que automaticamente resolvem, obtêm e instalam as dependências entre os diferentes artefatos. A maioria desses sistemas de pacotes adotam abordagens heurísticas baseadas na experiência de desenvolvedores e administradores para resolver as dependências entre os pacotes, uma vez que pesquisas recentes mostram que a correta resolução de dependências é um problema em aberto, conhecido como instalabilidade (do Inglês, *installability*) e redutível ao problema da satisfatibilidade (SAT) [5].

A partir dos requisitos existentes na literatura [47] e da observação da prática em adotar os sistemas de pacotes, este trabalho propõe o reuso de um sistema de pacotes para padronizar o processo de pós-desenvolvimento e sistematizar a instalação de componentes com dependências estáticas. Considerando, porém, que este trabalho destina-se a prover soluções para cenários distribuídos, multi-plataforma e multi-linguagem torna-se necessário ampliar os requisitos para permitir uma escolha tecnológica mais adequada:

8. Oferecer portabilidade para diversas plataformas (ao menos em Unix e Windows). Não basta, entretanto, que as ferramentas do sistema de pacotes operem em diferentes plataformas. É fundamental permitir a descrição de metadados e procedimentos de forma diferenciada por plataforma.

9. Permitir o uso de repositórios remotos de implementações.

10. Permitir o empacotamento de aplicações desenvolvidos em diversas linguagens.

11. Facilitar o reuso e a extensão do sistema de pacotes.

[2]O termo refere-se a utilitários que usam os sistemas de pacotes para melhorar a usabilidade, como o APT (`http://www.debian.org/doc/manuals/apt-howto`) e YUM (`http://yum.baseurl.org`).

4.2 Reuso do Sistema de Pacotes LuaRocks

Ao longo das nossas pesquisas investigamos os sistemas de pacotes mais populares em plataformas GNU/Linux como DEB[49], RPM[50], ZEROINSTALL[51] e NIX[52]. A principal dificuldade em reusar uma dessas alternativas deve-se à incapacidade de suas ferramentas e utilitários serem portados para sistemas operacionais incompatíveis com a especificação POSIX[53] (como o Microsoft Windows). Este trabalho não considerou o uso de emuladores do subsistema POSIX como Cygwin[54] pois sua aplicabilidade é limitada, não interopera com aplicações nativas e dificulta a administração das instalações.

Ao mesmo tempo, a popularidade de linguagens dinâmicas como Perl, Ruby e Lua [45] tem motivado o uso de sistemas de pacotes específicos por linguagem, mas portáveis para diferentes sistemas operacionais. Alguns exemplos são CPAN[55], RUBYGEMS[56] e LUAROCKS[57], respectivamente.

Diante desse cenário, este trabalho escolheu o sistema LUAROCKS, que é desenvolvido totalmente em Lua. Os recursos disponíveis no LUAROCKS cumprem os requisitos apresentados na Seção 4.1, conforme é apresentado nas subseções a seguir. A única exceção é sua falta de suporte ao empacotamento de artefatos desenvolvidos em diferentes linguagens. Nesse sentido, realizamos as modificações necessárias no LUAROCKS para oferecer esse tipo de suporte, tornando-o multi-linguagem, conforme é apresentado na Seção 4.3. Após essas modificações, integramos o LUAROCKS à infraestrutura de execução para componentes SCS, conforme é explicado na Seção 4.4.

4.2.1 Descrições de Pacotes, Dependências e Portabilidade

Os pacotes LUAROCKS possuem um arquivo de descrição através do qual o desenvolvedor especifica: metadados, dependências e procedimentos de compilação e instalação. O descritor que agrupa essas informações é conhecido como *rockspec* [58] e é escrito na linguagem Lua. Um exemplo ilustrativo de um descritor de pacote é apresentado no Código 4.1.

Um empacotador pode especificar dois tipos de dependências: (i) de outros pacotes usando a tabela *dependencies*; (ii) ou de artefatos pré-instalados usando a tabela *external_dependencies*. Esse último tipo é útil em situações que seja necessário verificar a existência de determinados artefatos (como arquivos de cabeçalho e bibliotecas dinâmicas) que não possam ser obtidos através de outros pacotes LUAROCKS.

Os procedimentos de compilação e instalação são definidos na tabela *build*. O LUAROCKS suporta diferentes tipos de procedimentos: (i) *module*,

Código 4.1: Exemplo de pacote descrito no formato *Rockspec* do LuaRocks

```
1  package = "HelloChecked"
2  version = "1.0-0"
3  source = {   url = "http://myhost.com/hellochecked-src-1.0.zip"   }
4  description = {
5    summary   = "Componente HelloChecked",
6    detailed = "Exemplo ilustrativo que retorna um hash MD5 de uma mensagem.",
7    license = "GPL",
8    homepage = "http://myhost.com/hellochecked",
9  }
10 dependencies = { "lua >= 5.1", "libmd5 >= 1.2" }
11 external_dependencies = {
12   DATE = { headers = { "date.h" }, libraries = { "date" } },
13 }
14 build = {
15   type = "make",
16   platforms = {
17     linux   = { build_variables={ LIB_OPTION="-shared", LIB_EXT=".so"} },
18     macosx = { build_variables={ LIB_OPTION="-bundle", LIB_EXT=".dylib" } },
19     win32   = { build_variables={ LIB_OPTION="/DLL /WX",LIB_EXT=".dll" } },
20   }
21 }
```

mais indicado para módulos *pure*-Lua; (ii) *make*, para uso das regras e variáveis de um Makefile; (iii) *cmake*, para uso das regras e variáveis do CMake; e (iv) *command*, para invocar algum comando de sistema para compilar e instalar. Cada tipo assume seu próprio conjunto de campos a ser preenchidos.

Todos os campos do descritor possuem validadores sintáticos, portanto quando um empacotador fornece seus descritores para os utilitários do LuaRocks, será realizada uma validação sintática antes de qualquer outra ação. Além disso, o usuário pode consultar todas as descrições, com exceção dos procedimentos, através da opção *search* no utilitário do LuaRocks.

A fim de ser portável, o LuaRocks é totalmente desenvolvido em Lua e permite a descrição de campos específicos por plataformas através do uso da tabela *platforms* no descritor. No exemplo do Código 4.1 essa tabela é usada dentro da tabela *build* para indicar variáveis de compilação diferenciadas para Linux, Windows e MacOSX. Os campos existentes em *platforms.linux*, *platforms.macosx* e *platforms.win32* são usados em substituição aos campos de mesmo nome presentes na tabela que contém a tabela *platforms*, no caso a *build*. Os campos específicos por plataforma são validados usando o mesmo validador sintático associado à tabela que contém a *platforms*. Portanto, esse mecanismo é genérico e pode ser usado em conjunto com tabelas *build, source, dependencies e external_dependencies*.

Assim, entendemos que o LuaRocks cumpre os requisitos 1, 5 e 8.

4.2.2 Empacotamento, Repositórios, Transparência de localização e Procedimento de Obtenção

Após criar os descritores no formato *rockspec*, o empacotador pode usar a opção *pack* ou *make* no utilitário do LuaRocks para criar um pacote válido, conhecido como *rock*. No LuaRocks, um pacote é um arquivo comprimido contendo o descritor e os artefatos binários (executáveis, bibliotecas, documentações) ou fontes. Caso o empacotador deseje evoluir uma versão de um pacote ou mesmo criar um novo, é possível reusar um descritor anterior, alterar os campos relevantes e então usar novamente o utilitário do LuaRocks para criar um novo pacote.

A partir de um conjunto de pacotes é possível criar um catálogo dos pacotes, conhecido como arquivo de manifesto (do Inglês, *manifest*). Esse catálogo agrupa nome, versão e uma identificação da plataforma sobre todos os pacotes. Para se ter um *repositório de pacotes*, conhecido também como *rock server*, basta prover uma forma de acesso ao arquivo do catálogo e aos pacotes.

O método mais comum de acesso remoto aos repositórios é através dos protocolos HTTP ou FTP. Por exemplo, considerando que um servidor HTTP abrigue um repositório na URL *http://myhost.com/*, o manifesto deverá estar acessível por *http://myhost.com/manifest* e o pacote ilustrado no Código 4.1 deverá estar acessível por *http://myhost.com/hellochecked-1.0-0.src.rock*[3]. Caso o repositório não esteja remoto, pode-se identificá-lo com URL's locais, a exemplo de *file:///caminho/para/pasta/*.

Para obter pacotes armazenados em repositórios só é preciso conhecer as URL's dos repositórios. O administrador (ou usuário) pode configurar um conjunto próprio de URL's no LuaRocks[4]. Quando solicita-se a instalação de um pacote através do seu nome e versão, o LuaRocks inicia uma busca em todos os repositórios. Essa busca consiste em localizar nos catálogos pacotes com nome e versão compatíveis. Caso o pacote possua dependências de outros pacotes, o LuaRocks tentará obter suas dependências subsequentes antes de proceder à instalação do pacote solicitado. Se alguma instalação ou obtenção de pacotes falhar, todo o processo é abortado sem alterar a árvore de instalação. Em particular, no LuaRocks a instalação consiste em extrair o pacote numa árvore de diretórios, conhecida como árvore de instalação ou *rocktree*.

Assim, entendemos que o LuaRocks cumpre os requisitos 4, 6, 7 e 9.

[3]O infixo *src* indica que é um pacote com fontes. Para pacotes contendo binários usa-se infixos que identificam a plataforma como *win32*, *linux-x86* e *macosx*.

[4]A instalação básica do LuaRocks usa a URL do repositório oficial do seu projeto.

4.2.3
Versionamento de Pacotes

No LuaRocks é possível manter pacotes em diferentes versões publicados no repositório (*rock server*), o que também é comum na maioria dos sistemas de pacotes, como DEB, RPM e NIX. Contudo, uma funcionalidade diferenciada é que o LuaRocks pode manter numa mesma árvore de instalação (*rocktree*) um pacote em diferentes versões sem comprometer a consistência. Para tanto, o LuaRocks mantém um catálogo dos pacotes instalados e suas dependências. Esse catálogo organiza a manutenção da árvore de instalações que possui subdiretórios para cada versão de um pacote.

A usabilidade desse recurso, originalmente, é possível porque o LuaRocks provê uma nova implementação do método de carga dos módulos Lua (instrução *require*) que permite a carga de um módulo em versão específica. Assim, quando um módulo é solicitado, busca-se no catálogo pelo módulo em questão e registra-se os caminhos para os locais de instalação de todas suas dependências. Isso possibilita que o carregador de módulos nativo de Lua possa carregar os módulos necessários corretamente. Com a integração do LuaRocks no SCS, esse tipo de uso passa a fazer parte da semântica do *Container* no SCS para permitir a carga de componentes versionados.

Assim, entendemos que o LuaRocks cumpre os requisitos 2 e 6.

4.2.4
Controle do Escopo

Assim como outros sistemas de pacotes como DEB, RPM ou NIX, o LuaRocks não possui nenhum recurso específico para controle de acesso. Entretanto, o LuaRocks é amplamente configurável e ajusta-se bem aos métodos convencionais como permissões de acesso a diretórios por usuário.

Um dos recursos mais interessantes no LuaRocks é sua capacidade em gerenciar diferentes árvores de instalações (*rocktree*). Isso é possível porque (i) todas as informações necessárias à resolução de dependências entre os pacotes instalados são catalogadas no arquivo de manifesto e (ii) todas as instalações registradas no catálogo estão em um mesmo diretório principal (*rocktree*). Para operar explicitamente sobre uma árvore ou outra, o LuaRocks permite o uso de um parâmetro com o caminho para o diretório de instalação e assim só realizará as instalações ou remoções sobre essa árvore, sem interferir nas instalações mantidas em outras árvores.

Essa capacidade é útil para facilitar o controle de escopo. Por exemplo, a instalação básica do LuaRocks já pré-configura duas árvores de instalações: (i) uma global para todo o sistema (tipicamente /usr/lib/luarocks); e (ii) outra

para cada usuário do sistema operacional (tipicamente /home/user/.luarocks). Assim as instalações ou remoções acontecem no diretório de cada usuário quando o usuário não tem permissão de alterar a árvore de instalação global.

Em relação ao controle de acesso aos pacotes no repositório, basta que o serviço de acesso remoto esteja configurado para só permitir o acesso às URL's mediante alguma política de acesso. Por exemplo, pode-se usar o campo *description.license* para restringir a obtenção de pacotes à prévia autenticação ou à localização das máquinas clientes.

Assim, entendemos que o LuaRocks cumpre o requisito 3.

4.2.5 Facilidade de Reuso e Extensão

O LuaRocks também é interessante pela sua modularidade. Todas as funcionalidades estão agrupadas em módulos Lua específicos que totalizam 22 arquivos e apenas 4350 linhas de código. Além disso, sua integração na infraestrutura de execução do SCS pode ser simples já que é programado em Lua.

Outras facilidades previstas para extensão são:

- Implementar novos tipos de procedimentos de compilação e instalação. Originalmente suporta os tipos: *module*, *make*, *cmake* e *command*. Para disponibilizar um novo tipo, basta criar um novo submódulo de *luarocks.build* que forneça uma função *run* para manipular o descritor e executar as tarefas necessárias.
- Implementar novos métodos de acesso remoto às URL's. Originalmente suporta os métodos: *http*, *ftp*, *sscm*[5], *git* e *cvs*. Para disponibilizar um novo tipo, basta criar um novo submódulo de *luarocks.fetch* que forneça uma função *run* para manipular a URL e obter os artefatos. Os métodos de acesso são usados para obter fontes, catálogos e pacotes.
- Definir novos campos e validadores do descritor editando o módulo *luarocks.type_check* que é usado na validação dos descritores.

Assim, entendemos que o LuaRocks cumpre o requisito 11.

[5] `http://www.seapine.com/surroundscm.html`

4.3 Modificações no LuaRocks para Suporte Multi-Linguagem

O único requisito que o LuaRocks originalmente não cumpre é o suporte ao empacotamento de aplicações desenvolvidas em outras linguagens além de Lua. Para tanto, este trabalho adicionou os seguintes campos no descritor e seus respectivos tratadores:

- Um campo opcional chamado *language* para indicar qual a linguagem usada e permitir um comportamento diferenciado durante a compilação, empacotamento e instalação. Os valores válidos são *java*, *lua* e *cpp*.
- A adição do campo *language* provocou a adição do campo *jars* para viabilizar que na linguagem Java informe-se uma lista de arquivos JAR a serem incluídos no pacote. Esses arquivos JAR podem, por exemplo, ser gerados por regras do utilitário Ant [59].

4.4 Incorporando o Sistema de Pacotes na Infraestrutura de Execução SCS

A partir da escolha do sistema de pacotes, identificamos as modificações necessárias sobre a infraestrutura de execução proposta por Augusto [42]. Incorporamos o LuaRocks no sistema de componentes SCS como um subsistema de empacotamento que define as regras para o empacotamento das implementações dos componentes e provê os utilitários para instalação, atualização e remoção de pacotes localmente em cada máquina.

Na primeira etapa da integração, criamos a noção de *pacote* para o sistema de componentes SCS. Um pacote de componente SCS é um pacote LuaRocks que inclui também um descritor sobre a implementação do componente. Contudo, essa novidade motivou uma revisão na forma de descrever componentes SCS. No trabalho de Augusto [42] não havia uma forma clara de descrever a estrutura lógica do componente nem mesmo os detalhes da implementação. Dessa forma, este trabalho define dois descritores:

1. **Modelo do componente**. O descritor do modelo possui nome, versão, facetas e receptáculos para um determinado tipo de componente. Esse descritor é especificado em OMG IDL pela estrutura conhecida como *ComponentTemplate*, apresentada no Código 4.2. O tipo do componente é unicamente identificado pela estrutura conhecida como *ComponentId* que serve para distinguir as estruturas lógicas dos componentes (*templates*).

2. **Implementação do componente**. O descritor da implementação completa as informações necessárias para distinguir as implementações e possibilitar a carga de uma implementação de um tipo de componente num

apropriado ambiente de execução. Esse descritor é especificado em OMG IDL pela estrutura conhecida como *ComponentDescription*, apresentada no Código 4.3. Esse descritor possui o identificador do tipo do componente, da linguagem e da plataforma e as localizações dos artefatos da fábrica de componentes (comentada na Seção 3.3) e das implementações das facetas e receptáculos.

Código 4.2: Definição do descritor do modelo do componente em OMG IDL

```
module scs {
module core {
 struct ComponentId {
   string name;
   octet major_version; octet minor_version; octet patch_version;
 }; };
module deployer {
 struct Facet      { string name; string interface_name; };
 struct Receptacle { string name; string interface_name;
                     boolean is_multiplex; };
 typedef sequence<Facet> FacetSeq;
 typedef sequence<Receptacle> ReceptacleSeq;

 struct ComponentTemplate {
   core::ComponentId id;
   FacetSeq facets;
   ReceptacleSeq receptacles;
 };
}; };
```

Código 4.3: Definição do descritor da implementação em OMG IDL

```
module scs {
module deployer {
  struct Artefact { string name; string class_name; };
  typedef sequence<Artefact> ArtefactSeq;

  struct ComponentDescription {
    core::ComponentId template_id;
    string lang;
    string plat;
    string entry_point;
    ArtefactSeq facets;
    ArtefactSeq receptacles;
  };
}; };
```

Esses descritores devem ser usados na implementação da fábrica de componentes para reconhecer as informações necessárias para instanciar um componente, como por exemplo, para reconhecer o artefato que implementa uma faceta e que será instanciado em memória. O Código 4.4 ilustra um trecho de código da fábrica de uma implementação Java do componente *HelloChecked*.

O desenvolvedor deve preencher esses dois descritores, pois serão usados no empacotamento e na carga do componente. O procedimento para criar um pacote SCS é o mesmo para um pacote LuaRocks e pode-se usar a opção *build* do utilitário de linha de comando do LuaRocks. Ao especificar um pacote,

Código 4.4: Fábrica de um componente implementado em Java usando os novos descritores

```
1  public class HelloCheckedFactory implements ComponentFactory {
2    // ...
3    public IComponent create(ComponentBuilder builder, String[] args)
4      throws LoadFailure {
5      String templateFilename =
6        new String("conf/hellochecked.template");
7      String descriptionFilename =
8        new String("conf/hellochecked.java.all.desc");
9      //objeto ComponentContext fornece acesso a estrutura do componente
10     ComponentContext context =
11       builder.newComponent(templateFilename, descriptionFilename);
12     System.out.println("New instance of: " + context.getComponentId());
13     return context.getFacetDescs().get("IComponent").facet_ref;
14   }
15   //...
16 }
```

conforme visto na Seção 4.2.1, o empacotador precisa indicar os arquivos de ambos descritores através da tabela *build.install*, como exemplificado em:

```
build = {
  install = {
    conf = { "hellochecked.template", "hellochecked.java.all.desc" }
  },
}
```

Uma vez criado o pacote do componente, o empacotador deve publicá-lo no componente *Repository* da infraestrutura de execução do SCS. A faceta *ComponentRepository* desse componente foi alterada. Conforme ilustrado no Código 4.5, agora existe suporte à estrutura *Package* e os métodos de publicação, de obtenção de pacotes e de consulta usam os novos descritores.

Na segunda e a mais importante etapa da integração, foi preciso uma revisão na forma de instalar os componentes SCS. No trabalho de Augusto [42] o componente responsável por essa atividade era o *Container*, abordagem que mostrou-se inadequada como comentado na Seção 3.3. Portanto, considerando que o componente *ExecutionNode* é o responsável pelo acesso aos recursos locais em cada máquina e que gerencia o ciclo de vida dos *Containers*, optamos por incluir no *ExecutionNode* o mecanismo de instalação de pacotes de componentes (reusando diretamente os módulos do LuaRocks). Criamos uma nova faceta chamada *Installer* e alteramos a configuração da infraestrutura de execução de forma que cada *Container* passa a delegar ao *ExecutionNode* (através do uso da faceta *Installer*) as atividades de manutenção das instalações. A faceta *Installer* permite instalar, atualizar, remover e buscar informações sobre as implementações dos componentes e suas dependências estáticas, conforme ilustrado no Código 4.6.

É importante salientar que o *ExecutionNode* é específico por plataforma e usado por contêineres de várias linguagens, assim, o método *install* da

Código 4.5: Interface OMG IDL da faceta *ComponentRepository*

```
module scs {
module deployer {
  struct Package {
    ComponentDescription info;
    OctetSeq file;
  };
};
module repository {
  interface ComponentRepository {
    // publicação
    void upload (in deployer::Package pkg, string help)
      raises (ComponentAlreadyUploaded, ComponentNotUploaded);
    // revogação
    void delete (in core::ComponentId id, in string lang, in string plat)
      raises (ComponentNotUploaded);
    // obtenção
    deployer::Package download (in core::ComponentId id, in string lang,
      in string plat)                    raises (ComponentNotUploaded);
    // consultas
    deployer::ComponentDescription getDescription (in core::ComponentId id,
      in string lang, in string plat)    raises (ComponentNotUploaded);
    deployer::ComponentDescriptionSeq getAllDescriptions();
    deployer::ComponentDescriptionSeq searchById (in core::ComponentId id);
  };
}; };
```

Código 4.6: Interface OMG IDL da faceta *Installer*

```
module scs { module execution_node {
interface Installer {
  void install(in core::ComponentId id, in string lang);
    raises (ComponentNotInstalled);
  void uninstall(in core::ComponentId id, in string lang);
  deployer::ComponentDescription getDescription(in core::ComponentId id,
    in string lang)
    raises (ComponentNotInstalled);
  deployer::ComponentDescriptionSeq getInstalledComponents(in string lang);
  // informa a raiz da árvore de instalações por linguagem
  string getRootPath(in string lang);
  // informa local da instalação de um componente específico
  string getInstallPath(in core::ComponentId id, in string lang);
  // informa a lista de localizações das dependências de um componente
  StringSeq getDependenciesPath(in core::ComponentId id, in string lang);
}; }; };
```


faceta *Installer* precisa receber (i) o identificador do componente, conhecido como *ComponentId* (composto por nome e versão), e (ii) o identificador da linguagem referente ao *Container* (composto por uma *string*, exemplos: *lua*, *java* ou *cpp*). Por fim, a implementação da faceta *Installer* busca os pacotes de componentes no *Repository* diferenciando as implementações através dos identificadores (i) do componente, (ii) da linguagem, e (iii) da plataforma (composto por uma *string*, exemplos: *macosx*, *linux-x86* ou *win32*). De posse do pacote do componente, a tarefa de instalação passa a ser comandada pelo LuaRocks que decide como instalar o pacote e suas dependências estáticas. O *ExecutionNode* ainda explora duas importantes capacidades do LuaRocks para: (i) manipular diferentes árvores de instalações (vide Seção 4.2.4) para manter uma árvore para cada par: linguagem e plataforma; e (ii) permitir a

instalação de pacotes versionados (vide Seção 4.2.3). Assim torna-se possível um melhor isolamento entre as instalações e evita-se problemas na carga de componentes relacionados à má organização entre os artefatos dependentes.

A nova arquitetura da infraestrutura de execução do SCS após essas mudanças é ilustrada na Figura 4.1. Nessa nova arquitetura, o *ExecutionNode* deve estar conectado a um ou mais componentes *Repository*. Quando o *Container* recebe uma solicitação de carga (ou descarga) de um componente, ele solicita a tarefa de instalação (ou desinstalação) através da faceta *Installer* do *ExecutionNode*. Caso o componente ainda não esteja instalado, *ExecutionNode* instala os componentes após obter a implementação disponível no *Repository* que seja compatível com a solicitação requisitada pelo *Container*.

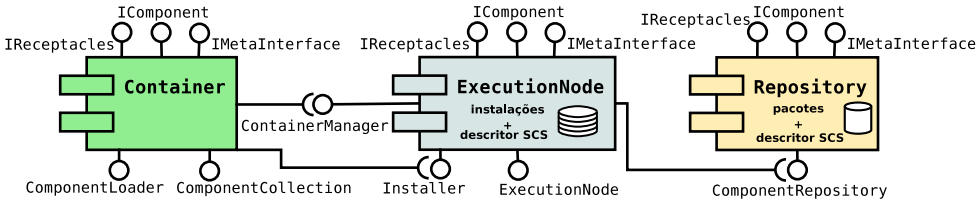

Figura 4.1: infraestrutura de execução SCS com suporte a empacotamento

4.5 Considerações Finais

A experiência de integrar o LuaRocks ao SCS confirmou as premissas da facilidade de reuso e extensão do LuaRocks. O uso de procedimentos de empacotamento padronizados oferecem uma maior confiabilidade nos pacotes que são publicados nos repositórios e tendem a diminuir erros por falha humana. Além disso, o fato de diferentes grupos de desenvolvedores, administradores e usuários usarem procedimentos simples, mais disciplinados e unificados no ciclo de pós-desenvolvimento é benéfico, pois permite a comunicação entre os grupos em um nível de abstração mais alto [47].