

Pontifícia Universidade Católica do Rio de Janeiro  
Departamento de Informática  
Mestrado em Informática

# Implementação de Repositório e Instalador para um Sistema de Componentes de Software Distribuído

**Aluno:**

Amadeu Andrade Barbosa Júnior

**Matrícula do aluno:**

0721312

**Orientador:**

Prof<sup>º</sup> Renato Cerqueira

**Disciplina:**

Projeto Final de Programação

**Coordenador da Disciplina:**

Prof<sup>º</sup> Carlos José P. De Lucena

# ÍNDICE

<b>1. INTRODUÇÃO.....</b>	<b>3</b>
<b>2. ESPECIFICAÇÃO.....</b>	<b>4</b>
2.1. CONTEXTUALIZAÇÃO.....	4
2.2. MOTIVAÇÃO.....	5
2.3. OBJETIVOS.....	6
2.4. CASOS DE USO.....	6
2.4.1. <i>Caso de uso: Carregar Componente.....</i>	<i>7</i>
2.4.2. <i>Caso de uso: Descarregar Componente.....</i>	<i>8</i>
2.4.3. <i>Caso de uso: Instalar Componente.....</i>	<i>8</i>
2.4.4. <i>Caso de uso: Desinstalar Componente.....</i>	<i>9</i>
2.4.5. <i>Caso de uso: Publicar Componente no Repositório.....</i>	<i>9</i>
2.4.6. <i>Caso de uso: Remover Componente do Repositório.....</i>	<i>10</i>
<b>3. IMPLEMENTAÇÃO DO INSTALADOR E REPOSITÓRIO.....</b>	<b>11</b>
3.1. DETALHES SOBRE O REUSO DO SISTEMA DE EMPACOTAMENTO.....	11
3.2. DIAGRAMA DE COMPONENTES.....	14
3.3. DIAGRAMAS DE SEQÜÊNCIA.....	15
3.3.1. <i>Carregar Componente.....</i>	<i>15</i>
3.3.2. <i>Descarregar Componente.....</i>	<i>16</i>
3.3.3. <i>Instalar Componente.....</i>	<i>16</i>
3.3.4. <i>Desinstalar Componente.....</i>	<i>17</i>
3.3.5. <i>Publicar Componente no Repositório.....</i>	<i>17</i>
3.3.6. <i>Remover Componente do Repositório.....</i>	<i>17</i>
<b>4. EXEMPLO DE USO, DOCUMENTAÇÃO E TESTES.....</b>	<b>18</b>
4.1. LANÇAMENTO DA INFRA-ESTRUTURA DE EXECUÇÃO DO SCS.....	18
4.1.1. <i>Lançar nó de execução.....</i>	<i>19</i>
4.1.2. <i>Lançar repositório.....</i>	<i>19</i>
4.2. APLICAÇÃO USUÁRIA.....	19
4.3. DOCUMENTAÇÃO E TESTES AUTOMATIZADOS.....	22
4.3.1. <i>Documentação de código.....</i>	<i>22</i>
4.3.2. <i>Testes automatizados.....</i>	<i>22</i>
<b>5. FUTURAS IMPLEMENTAÇÕES.....</b>	<b>23</b>
5.1. AMPLIAR OS TESTES MULTI-PLATAFORMA E MULTI-LINGUAGEM.....	23
5.2. PACOTES COMPARTILHADOS ENTRE COMPONENTES EM DIFERENTES LINGUAGENS.....	23
5.3. FACILITAR O EMPACOTAMENTO E PUBLICAÇÃO.....	23
5.4. FERRAMENTA DE MODELAGEM DE COMPONENTES.....	23
<b>6. CONSIDERAÇÕES FINAIS.....</b>	<b>24</b>
<b>7. REFERÊNCIAS BIBLIOGRÁFICAS.....</b>	<b>25</b>

# 1. Introdução

Neste trabalho será apresentada a implementação de um Repositório de Componentes e um Instalador de Componentes que permitam a descrição e instalação de dependências externas ao componente de software para cenários multi-linguagem e multi-plataforma. Tais implementações foram realizadas sobre o *framework* SCS [1,2], que implementa um modelo próprio de componentes e utiliza a tecnologia CORBA para interoperar componentes desenvolvidos em diferentes linguagens e para diferentes plataformas.

No capítulo 2 apresenta-se uma contextualização à infra-estrutura provida pelo SCS e os requisitos identificados para publicar e instalar componentes multi-linguagem e multi-plataforma. Depois, no capítulo 3, os detalhes tecnológicos da solução, bem como os diagramas de componentes e sequência do que está implementado. Assim, apresenta-se, no capítulo 4, um exemplo de uso, a documentação de código e testes. Já no capítulo 5 lista-se possibilidades para futuras implementações. Por fim, no capítulo 6 são apresentadas as considerações finais.

## 2. Especificação

### 2.1. Contextualização

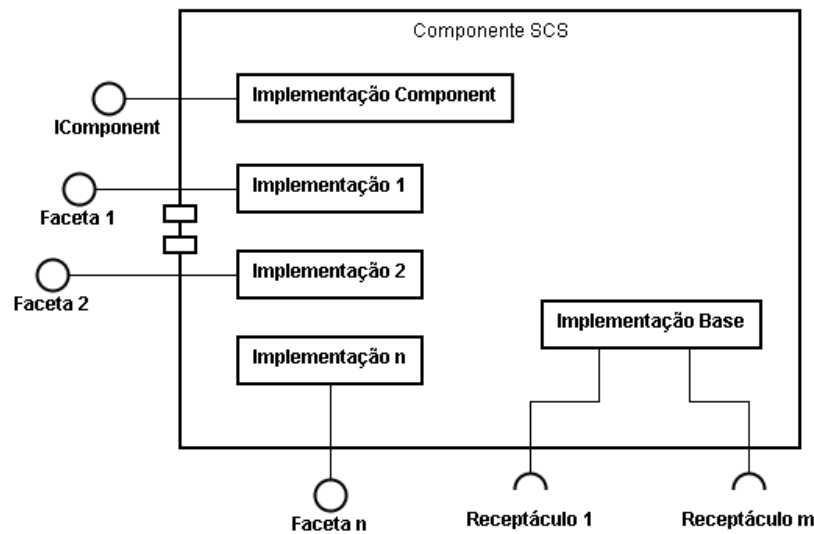


Figura 1: Modelo de Componentes SCS

Em [1] apresenta-se o modelo de componente SCS. Nesse um componente é representado pelas suas **facet**as (interfaces de serviços providos) e **receptáculos** (interfaces de serviços requeridos), conforme **Figura 1**. Em particular, a faceta **IComponent** é obrigatória e provém na sua interface métodos para início, término, auto-identificação e obtenção das outras facetas do componente. Opcionalmente, o modelo também disponibiliza mais duas facetas para inspeção, são elas **IReceptacles**, responsável pelo gerenciamento de conexões aos receptáculos do usuário, e **IMetaInterface**, que permite obter descrições de todas as facetas e receptáculos que formam o componente. Todas as facetas e receptáculos em SCS possuem interfaces CORBA associadas, permitindo assim que um componente SCS interopere com programas escritos em outras linguagens e que rodem em diferentes plataformas (sistemas operacionais). É importante salientar que os componentes SCS usam interfaces CORBA 2.x [3] e não utiliza a especificação CORBA Component Model [4], exatamente na tentativa de simplificar o uso das tecnologias de componentes.

Ainda em [1] o autor descreve a implementação de uma infra-estrutura de apoio à implantação e execução de componentes SCS que pode ser visualizada na **Figura 2**. Essa é composta pelos componentes **ExecutionNode** (nó de execução), **ComponentContainer** (contêiner) e **ComponentRepository** (repositório). Para instanciar um componente, é preciso ter o mesmo publicado no repositório, para então



implementava-se o mesmo processo de instalação. Além disso, se é o contêiner o responsável pela instalação do componente, ele precisa lidar com a diversidade de utilitários de cada plataforma para proceder à instalação (compactadores, comandos de sistemas, diferentes nomes de diretórios). É fácil perceber que essa abordagem é propensa a erros e pode gerar inconsistências para um sistema final que utilize componentes em mais de uma linguagem.

Outra limitação é que não havia como descrever dependências externas aos componentes de software (como bibliotecas dinâmicas, módulos (JAR ou arquivos lua) adicionais), o que é muito comum na maioria dos softwares em produção. Assim, ficava a cargo do programador a tarefa onerosa de encapsular em seu componente todas as dependências externas para cada combinação de linguagem e plataforma. Caso algum software dependente não fosse incluído no componente ter-se-ia uma exceção em tempo de execução (no ato da carga do componente pelo contêiner).

Por outro lado, o nó de execução não precisa estar implementado na mesma linguagem do componente que deseja-se carregar, inclusive o nó de execução não age em momento algum durante a carga de um componente, conforme infra-estrutura em [1]. Dessa forma, é caro manter um nó de execução implementado em duas linguagens considerando-se que não há especificidade em seu funcionamento que demande ter mais de uma implementação. Para o nó de execução só é importante tratar o lançamento dos processos dos contêineres das diferentes linguagens e em diferentes plataformas.

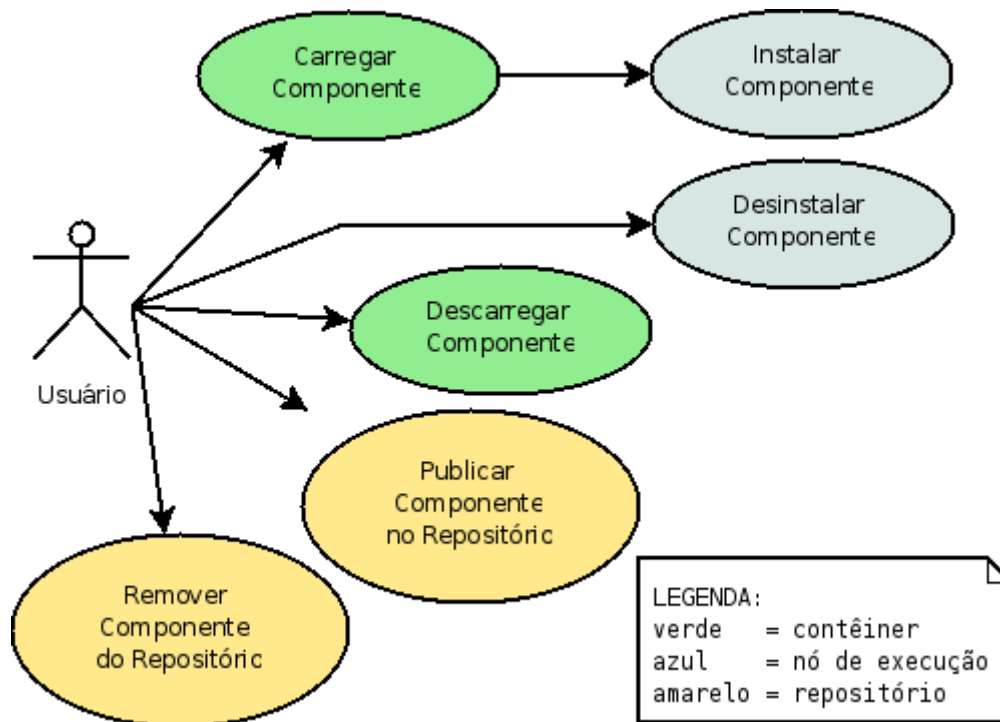
### **2.3. Objetivos**

Conforme as motivações identificados na seção 2.2 deste, assumiu-se por objetivos específicos:

1. Incorporar ou desenvolver um sistema de empacotamento capaz de descrever e instalar dependências externas aos componentes de software
2. Delegar ao nó de execução a tarefa de instalar os componentes no sistema de arquivos local do contêiner
3. Adaptar o repositório ao sistema de empacotamento escolhido
4. Gerar testes e documentação de código para todo *framework* SCS

### **2.4. Casos de uso**

A partir dos objetivos, a implementação deve respeitar os seguintes casos de uso.



### 2.4.1. Caso de uso: Carregar Componente

#### Visão Geral

O Usuário deseja carregar um novo componente para compor sua aplicação. Para isso utiliza-se um identificador do componente a ser carregado.

#### Atores

Usuário

#### Pré-Condições

- A infra-estrutura do SCS deve estar ativa
- O componente deve estar publicado em algum repositório conhecido pela infra-estrutura do SCS

#### Seqüência Típica de Eventos

1. Usuário informa o identificador do componente que deseja carregar;
2. É iniciado o caso de uso Instalar Componente;
3. O componente é carregado;
4. Usuário é informado sobre a nova instância do componente.

#### Fluxo Alternativo

No item 2, a instalação do componente pode falhar e interrompe esse caso de uso.

No item 3, a carga pode falhar e interrompe esse caso de uso.

## **2.4.2. Caso de uso: Descarregar Componente**

### **Visão Geral**

O Usuário deseja descarregar um componente previamente carregado. Para isso utiliza-se um identificador do componente a ser carregado.

### **Atores**

Usuário

### **Pré-Condições**

- A infra-estrutura do SCS deve estar ativa

### **Seqüência Típica de Eventos**

1. Usuário informa a instância do componente que deseja descarregar;
2. Sistema desativa o componente e libera a instância;

### **Fluxo Alternativo**

No item 2, a desativação do componente pode falhar e interrompe esse caso de uso.

## **2.4.3. Caso de uso: Instalar Componente**

### **Visão Geral**

O Usuário solicita a instalação de um determinado componente.

### **Atores**

Usuário

### **Pré-Condições**

- A infra-estrutura do SCS deve estar ativa
- O componente deve estar publicado em algum repositório conhecido pela infra-estrutura do SCS

### **Seqüência Típica de Eventos**

1. Usuário informa o identificador do componente a ser instalado;
2. Usuário informa a linguagem do componente a ser instalado;
3. Sistema obtém o componente de um repositório e instala-o no sistema de arquivos local do contêiner da linguagem informada;
4. Usuário é informado sobre as descrições do componente.

### **Fluxo Alternativo**

No item 3, caso o componente já esteja instalado, o caso de uso finaliza normalmente. Caso a obtenção do repositório ou a instalação falhem, esse caso de uso é interrompido.

#### **2.4.4. Caso de uso: Desinstalar Componente**

##### **Visão Geral**

O Usuário solicita a desinstalação de um determinado componente.

##### **Atores**

Usuário

##### **Pré-Condições**

- A infra-estrutura do SCS deve estar ativa

##### **Seqüência Típica de Eventos**

1. Usuário informa o identificador do componente a ser desinstalado;
2. Usuário informa a linguagem do componente a ser desinstalado;
3. Sistema remove o componente do sistema de arquivos local do contêiner da linguagem informada.

##### **Fluxo Alternativo**

No item 3, caso o componente não esteja instalado, o caso de uso finaliza normalmente. Caso contrário, se a remoção falhar então esse caso de uso é interrompido.

#### **2.4.5. Caso de uso: Publicar Componente no Repositório**

##### **Visão Geral**

O Usuário deseja publicar um componente em um repositório para posteriormente carregá-lo em algum contêiner.

##### **Atores**

Usuário

##### **Pré-Condições**

- Usuário precisa conhecer o repositório remoto onde deseja publicar

##### **Seqüência Típica de Eventos**

1. Usuário informa o identificador do componente a publicar;
2. Usuário informa a implementação do componente a publicar;
3. Usuário informa a linguagem em que o componente foi desenvolvido;
4. Usuário informa a plataforma suportada pelo componente;
5. Sistema publica e mantém o componente publicado.

### **Fluxo Alternativo**

No item 3, a publicação pode falhar caso já exista um componente similar publicado, o que interrompe esse caso de uso.

## **2.4.6. Caso de uso: Remover Componente do Repositório**

### **Visão Geral**

O Usuário deseja remover um componente publicado em um repositório.

### **Atores**

Usuário

### **Pré-Condições**

- Usuário precisa conhecer o repositório remoto do qual quer remover

### **Seqüência Típica de Eventos**

1. Usuário informa o identificador do componente a publicar;
2. Usuário informa a linguagem em que o componente foi desenvolvido;
3. Usuário informa a plataforma suportada pelo componente;
4. Sistema remove o componente do repositório tornando-o inacessível para consultas.

### **Fluxo Alternativo**

No item 4, caso o componente não esteja publicado, esse caso de uso é interrompido.

## 3. Implementação do Instalador e Repositório

Conforme os objetivos listados na seção 2.3, foi preciso avaliar e escolher um sistema de empacotamento para resolver dependências externas aos componentes de software. Os detalhes da escolha são comentados na seção 3.1 abaixo, depois apresenta-se os diagramas de componentes na seção 3.2 e os de sequência na seção 3.3.

### 3.1. Detalhes sobre o reuso do sistema de empacotamento

Em consonância ao primeiro objetivo, descrito na seção 2.3, já estava em andamento a realização do estudo aprofundado sobre técnicas de implantação de componentes de software distribuídos, conforme os trabalhos [5,6]. A escolha do sistema de empacotamento foi baseada nos seguintes critérios:

1. Capacidade de resolver dependências entre softwares
2. Capacidade de obter automaticamente de um repositório os softwares dependentes
3. Capacidade de manter instalado um mesmo software em diferentes versões
4. Capacidade de funcionar em diversas plataformas
5. Capacidade de instalar softwares desenvolvidos em diversas linguagens
6. Facilidade de reuso e extensão

A tecnologia que se mostrou mais adequada com a maioria dos critérios acima foi o sistema de empacotamento **LuaRocks**<sup>2</sup>. Embora o LuaRocks originalmente destine-se à instalação de módulos escritos na linguagem Lua, os critérios 3, 4 e 6 tiveram grande força na sua escolha. Sistemas de pacotes tradicionais como o **APT-GET**<sup>3</sup> e **RPM** não são capazes de manter a instalação de softwares em diferentes versões, e, principalmente, não funcionam em plataformas Microsoft. Uma discussão mais aprofundada sobre esse tema pode vista em [5].

Para tornar o LuaRocks capaz de instalar componentes em Java, realizou-se uma pequena modificação na sua rotina de instalação para considerar adicionalmente um diretório especializado para abrigar os arquivos JAR. Todo o LuaRocks foi reusado bastando adotar uma padronização para os diretórios de instalação. O LuaRocks possui a capacidade de operar sobre diferentes árvores de instalação através da configuração de uma variável de ambiente (parâmetro em todas suas rotinas principais – *install*, *remove*, *list*). Para cada árvore de instalação o LuaRocks gera um arquivo de manifesto com todas as informações necessárias para a gerência dos softwares instalados. Tal recurso

---

2 [www.luarocks.org](http://www.luarocks.org)

3 [www.debian.org/doc/manuals/apt-howto](http://www.debian.org/doc/manuals/apt-howto)

não foi encontrado em nenhum dos outros sistemas estudados e foi o diferencial para a adoção do LuaRocks.

Para interagir com o processo de instalação, foi criada uma nova faceta (**Installer**), no nó de execução, cuja interface é apresentada no **Código 1**. Assim delega-se ao nó de execução a tarefa de instalar e desinstalar os componentes de softwares e suas dependências externas. Dessa forma a integração com o LuaRocks tornou-se mais simples pois adotamos a implementação Lua do nó de execução. Conforme já discutido, o nó de execução não precisa estar implementado em diferentes linguagens e sua versão em Lua já trata a diversidade de plataformas (assim como o LuaRocks).

Dessa forma, agora o contêiner de componentes usa a nova faceta **Installer** do nó de execução. Após instalar um componente o contêiner obtém a descrição do componente instalado. Tal descrição contém uma string com o nome canônico da fábrica de componentes que deve ser instanciada<sup>4</sup> e usada para produzir instâncias de componentes.

```
module scs {
module execution_node {
    interface Installer {
        void install (in core::ComponentId id, in string lang, out
core::ComponentDescription desc) raises (ComponentNotInstalled);
        void uninstall (in core::ComponentId id, in string lang);
        void getInstalledComponents (in string lang, out
core::ComponentDescriptionSeq components);
        void getRootPath (in string lang, out path);
        void getDependenciesPath (in core::ComponentId id, in string lang,
out StringSeq paths);
    };
};};
```

**Código 1: IDL CORBA da faceta para instalação**

Por outro lado, no repositório de componentes a implementação do componente deve estar armazenada no formato de pacotes do LuaRocks. Para isso, dispõe-se de um utilitário para **empacotar** os softwares do usuário que é usado para produzir um pacote (chamado de *rock*) válido. Atualmente, o programador que deseja publicar um software no repositório deve usar diretamente tal utilitário pela linha de comando para criar um *rock* e então enviá-lo para o repositório.

<sup>4</sup> Em Lua usa-se “require” passando a string da fábrica, em Java usa-se o *Class.forName*.

O utilitário de empacotamento do LuaRocks faz uso de um arquivo de descrição próprio conhecido como **rockspec**, exemplificado no **Código 2**. Nessa descrição é possível indicar quais as dependências precisam ser satisfeitas para instalar um *rock*. No exemplo usa-se tal campo para informar que o pacote *PingPong* depende dos binários do *Lua* versão 5.1 e do módulo *md5* em qualquer versão. Num cenário de uso real é preciso ter um **rockspec** para cada implementação (Lua, Java) do componente PingPong. Em particular, o campo *source* é usado para indicar onde encontra-se os fontes do componente para que de um **rockspec** seja possível gerar um *rock*.

```
package = "PingPong"
version = "1.0-0"
description = {
    summary = "This rocks will be a component some day (v1)",
    maintainer = "Amadeu A. B. Jr",
}
dependencies = { "lua >= 5.1", "md5" }
source = { url = "file://PingPong.zip" }
```

**Código 2: Arquivo Rockspec que descreve um pacote LuaRocks**

A interface original da faceta **ComponentRepository** foi pouco modificada e pode ser conferida no **Código 3** :

1. Alguns métodos foram renomeados: (i) de *install* para *upload*; (ii) de *GetComponentFile* para *download*; (iii) de *uninstall* para *delete*;
2. No método **upload** houve uma redução de parâmetros (antes eram 6, agora são 3) em favor do uso da estrutura **ComponentDescription** (que já existia anteriormente) como parâmetro. Nessa háconsta as informações: nome da fábrica do componente, identificador do componente, linguagem, plataforma e um booleano indicando se deve ser usado como *singleton*.

No repositório, os pacotes são mantidos em memória e em disco, indexados por linguagem e plataforma. As descrições providas na publicação são persistidas para no relançamento do repositório os componentes já estarem disponíveis para obtenção. Atualmente, durante a instalação de um componente no nó de execução, ao identificar que uma dependência externa precisa ser obtida, usar-se-á um outro repositório básico<sup>5</sup> do LuaRocks. Essa estratégia pode ser aprimorada, pois o LuaRocks permite configurar quais repositórios devem ser usados para a obtenção dos rocks. Embora, o LuaRocks aceite repositórios nos protocolos HTTP, CVS, SCM (dois últimos são para sistemas de controle de versão), o que pode ser facilmente estendido para CORBA.

<sup>5</sup> Tipicamente o <http://luarocks.luaforge.net/rocks> usando o protocolo HTTP

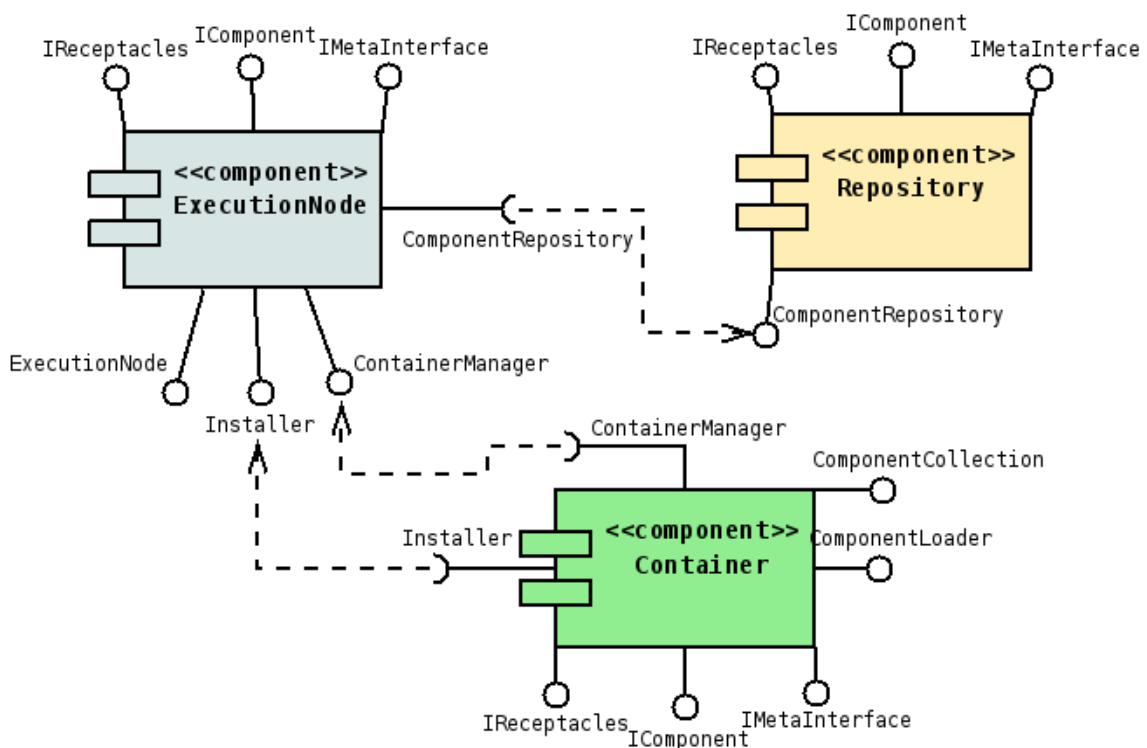
```

module scs { module repository {
    interface ComponentRepository {
        void upload (in core::ComponentDescription desc, in
container::OctetSeq file, in string help_info) raises
(ComponentAlreadyUploaded, ComponentNotUploaded);
        void delete (in core::ComponentId id, in string lang, in string plat)
raises (ComponentNotUploaded);
        void copy (in core::ComponentDescription desc, in ComponentRepository
rep) raises (ComponentAlreadyUploaded, ComponentNotUploaded);
        void download (in core::ComponentId id, in string lang, in string
plat, out container::OctetSeq impl) raises (ComponentNotUploaded);
        void downloadByPieces(in core::ComponentId id, in string lang, in
string plat, in long size, in long start, out boolean finished, out
container::OctetSeq buffer) raises (ComponentNotUploaded);
        void getDescription (in core::ComponentId id, in string lang, in
string plat, out core::ComponentDescription desc) raises
(ComponentNotUploaded);
        void getAllDescriptions (out core::ComponentDescriptionSeq results);
    };
}};

```

**Código 3: IDL CORBA para faceta do repositório**

### 3.2. Diagrama de componentes



**Figura 4: Diagrama de componentes de todo SCS**

### 3.3. Diagramas de Seqüência

#### 3.3.1. Carregar Componente

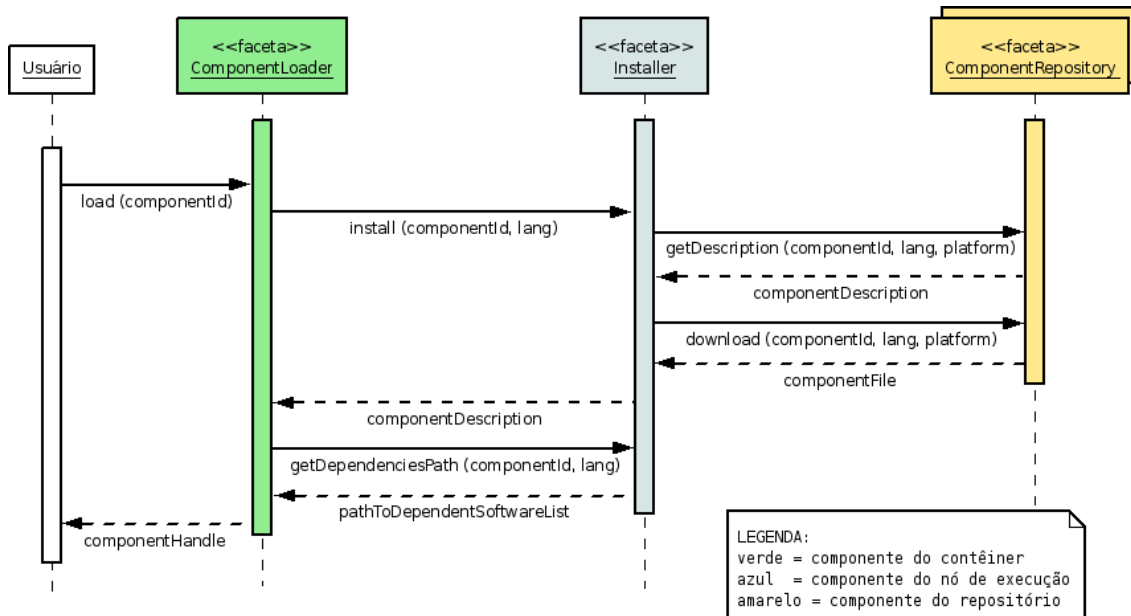


Figura 5: Seqüência de eventos para carga de um componente

#### 3.3.2. Descarregar Componente

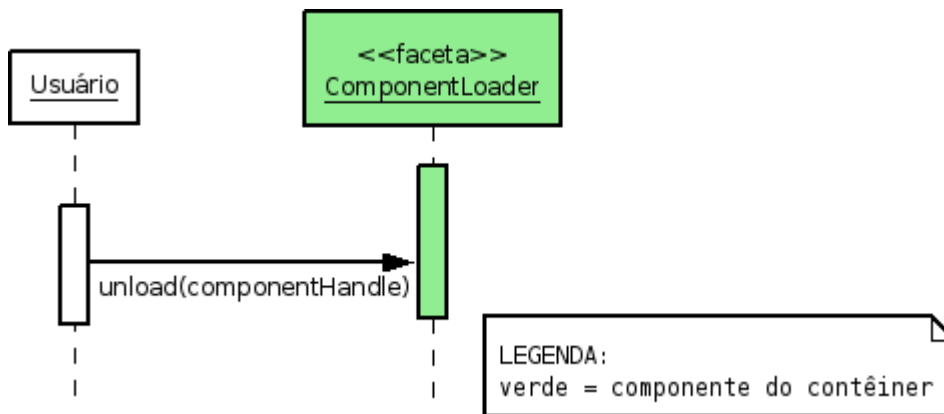


Figura 6: Seqüência de eventos para descarrega uma instância de componente

### 3.3.3. Instalar Componente

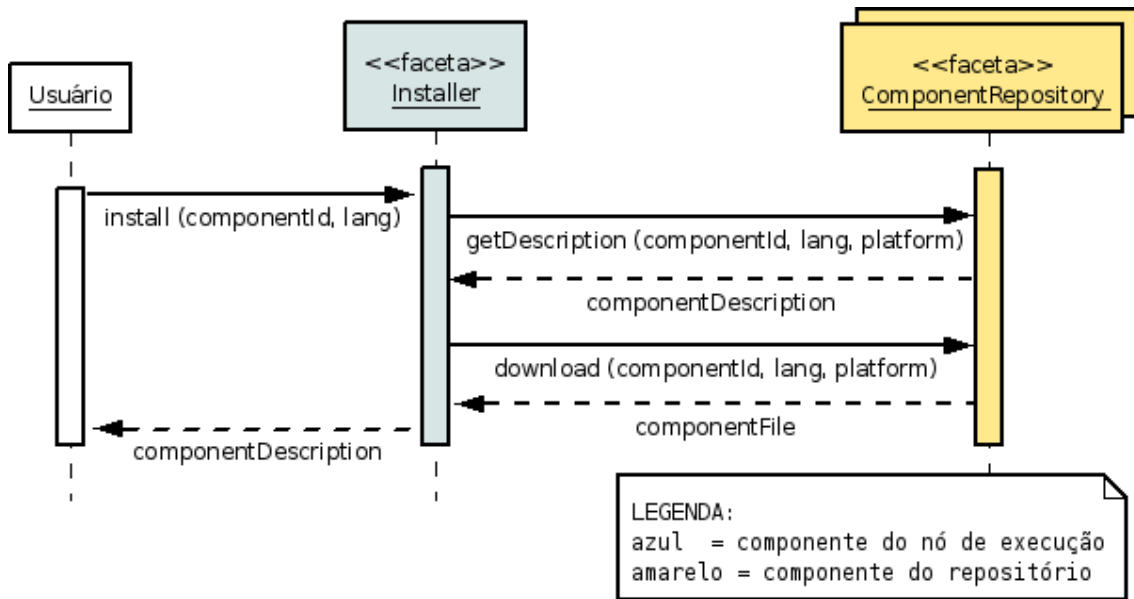


Figura 7: Sequência de eventos para instalar um componente no sistema de arquivos para certa linguagem

### 3.3.4. Desinstalar Componente

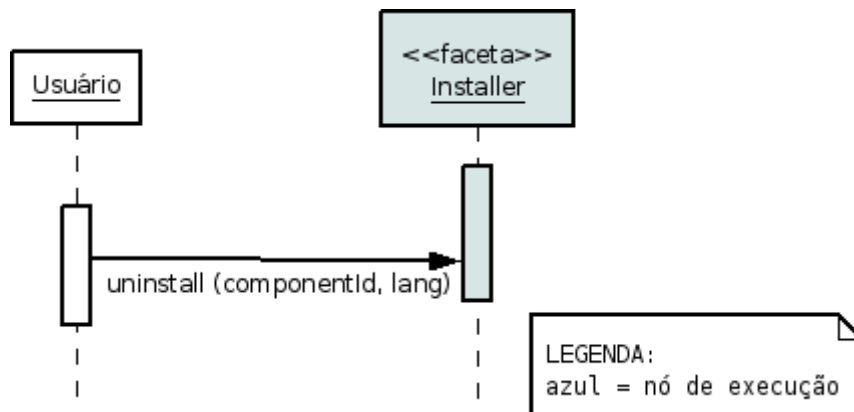


Figura 8: Sequência de eventos para desinstalar um componente do sistema de arquivos para certa linguagem

### 3.3.5. Publicar Componente no Repositório

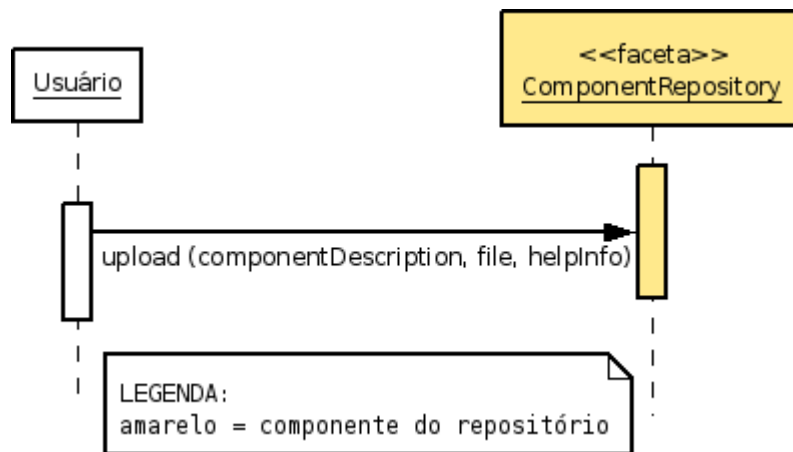


Figura 9: Sequência de eventos para publicar um componente num repositório

### 3.3.6. Remover Componente do Repositório

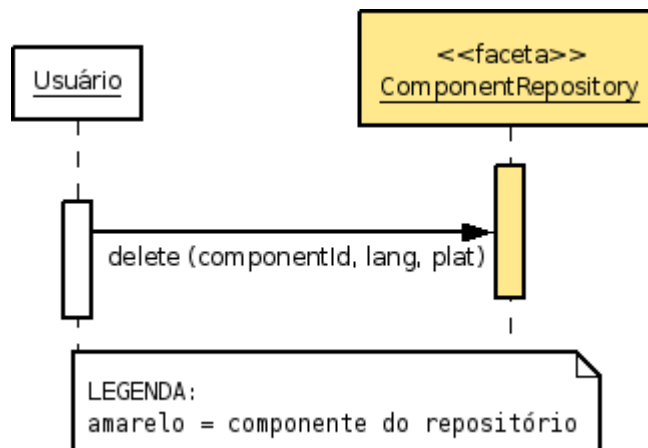


Figura 10: Sequência de eventos para remover um componente de um repositório

## 4. Exemplo de Uso, documentação e testes

Nesta seção exemplifica-se o lançamento da infra-estrutura de execução e uma aplicação usuária que publicará um componente **PingPong** e carregar duas instâncias desse componente. Porém, antes é preciso assegurar-se de que os seguintes softwares estejam previamente instalados:

- SCS Lua:
  - Lua 5.1 ou posterior – <http://www.lua.org>
  - LuaRocks – <http://www.luarocks.org>
  - OiL (pode ser instalado através do LuaRocks) – <http://oil.luaforge.net>
  - Configuração da variável de ambiente LUA\_PATH:  
**LUA\_PATH="<instalação-scs>/src/lua/?lua;"**
- SCS Java:
  - Sun Java 6 – <http://java.sun.com>
  - JacORB (jars incluídos na pasta libs) – <http://www.jacorb.org>
  - JUnit 4.3 ou posterior (apenas para os testes) – <http://junit.org>
  - Para compilar o SCS, entre na pasta **src/java** e:
    1. Compile as IDL's CORBA: **make idl-generation**
    2. Compile o código do SCS manualmente: **make just-compile**

### 4.1. Lançamento da infra-estrutura de execução do SCS

Como vimos na seção 2.1, a infra-estrutura de execução do SCS é composta pelos componentes: nó de execução, contêiner e repositório. O interessante desses serem componentes é que o lançamento pode ser personalizado de acordo com a necessidade de cada usuário. Por exemplo, a partir do nó de execução é possível criar novos contêiners, carregar um ou mais repositórios e conectá-los em um ou mais nós de execução. Contudo neste documento, ilustramos um cenário de lançamento básico para um usuário que deseja ter um repositório disponível para publicar seus componentes e então poder carregá-los num contêiner. Para concretizar tal cenário é preciso: (i) lançar o nó de execução e (ii) lançar o repositório de componentes. O lançamento do contêiner ficará a cargo da aplicação do usuário.

#### 4.1.1. Lançar nó de execução

O nó de execução fica localizado na pasta **src/lu/execuion\_node** e lá existe o arquivo **Properties.txt**, conforme **Código 4**. Há duas propriedades<sup>6</sup> novas e importantes que são: (i) **containerPath** que indica o diretório que armazenará todas as instalações dos componentes e suas dependências, (ii) **platform** onde especifica-se qual a plataforma atual do nó de execução. A propriedade **enIOR** é importante para que o contêiner encontre a referência remota para o nó de execução.

```
#platform = linux-x86_64
#supported_langs = lua java
#host = localhost
#port = 20202
#lua_name = lua
#path = ../../scs
#containerPath = ../container
enIOR = /tmp/execution_node.ior
```

**Código 4: Arquivo de propriedades do nó de execução**

Uma vez que as propriedades forem preenchidas basta executar o comando:

```
lua ExecutionNode.lua
```

#### 4.1.2. Lançar repositório

O repositório de componentes fica localizado na pasta **src/lu/repository** e lá existe o arquivo **Properties.txt**, conforme **Código 5**. Há uma propriedade nova que é a **pkgdir** que indica o diretório que armazenará os pacotes *rock* publicados no repositório. Tendo as propriedades preenchidas basta executar o comando:

```
lua repositoryconfig.lua
```

```
use_openbus = false
pkgdir = ../repository/packages/
```

**Código 5: Arquivo de propriedades do repositório**

## 4.2. Aplicação usuária

Para carregar o componente é preciso, conforme o caso de uso Carregar Componente da seção 2.4.1, ter a infra-estrutura ativa e o componente publicado num repositório. Logo para isso a aplicação usuária, a seguir, precisa cuidar do caso de uso Publicar Componente, seção 2.4.5. Só então será possível carregar o componente. O

<sup>6</sup> Os arquivos de propriedades são específicos do SCS, e o caractere # significa que a propriedade é apenas leitura e não pode ser alterada ao longo da execução.

código está dividido em trechos (Código 6, Código 7, Código 8, Código 9) complementares para melhor visualizar o que está sendo executado. Esse mesmo código está na pasta `src/luascscs/demos/pingpong`, no arquivo `pingpongjavaconfig.lua`. Para executá-lo usa-se o comando:

### lua pingpongjavaconfig.lua

```
--- inicialização do ORB CORBA
oil = require "oil"
oil.orb = oil.init({host = "localhost", port = 20287})
oil.orb:loadidlfile("../..../idl/repository.idl")
oil.orb:loadidlfile("../..../idl/deployment.idl")
oil.orb:loadidlfile("../..../idl/pingPong.idl")
oil.main(function()
  --- passos necessários para obter as referências à infra-estrutura
  -- obtendo a referência ao nó de execução
  exNodeComponent = oil.orb:newproxy(oil.readfrom("/tmp/execution_node.ior"))
  -- obtendo a faceta ExecutionNode
  exNodeFacet =
  exNodeComponent:getFacet("IDL:scs/execution_node/ExecutionNode:1.0")
  exNodeFacet = oil.orb:narrow(exNodeFacet)
```

**Código 6: Inicialização e referências para a infra-estrutura**

```
--- passos necessários para lançar o contêiner
-- criando um contêiner Java
containerName = "ComponentContainerJava"
containerPropertySeq = {
  { name = "language" , value = "java", read_only = false },
  { name = "classpath" , value = "../..../java", read_only = false},
}
container = exNodeFacet:startContainer(containerName,containerPropertySeq)
container:startup()

-- obtendo a faceta ComponentLoader do container Java
loaderFacet = container:getFacetByName("ComponentLoader")
loaderFacet = oil.orb:narrow(loaderFacet,
"IDL:scs/container/ComponentLoader:1.0")
```

**Código 7: Criando um contêiner**

```

--- passos necessários para publicar no repositório
-- criando pacote rock
os.execute("zip -r PingPong.zip PingPong/")
os.execute("luarocks pack pingpong-1.0-0.rockspec")
impl = assert(io.open("pingpong-1.0-0.src.rock","rb"):read("*a"))
componentId = { name = "PingPongServer",
                major_version = 1, minor_version = 0, patch_version = 0, }
componentDesc = { id = componentId,
                  lang = "java", plat = "linux-x86_64", shared = false,
                  entry_point = "scs.demos.pingpong.servant.PingPongServerFactory", }
-- obtendo a referência para o repositório
exNodeRecept =
oil.orb:narrow(exNodeComponent:getFacetByName("IReceptacles"))
repoConnections = exNodeRecept:getConnections("ComponentRepository")
firstRepo = oil.orb:narrow(assert(repoConnections[1].objref))
-- publicando componentes no repositório
firstRepo:upload(componentDesc,impl,"Some text about")

```

**Código 8: Publicação no repositório**

```

--- finalmente, é possível carregar os componentes
handle1 = loaderFacet:load(componentId, {}); handle1.cmp:startup()
handle2 = loaderFacet:load(componentId, {}); handle2.cmp:startup()

- -- agora todo código abaixo é específico da aplicação PingPong
-- obtendo a faceta IReceptacles de cada componente
pp1Rec = handle1.cmp:getFacetByName("IReceptacles")
pp1Rec = oil.orb:narrow(pp1Rec, "scs::core::IReceptacles")
pp2Rec = handle2.cmp:getFacetByName("IReceptacles")
pp2Rec = oil.orb:narrow(pp2Rec, "scs::core::IReceptacles")
-- obtendo a faceta PingPongServer de cada componente
pp1 = handle1.cmp:getFacetByName("PingPongServer")
pp1 = oil.orb:narrow(pp1, "scs::demos::pingpong::PingPongServer")
pp2 = handle2.cmp:getFacetByName("PingPongServer")
pp2 = oil.orb:narrow(pp2, "scs::demos::pingpong::PingPongServer")
-- conectando os componentes
pp1Rec:connect("PingPongServer", pp2)
pp2Rec:connect("PingPongServer", pp1)
-- definindo uma identificação
pp1:setId(1)
pp2:setId(2)
-- iniciando a rodada de pings e pongs
pp1:start()
end)

```

**Código 9: Carga de componente e tarefas específicas da aplicação usuária**

### 4.3. Documentação e testes automatizados

Embora neste trabalho tenha sido necessário alterar apenas uma parte da infra-estrutura de execução proposta em [1], como a implementação disponível dessa infra-estrutura não possuía documentação de código nem testes unitários, foi preciso documentar e testar todo código, incluindo as partes onde não foi houve mudanças.

#### 4.3.1. Documentação de código

Uma vez que a implementação da infra-estrutura consta em duas linguagens foi necessário documentar ambos códigos. Em Java utilizou-se anotações **Javadoc**, e em Lua anotações do **LuaDoc**<sup>7</sup>. A documentação está na pasta **doc/java** e **doc/lu**.

#### 4.3.2. Testes automatizados

Atualmente só há testes automatizados escritos em Java utilizando JUnit4. Contudo já é satisfatório para testar as funcionalidades expostas pelas facetas dos componentes, uma vez que elas possuem interfaces IDL CORBA associadas e podem ter seus métodos invocados a partir de Java.

Em particular, o cenário distribuído dificulta o uso natural dos testes unitários, nesse trabalho adaptamos nossas suítes de teste para lançar os processos de sistema referentes à infra-estrutura do SCS. Logo não é preciso lançar manualmente o nó de execução ou repositório. Isso é importante para que a suíte de testes possa monitorar os processos e interrompê-los ao final. Desenvolveu-se as seguintes suítes de teste:

1. **tests.scs.container.ComponentLoaderSuite** composta por 11 testes cujo casos:
  1. **StartContainerTest** – lança os processos de sistema do nó de execução e repositório, e então inicia o contêiner
  2. **UploadingToRepositoryTest** – testa envio ao repositório
  3. **LoadingComponentsTest** – carrega e descarrega componentes no contêiner
  4. **StopContainerTest** – para o contêiner e os processos externos do nó de execução e repositório

Para executar os testes, sem a ajuda de uma IDE de programação, basta executar o comando no diretório raiz do SCS:

```
java -Djava.endorsed.dirs=libs/jacorb -Djacorb.home=libs/jacorb \  
-Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB \  
-Dorg.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton \  
-Djava.library.path=libs \  
-classpath src/java:<instalacaoJUnit4>/junit4.jar \  
org.junit.runner.JUnitCore tests.scs.container.ComponentLoaderSuite
```

<sup>7</sup> <http://luadoc.luaforge.net>

## 5. Futuras Implementações

### 5.1. Ampliar os testes multi-plataforma e multi-linguagem

Atualmente, sabe-se que o SCS funciona como esperado em ambientes multi-plataforma e multi-linguagem, contudo não há uma metodologia organizada de testes que exercitem todas funcionalidades. A partir do conjunto de testes produzidos neste, passa a ser importante sua ampliação para contemplar o empacotamento, publicação e carga de componentes Lua e Java permutados, inclusive em outras plataformas.

### 5.2. Pacotes compartilhados entre componentes em diferentes linguagens

Na abordagem deste trabalho, as dependências de cada componente são instaladas em uma árvore de diretórios indexada pela plataforma e linguagem. Dessa forma, se, por exemplo, um componente em Java depende uma biblioteca de sistema programada em C++, tal dependência será instalada na pasta do contêiner Java. Se houver um componente em Lua que precise das mesmas bibliotecas, a instalação será duplicada na pasta do contêiner Lua. É possível uniformizar tal situação para haver uma instalação comum ao contêiner Java e Lua para casos como esse, uma vez que, o LuaRocks permite estender sua notação de plataformas.

### 5.3. Facilitar o empacotamento e publicação

O processo atual de empacotamento é o mesmo do LuaRocks e pode ser automatizado. Uma alternativa, é dispor de um componente que empacota e publica os componentes para o usuário, um **ComponentBuilder**. Assim seria possível diminuir uma porção considerável de código das aplicações finais.

### 5.4. Ferramenta de modelagem de componentes

No SCS atual, descrever um componente e suas portas (facetas e receptáculos) é muito oneroso ao programador. A adoção de uma ferramenta de modelagem que permita a definição gráfica de um componente e sua composição, pode ajudar. Tal ferramenta deveria gerar o código que efetiva as conexões entre receptáculos e carrega os componentes em seus contêiners. Essa ferramenta ainda pode ajudar no reuso em fase de projeto dos componentes pela integração ao **ComponentBuilder** e ao repositório. Uma vez descrito e empacotado, o componente pode ser publicado num repositório que outros programadores observem e, assim, a ferramenta de modelagem pode ser notificada para apresentar ao programador o mais novo componente disponível para reuso, ainda em projeto.

## 6. Considerações Finais

Neste trabalho implementa-se e descreve-se como usar um instalador e repositório de componentes. O instalador adota o LuaRocks como um subsistema para empacotamento e instalação que é exposto como uma faceta no nó de execução. O LuaRocks foi reusado integralmente com pouca mudança de código. Esse fato aumenta a confiabilidade do sistema desenvolvido uma vez que o LuaRocks é amplamente usado na comunidade de usuários/programadores Lua. Por outro lado, o repositório permite a publicação e obtenção remota dos componentes criados pelo programador, atuando como um cache de implementações. O instalador usa diretamente o repositório para obter os componentes.

Foi uma importante contribuição deste a geração da documentação de código e criação do primeiro conjunto de testes sistemáticos. Tal tarefa já gerou bons resultados, como a resolução de alguns bugs até então não percebidos. Por fim, na seção 5, há indicações de trabalhos que podem ser derivados deste.

## 7. Referências Bibliográficas

1. AUGUSTO, Carlos Eduardo Lara.  
*Uma Infra-Estrutura para Execução Distribuída de Componentes de Software*  
Dissertação de Mestrado em Informática  
Orientador Renato F. de Gusmão Cerqueira  
Rio de Janeiro, PUC-Rio, Setembro 2008
2. AUGUSTO, Carlos Eduardo Lara e CORREA, Sand.  
*Lightweight Software Component System*  
<http://www.tecgraf.puc-rio.br/~scorrea/scs>  
Rio de Janeiro, PUC-Rio, Julho 2008
3. OMG  
Common Object Request Broker Architecture  
[http://www.omg.org/technology/documents/vault.htm#CORBA\\_IOP](http://www.omg.org/technology/documents/vault.htm#CORBA_IOP)
4. OMG  
Component Model Specification  
Technical Report, 2006  
<http://www.omg.org/technology/documents/formal/components.htm>
5. BARBOSA Jr, Amadeu Andrade.  
*Implantação de Componentes de Software e Grades Computacionais: desafios compartilhados.*  
Monografia para disciplina INF2545  
<http://www.inf.puc-rio.br/~ajunior/puc/inf2545/artigos/monografia>  
Rio de Janeiro, PUC-Rio, Julho 2008
6. BARBOSA Jr, Amadeu Andrade.  
*Estudo sobre Implantação de Sistemas Distribuídos Baseados em Componentes de Software*  
Monografia para disciplina INF2556  
<http://www.inf.puc-rio.br/~ajunior/puc/inf2556/monografia.pdf>  
Rio de Janeiro, PUC-Rio, Novembro 2008