3
O Sistema de Componentes de Software

Como este trabalho lida diretamente com componentes de software, torna-se importante a escolha da arquitetura que dará suporte a esse tipo de paradigma de programação. Como queremos realizar a monitoração do ambiente de execução de componentes distribuídos da forma mais transparente possível para o usuário, certamente necessitaremos de uma arquitetura que possa ser estendida, evitando assim um grande esforço para adaptá-la aos nossos objetivos.

O modelo de componentes de software incorpora vários conceitos do paradigma de objetos, como encapsulamento e separação entre interface e implementação, mas, além disso, também torna explícito conceitos como dependências e conexões entre componentes. Mais especificamente, componentes de software podem ser definidos como unidades de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, além de poderem ser independentemente implantados e estarem sujeitos a composição por terceiros [6]. Tal separação entre a definição do componente (suas interfaces, serviços e dependências) e sua implementação permite que o mesmo possa ser manipulado como uma caixa-preta, ou seja, com base exclusivamente na sua definição. Esta, por sua vez, especifica um conjunto de conectores[1] através dos quais é possível acessar os serviços do componente, bem como fornecer os recursos por ele esperados, definidos como suas dependências.

A construção de aplicações baseadas em componentes é feita estabelecendo-se conexões entre componentes de software através da ligação de seus conectores, de forma que as dependências de um componente sejam supridas pelos serviços oferecidos por outro. Neste contexto, um middleware baseado em componentes envolve um modelo que define a estrutura dos componentes do sistema, um modelo que descreve a forma de interação entre

[1]Neste trabalho, usamos o termo *conector* para designar tanto as interfaces oferecidas quanto as requeridas por um componente

eles e a definição de um ambiente genérico para execução desses componentes.

SCS é um sistema de componentes de software projetado sobre a tecnologia CORBA, cujo objetivo é prover uma infra-estrutura de distribuição, instalação, configuração e execução de componentes. Inspirado em COM (*Component Object Model*) [20] e CCM(*CORBA Component Model*) [21], seu modelo de componentes foi idealizado visando flexibilidade, simplicidade e facilidade de uso através de um conjunto pequeno de APIs, as quais são implementadas pelos componentes de acordo com suas necessidades. Um componente em SCS é uma unidade lógica pronta para composição e reuso, podendo também encapsular um modelo de interação, configuração e introspecção.

Inicialmente, nas seções 3.1,3.2 e 3.3 apresentamos uma visão geral dos três modelos abordados pelo SCS. Já na seção 3.4 encontra-se a descrição (estrutura, conectores e dependências) da sua principal entidade, o componente SCS e, na seção 3.5, a descrição do seu ambiente de execução (mecanismo de instanciação e execução dos componentes de uma aplicação). Por fim, na seção 3.6 apresentamos detalhes das implementações Java e Lua que foram utilizadas nesta dissertação.

3.1 Modelo de Interação

Como mencionado anteriormente, um sistema de componentes constrói aplicações por composição através de portas de serviços que se conectam entre si. Portas de serviço definem os conectores através dos quais um componente pode ser interligado a outros componentes para formar um sistema. Neste sentido, SCS provê dois tipos de portas de serviços: facetas (ou portas de provisão de serviços) e receptáculos(ou portas de requisição de serviços).

Facetas são portas de provisão de serviços que oferecem uma determinada interface. Comumente, os serviços oferecidos pelo componente são disponibilizados através de um conjunto de facetas. Adicionalmente, é possível que um componente ofereça uma mesma interface através de portas distintas, mas com diferentes implementações e comportamentos. Facetas são implementadas por objetos que expõem interfaces CORBA comuns, podendo ser acessadas por clientes CORBA quaisquer.

Em SCS, uma faceta é definida por um tipo, um nome simbólico atribuído pelo desenvolvedor e o objeto CORBA que a implementa. O tipo de uma faceta,

por sua vez, consiste na interface correspondente, a qual é definida em IDL (*Interface Definition Language*). Uma vez definida uma faceta, a mesma pode ser identificada através do seu tipo ou do seu nome simbólico.

Receptáculos são portas através das quais um componente requisita um serviço, ou seja, são pontos de acesso a serviços os quais um componente depende ou utiliza. Conexões entre componentes são estabelecidas conectando-se a faceta de um componente provedor ao receptáculo de um componente requisitante do serviço.

É importante mencionar que o modelo de componentes definido para o SCS não impõe nenhuma restrição quanto à cardinalidade das conexões, ou seja, é possível que uma faceta seja conectada a nenhum ou a vários receptáculos simultaneamente. Analogamente, um receptáculo pode estar conectado a diversas implementações de objetos de uma mesma interface, situação em que o mesmo é denominado receptáculo múltiplo.

Em SCS, um receptáculo é definido por um nome simbólico atribuído pelo desenvolvedor, o nome da interface que se conecta ao receptáculo, os objetos CORBA que a implementam e que estão conectados ao receptáculo e uma propriedade para indicar se o receptáculo é múltiplo ou simples. Entretanto, uma vez definido, um receptáculo é identificado unicamente pelo seu nome simbólico.

3.2 Modelo de Configuração

O modelo de interação propicia, de forma inerente, um modelo de configuração com granularidade fina. Através de operações de conexões, componentes podem ser substituídos em tempo de execução, mudando-se a configuração das aplicações.

Adicionalmente, o conceito de facetas permite que novas interfaces sejam adicionadas estaticamente a componentes já existentes, sem contudo, afetar as aplicações que já utilizam seus serviços. Esta capacidade é especialmente interessante para o gerenciamento de versões, uma vez que é possível evoluir ou disponibilizar novas versões de um serviço sem contudo modificar os clientes de versões anteriores.

3.3
Modelo de Introspecção

O modelo de componentes de SCS define um conjunto de operações que permitem inspecionar, em tempo de execução, um componente. O desenvolvedor do componente que desejar fornecer suporte para introspecção deverá implementar a faceta *IMetaInterface*, que irá fornecer as descrições de todas as demais facetas e receptáculos oferecidos pelo componente. É importante salientar que o modelo de introspecção não é obrigatório para os componentes implementados com o SCS, como indicado anteriormente. Ele só estará disponível se o desenvolvedor optar por implementar a faceta *IMetaInterface*.

3.4
O Componente SCS

Em geral, um componente pode apresentar quantidades quaisquer de facetas e receptáculos. Porém, três interfaces específicas (vide listagem 3.1) são oferecidas pelo modelo e podem compor o comportamento de um componente SCS:

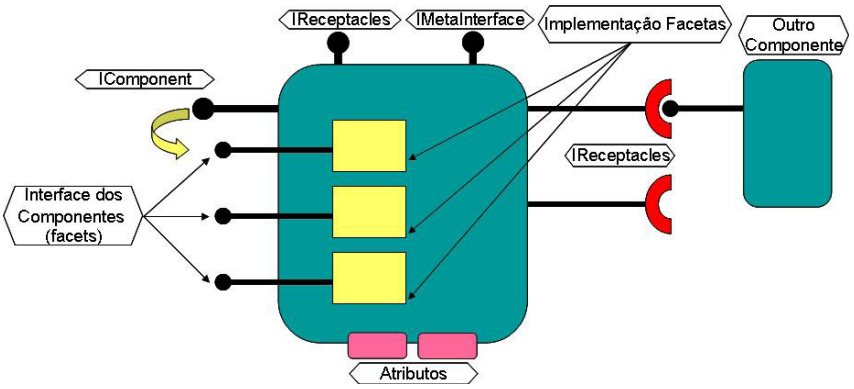

Figura 3.1: A Estrutura de um Componente SCS

– *IComponent* (linhas 6-12) define o tipo "componente"em SCS, com operações para ativação e desativação do componente, e para obter suas demais facetas. No método *startup()*, que deve ser chamado sempre no início do uso do componente, o desenvolvedor pode inicializar o ambiente necessário para o que o componente execute, realizando tarefas como conectar os receptáculos do componente a facetas de outros componentes do qual ele depende. No método *shutdown*, chamado quando

o componente for encerrar sua execução, o desenvolvedor pode garantir que todos os objetos instanciados pelo componente sejam removidos, e que as conexões feitas sejam devidamente encerradas, ou ainda informar a outros componentes que ele irá encerrar. Todo componente deve implementar pelo menos a interface *IComponent*.

- *IReceptacles* (linhas 14-22) define operações para gerenciar conexões de receptáculos, como por exemplo, métodos para conectar e desconectar uma faceta em um receptáculo e métodos para listar todos as facetas a ele conectadas. Um receptáculo pode ser configurado para receber apenas uma ou várias conexões.
- *IMetaInterface* (linhas 24-31) define operações básicas para introspecção de facetas e receptáculos do componente. Com os métodos dessa faceta o desenvolvedor pode ter acesso a estruturas que descrevem as configurações de todas as facetas e receptáculos presentes naquele componente. As estruturas retornadas e passadas para cada um dos métodos da listagem 3.1 podem ser encontradas em [13].

```
1  module scs {
2    module core {
3
4      ...
5
6      interface IComponent {
7        void startup() raises (StartupFailed);
8        void shutdown() raises (ShutdownFailed);
9        Object getFacet (in string facet_interface);
10       Object getFacetByName (in string facet);
11       ComponentId getComponentId ();
12     };
13
14     interface IReceptacles {
15       ConnectionId connect (in string receptacle, in Object obj)
16         raises (InvalidName, InvalidConnection, AlreadyConnected,
17           ExceededConnectionLimit);
18       void disconnect (in ConnectionId id)
19         raises (InvalidConnection, NoConnection);
20       ConnectionDescriptions getConnections (in string receptacle)
21         raises (InvalidName);
22     };
23
24     interface IMetaInterface {
25       FacetDescriptions getFacets();
26       FacetDescriptions getFacetsByName(in NameList names)
27         raises (InvalidName);
28       ReceptacleDescriptions getReceptacles();
29       ReceptacleDescriptions getReceptaclesByName(in NameList names)
30         raises (InvalidName);
31     };
32   };
33 };
```

Listagem 3.1: Facetas do Componente SCS

3.5 Ambiente de Execução

O Ambiente de Execução no SCS consiste em um mecanismo padrão para instanciação, carga, configuração e execução dos componentes do sistema. Dois elementos principais formam tal mecanismo: *container* e nó de execução.

3.5.1 *Container*

Um *container* é uma abstração que define um ambiente de execução protegido, onde implementações de componentes são implantadas, criadas e executadas. Interações entre a implementação de um componente e o mundo externo são intermediadas pelo *container*, o qual oferece serviços e funcionalidades para implementações de componentes. Isto promove também uma forma padronizada para implantar, criar, ativar e fornecer serviços para implementações de componentes. De maneira geral, podemos listar as seguintes responsabilidades para um *container*:

- Criação de componentes: Um *container* deve ser capaz de construir uma instância inteiramente funcional de um componente, a partir da sua implementação.
- Ativação de componentes: Um *container* deve ser capaz de ativar e desativar um componente de acordo com determinadas políticas, bem como controlar o estado das conexões do componente.
- Definição de políticas para controle de qualidade de serviço: *containers* diferentes podem ser definidos para tratar ou garantir diferentes tipos de qualidade de serviço, como tolerância a falhas ou reserva de recursos.
- Acesso a serviços CORBA: Um *container* deve prover acesso aos serviços de objetos CORBA (como eventos, transações, persistência, etc), fornecendo assim um mecanismo padrão de obtenção desses serviços, bem como compartilhamento dos mesmos visando economia de recursos.

Em SCS, *containers* são implementados através de componentes denominados *Container* (figura 3.2). Tais componentes definem um espaço de endereçamento para os componentes de um serviço, isolando-os dos componentes de outros serviços. Em geral, componentes de um serviço SCS executam no

mesmo espaço de endereçamento do seu componente *Container*, o qual possui dois tipos de facetas. A faceta *ComponentLoader* oferece as operações para o carregamento de componentes, enquanto a faceta *ComponentCollection* permite consultar os componentes carregados no *container*.

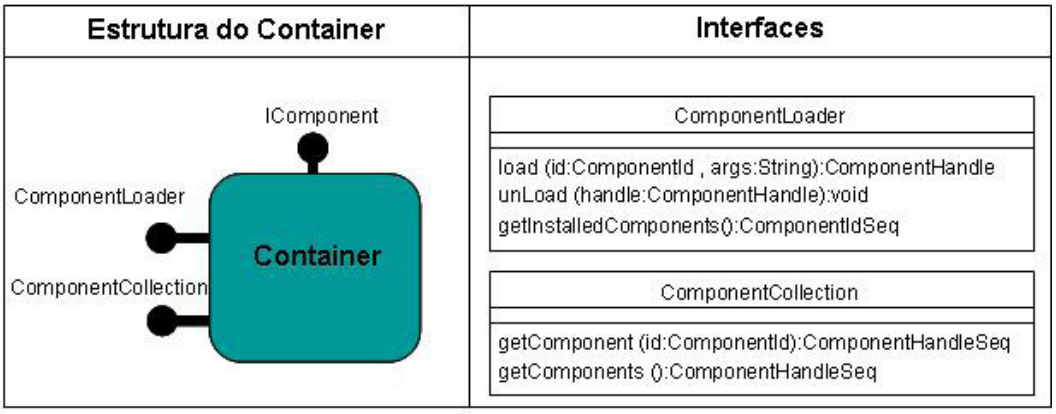

Figura 3.2: *Container* SCS

É importante mencionar também que um *container* SCS gerencia implementações de componentes em uma mesma linguagem de programação. Dessa forma, todos os componentes em um *container* são implementados em uma mesma linguagem. Essa decisão de projeto visa garantir simplicidade no uso de um *container* SCS, visto que *containers* multi-linguagens são utilizados raramente e não justificam sua complexidade adicional.

3.5.2 Nós de Execução

Num ambiente de execução, um nó de execução corresponde a um dispositivo físico *host* onde *containers* executam. Em SCS, um nó de execução é representado por um componente denominado *ExecutionNode* (figura 3.3), cuja finalidade é gerenciar os *containers* nele instanciados. Para tanto, este componente implementa uma faceta, também denominada ExecutionNode, a qual oferece operações para disparar novos *containers* e consultar os *containers* em execução. A seguir definimos tal interface com mais detalhes.

A listagem 3.2 representa a IDL do nó de execução, na qual estão definidas suas as operações e atributos. Dentre as operações existentes destacamos a *startContainer* que é responsável por inicializar um novo *container*. O parâmetro *props* pode definir um conjunto de propriedades que caracterizem o *container* que deve ser disparado (como, por exemplo, qual tipo de linguagem

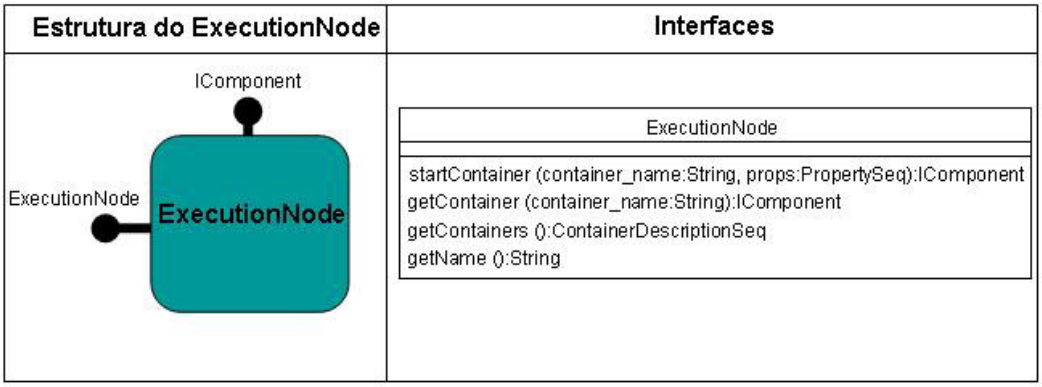

Figura 3.3: *container* SCS

os componentes carregados nele devem possuir, C++, Lua ou Java), assim como eventuais argumentos que devem ser passados para o mesmo durante a sua inicialização.

```
1  module scs {
2     ...
3
4    module execution_node {
5      exception ContainerAlreadyExists{};
6      exception InvalidContainer{};
7      exception RequirementNotMet{ string reason; };
8
9      struct Property {
10        string name;
11        string value;
12        boolean read_only;
13     };
14     typedef sequence<Property> PropertySeq;
15
16     struct ContainerDescription {
17        core::IComponent container;
18        string container_name;
19        core::IComponent execution_node;
20     };
21     typedef sequence<ContainerDescription> ContainerDescriptionSeq;
22
23     interface ExecutionNode {
24        core::IComponent startContainer (in string container_name,
25                                         in PropertySeq props)
26           raises (ContainerAlreadyExists);
27        core::IComponent getContainer (in string container_name);
28        ContainerDescriptionSeq getContainers ();
29        string getName();
30     };
31
32     interface ContainerManager {
33           void registerContainer(in string name, in core::IComponent ctr)
34                raises (ContainerAlreadyExists,    InvalidContainer);
35           void unregisterContainer(in string name)
36                raises (core::InvalidName);
37         };
38   };
```

```
39
40      ...
41
42  };
```

Listagem 3.2: Execution Node

3.6 Implementações do SCS

O SCS é uma arquitetura que se propõe a lidar com diversas linguagens, porém, como foi mencionado na seção 3.5.1 um *container* não é multi-linguagem, logo, é necessário existir um *container* para cada linguagem de programação com a qual se deseje implementar um componente.

No caso deste trabalho, utilizamos duas implementações do SCS, uma em Java e outra em Lua. Cada uma dessas implementações possui não só o *container* como também todos os demais componentes implementados para a linguagem. As duas implementações seguem a mesma IDL, de modo que existem pequenas diferenças entre elas devido a especificidades de cada linguagem, e que não são percebidas pelo usuário. Isso permite que aplicações SCS usem diversos componentes implementados em diferentes linguagens de modo transparente.

O SCS Java foi implementado baseado no ORB do JDK1.5 da SUN [22]. Para facilitar o desenvolvedor da aplicação, esta versão oferece algumas classes básicas que implementam os principais comportamentos das classes base do SCS.

O SCS Lua foi baseado no OiL [23], uma implementação da especificação CORBA para linguagem Lua. O OiL compartilha as principais características de Lua, como simplicidade, eficiência, portabilidade e flexibilidade (devido às características dinâmicas de linguagens de *script*)[24].