2
A Infra-Estrutura SCS

O SCS [7] é uma infra-estrutura para dar apoio à instalação, carga e execução de componentes de software em um ambiente distribuído. Atualmente, é fornecida uma implementação do modelo e uma biblioteca de suporte à programação para as linguagens Lua [22, 23], Java [15] e C++. Diferentemente de outros trabalhos relacionados mais complexos e pesados, o SCS visa reunir em sua arquitetura apenas as funcionalidades críticas necessárias para a criação de um sistema orientado a componentes [24], disponibilizando interfaces pequenas e simples de usar. Além disso, o SCS já é utilizado como ferramenta de ensino e para experimentação de novas técnicas relacionadas a componentes de software no nosso grupo de pesquisa. Sendo assim, o SCS é a infra-estrutura ideal para utilizarmos como base deste trabalho.

Este capítulo visa apresentar os principais conceitos e fundamentos do SCS, assim como seu modelo de componentes e seu modelo de programação. Também serão apresentados os principais componentes que compõem a sua arquitetura de implementação.

2.1
Modelo de Componentes

Em sistemas baseados em componentes de software, o modelo de componentes define as entidades abstratas que compõem um componente, assim como o relacionamento entre elas.

O modelo de componentes original do SCS é composto por três entidades principais: o componente, a faceta e o receptáculo. As facetas e receptáculos são as interfaces de comunicação entre os componentes, onde a faceta representa a interface de um conjunto de operações publicadas como serviços pelo componente, e o receptáculo representa a interface de um conjunto de operações necessárias para o funcionamento do componente. Todo componente SCS possui uma faceta obrigatória chamada *IComponent*, através da qual se tem acesso a todas as facetas e receptáculos do componente. Desta forma, tem-se uma interface única compartilhada entre componentes. A figura 2.1 ilustra um componente SCS.

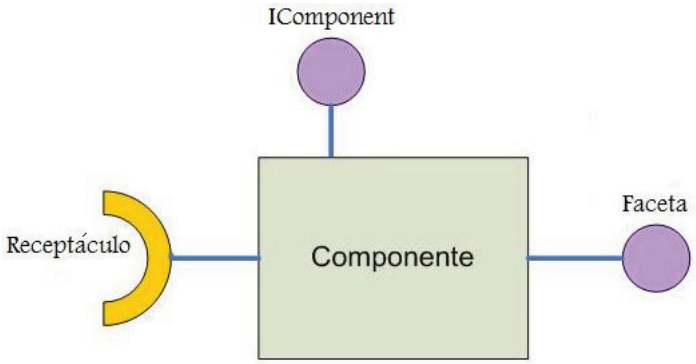

Figura 2.1: Componente SCS

Tipicamente, os serviços publicados por uma faceta de um componente devem ser consumidos por um receptáculo de outro componente. Sendo assim, o receptáculo é o ponto de conexão de um componente com outros componentes. A figura 2.2 ilustra essa idéia.

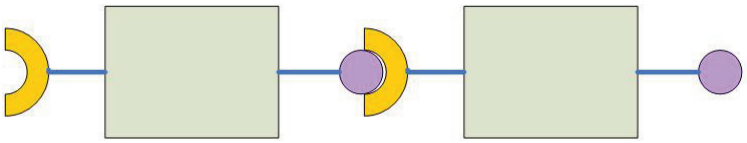

Figura 2.2: Conexão de Componentes SCS

O modelo de distribuição de objetos do SCS é baseado na arquitetura CORBA [25, 26], e tem suas interfaces definidas através de arquivos de definição IDL, segundo o padrão definido pelo *Object Management Group* [27].

A figura 2.3 exemplifica como que, na prática, ocorrem as conexões entre componentes SCS em um ambiente distribuído, onde uma faceta de um componente pode ser conectada a múltiplos receptáculos de diversos componentes, assim como um receptáculo também pode ser conectado a várias facetas. Um receptáculo que pode receber múltiplas conexões é denominado receptáculo multiplex.

A seguir, então, são detalhadas as principais entidades do modelo SCS. A figura 2.4 mostra o relacionamento entre elas.

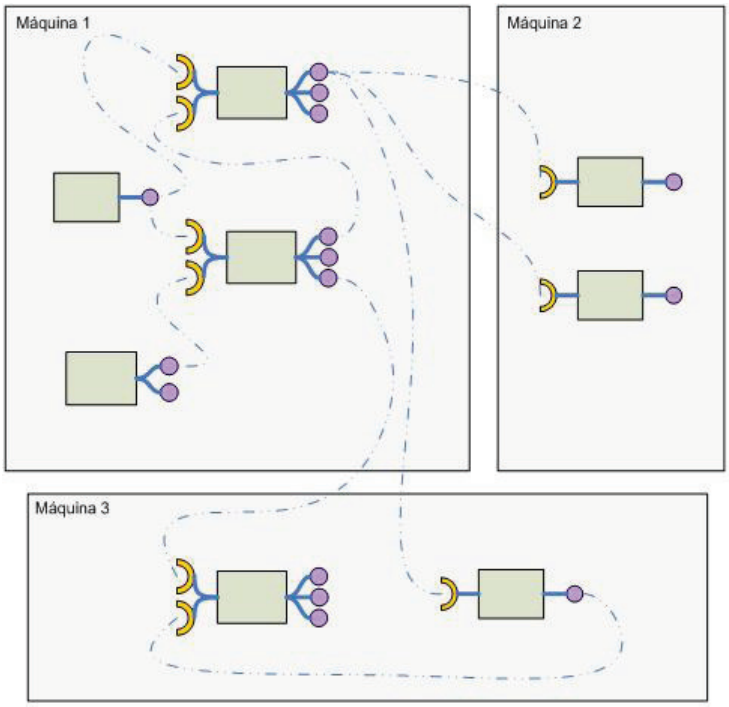

Figura 2.3: Conexões SCS

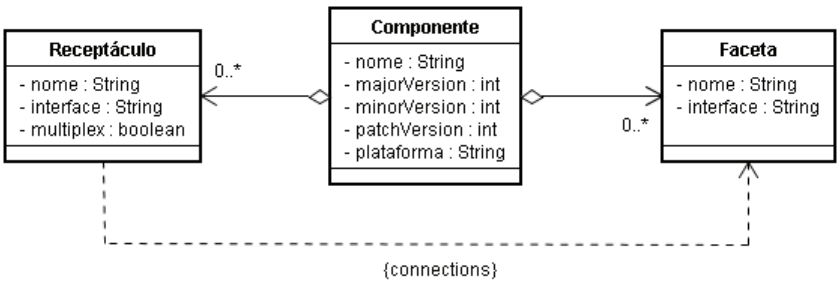

Figura 2.4: Principais Entidades e Relacionamentos do Modelo SCS

Onde:

* Componente - Representa um componente SCS que contém um conjunto de facetas e receptáculos. Suas propriedades são apenas descritivas.

  nome - Representa o nome identificador do componente.

  major, minor e patch versions - Números identificadores da versão do componente.

  plataforma - Indica a plataforma relativa à implementação do componente. (Exemplo: Java 1.6)

* Faceta - Representa um conjunto de operações publicadas pelo componente.

  nome - Nome identificador da faceta no ambiente distribuído.

interface - Indica o arquivo IDL com a descrição das operações publicadas pela faceta.

* Receptáculo - Representa uma referência a um objeto remoto, necessário para o funcionamento do componente.

  nome - Nome identificador do receptáculo no ambiente distribuído.

  interface - Indica o arquivo IDL com a descrição das operações publicadas pela faceta que deve ser conectada ao receptáculo.

  multiplex - Indica se o receptáculo suporta múltiplas conexões simultâneas.

Componentes SCS são encapsulados em contêineres, que fornecem um ambiente de execução permitindo o acesso, a ativação e a desativação desses componentes.

2.2 Modelo de Programação

Um framework que dá suporte à programação orientada a componentes normalmente é uma instanciação ou implementação de um modelo de componentes. Esta instanciação será então responsável por tornar concretas as abstrações propostas pelo modelo de componentes, fornecendo uma API em uma determinada linguagem de programação para dar apoio à manipulação de componentes de software em ambientes distribuídos.

Atualmente, o SCS [7] possui suporte às linguagens Java, C++ e Lua, oferecendo um framework com um modelo de programação específico para cada uma delas. Neste trabalho iremos focar no modelo de programação Java do SCS.

O framework SCS-java oferece um conjunto de classes principais com as quais o desenvolvedor do componente deve interagir para criar, carregar, implantar, e executar o componente. O conjunto dessas classes forma uma API de introspecção disponível para o desenvolvedor nas diferentes etapas de construção da aplicação. A seguir são apresentadas as principais classes:

* ComponentId

  Agrega informações descritivas sobre o componente como o nome, a versão e a plataforma de implementação.

* IComponent

  Faceta obrigatória presente em todos os componentes SCS. Através das operações *getFacet()* e *getFacetByName()* presentes no IComponent,

é possível ter acesso a todas as facetas publicadas pelo componente. Através da operação *getComponentId* é possível ter acesso às informações descritivas do componente.

* IReceptacles

  Faceta utilizada pelo desenvolvedor da aplicação para conectar facetas e receptáculos de componentes SCS. Apresenta as operações *connect()*, *disconnect()* e *getConnections*.

* IMetaInterface

  Faceta que permite a introspecção no componente. É possível, por exemplo, obter as descrições das facetas e receptáculos do componente através dos métodos *getFacets()* e *getReceptacles()*.

* ComponentContext

  Representa o contexto de um componente. O contexto do componente encapsula o IComponent e disponibiliza mapas para acesso a informações como as descrições de todas as facetas e receptáculos do componente.

* ExtendedFacetDescription

  Armazena o nome, o identificador da interface definida no arquivo IDL, e a classe de implementação de uma faceta de um componente SCS.

* ReceptacleDescription

  Armazena as informações de um receptáculo de um componente SCS. Através desta classe é possível obter o nome, o identificador da interface definida no arquivo IDL, as conexões relacionadas ao receptáculo e um atributo informando se o receptáculo permite múltiplas conexões.

* ComponentBuilder

  Responsável por criar o contexto do componente baseado no *ComponentId* e no conjunto de descrições de facetas e receptáculos do componente.

* ComponentFactory

  Interface relacionada ao contêiner SCS, normalmente implementada pelo desenvolvedor do componente, que é responsável por criar o contexto do componente SCS através do *ComponentBuilder*. Disponibiliza as operações *create()* e *destroy()*.

* ComponentLoader

Faceta relacionada ao contêiner SCS que permite ativar e desativar componentes. Possui as operações *load()*, *unload()* e *getInstalledComponents()*.

2.3 Construindo uma Aplicação SCS

Esta seção apresenta um passo-a-passo para a construção de uma aplicação SCS. A seção é dividia em três etapas: definição dos componentes através da criação do arquivo IDL, implementação, e implantação.

O exemplo apresentado nesta seção é uma aplicação *HelloWorld*. O componente *Hello* apresenta uma faceta com uma operação que retorna uma mensagem de "HelloWorld", enquanto o componente *HelloPrinter*, através de um receptáculo, consome o serviço desta faceta, imprimindo o retorno no console.

2.3.1 Definição (Arquivo IDL)

O primeiro passo para a criação de uma aplicação SCS é a definição das interfaces dos componentes através da confecção do arquivo IDL. Este arquivo, como mostrado no trecho de código abaixo, descreve a interface pública de todos os serviços que serão publicados pelos componentes.

Código 2.1: IDL - HelloWorld

```
1 module scs{
2   module demos{
3    module ascs{
4     module helloworld {
5
6       interface Hello {
7         string sayHello();
8       };
9
10      interface HelloPrinter {
11        void printHello();
12      };
13
14    };
15   };
16  };
17 };
```

O trecho a seguir exemplifica a linha de comando utilizada para compilar o arquivo idl, gerando as classes java necessárias para o desenvolvimento dos componentes.

Código 2.2: IDL - Compilação

```
1 idlj -td diretorio_de_destino -fallTIE .../arquivo_idl.idl
```

2.3.2 Implementação

Em um middleware CORBA, um *servant* é a classe que contém a implementação de um componente e que estende um objeto POA.

Após a compilação do arquivo IDL, já podemos codificar os *servants* dos componentes utilizando o modelo de programação do SCS. O trecho de código mostrado na figura 2.3 exemplifica a implementação do *servant* do componente *HelloPrinter* definido na IDL anteriormente.

Código 2.3: Servant SCS

```
1  /**
2   * Componente que implementa o serviço de exemplo
3   * consultando a faceta do componente HelloComponent
4   * e imprimindo HelloWorld! no console.
5   */
6  public class HelloPrinterServant implements HelloPrinterPOA {
7
8    private ComponentContext myComponent = null;
9
10     public HelloPrinterServant(ComponentContext myComponent) {
11       this.myComponent = myComponent;
12   }
13
14   @Override
15   public void printHello() {
16     ArrayList<ConnectionDescription> conns =
17           this.myComponent.getReceptacles().get(HelloPrinterFactory.
                 FACET_PRINTER)
18                .getConnections();
19
20     for (ConnectionDescription desc : conns) {
21           Hello helloFacet = HelloHelper.narrow(desc.objref);
22       System.out.println(helloFacet.sayHello());
23     }
24
25   }
26
27   @Override
28   public org.omg.CORBA.Object _get_component() {
29         return myComponent.getIComponent();
30   }
31
32 }
```

A classe de implementação de um componente SCS precisa estender a classe POA correspondente que foi gerada a partir da IDL, para então, sobrescrever os métodos a serem implementados. No caso, o método a ser implementado é o *printHello()*).

Para criar um *servant* de um componente SCS, é preciso sobrescrever o método *_get_component()* da classe POA para que o núcleo do SCS seja capaz de objter o objeto *IComponent* relativo ao componente. Também é preciso definir um construtor específico que recebe o contexto do componente conforme

demonstrado no exemplo. Esta restrição existe para que o componente possa ser instanciado em tempo de execução através de mecanismos de reflexão computacional.

Na implementação do método *printHello()* é realizada uma iteração entre todos os objetos remotos conectados ao receptáculo e para cada um é realizada a chamada remota ao método *sayHello()* imprimindo o retorno no console.

Após a implementação do *servant* do componente, o desenvolvedor precisa implementar a fábrica responsável por criar o componente. Esta fábrica será utilizada pelo núcleo do SCS no momento da implantação do componente. A implementação da fábrica é feita através da extensão da classe *ComponentFactory* e da implementação do método *create()*. A implementação deste método consiste na criação do *ComponentId* e dos objetos que representam as informações das facetas e receptáculos do componente, e então a utilização da classe *ComponentBuilder* para construir o componente de fato. O trecho de código mostrado na figura 2.4 exemplifica uma fábrica relativa ao componente implementado na figura 2.3.

Código 2.4: Factory SCS

```
1  public ComponentContext create(ComponentId id, ComponentBuilder builder,
2      String[] args, String factoryArg) throws LoadFailure {
3      try {
4        ExtendedFacetDescription[] extDescs = new ExtendedFacetDescription
             [1];
5        ReceptacleDescription[] recepDescs = new ReceptacleDescription[1];
6        ExtendedFacetDescription printerExtDesc =
7          new ExtendedFacetDescription(FACET_PRINTER, IFACE_PRINTER,
               CLASS_PRINTER);
8        extDescs[0] = printerExtDesc;
9
10       ReceptacleDescription printerReceptDesc =
11         new ReceptacleDescription(FACET_PRINTER, HelloHelper.id(), false,
               null);
12       recepDescs[0] = printerReceptDesc;
13
14       ComponentId cpId =
15         new ComponentId(componentName, majorVersion, minorVersion,
16           patchVersion, platformSpec);
17       ComponentContext instance =
18         builder.newComponent(extDescs, recepDescs, cpId);
19       return instance;
20     }
21     catch (Exception e) {
22       e.printStackTrace();
23       throw new LoadFailure("Error creating instance!");
24     }
25   }
```

2.3.3 Implantação

Após a codificação dos dois componentes definidos no arquivo IDL e de suas fábricas, o próximo passo, já relativo à implantação do componente, é criar os arquivos de descrição. A figura abaixo mostra como deve ser o arquivo de descrição para o componente *HelloPrinter*.

Código 2.5: Arquivo de descrição do componente HelloPrinter

```
1 name = aHelloPrinter
2 major_version = 1
3 minor_version = 0
4 patch_version = 0
5 platform_spec = none
6 entry_point = scs.demos.helloworld.servant.HelloPrinterFactory
7 extension = class
```

O campo entry_point corresponde à fábrica que será utilizada para criar o componente. No caso do exemplo, esse campo deve ser configurado com o caminho completo da classe codificada na figura 2.4.

O nome dos arquivos de descrição deve obrigatoriamente seguir um padrão onde os três números identificadores da versão seguem o nome do componente. Os arquivos devem possuir a extensão ".desc". Neste caso, o arquivo deve ser nomeado HelloPrinter100.desc. Esta restrição existe pois o SCS carregará automaticamente estes arquivos usando como base as informações contidas no id do componente.

Após a codificação e a definição do arquivo de descrição, o último passo é o desenvolvimento da aplicação que usará os componentes. A classe demonstrada a seguir será responsável por carregar os componentes no contêiner SCS, realizar a amarração entre facetas e receptáculos e por fim fazer a chamada remota ao método do componente HelloPrinter. No trecho a seguir, os tratamentos de exceção foram omitidos a fim de simplificar o código do exemplo. Recomenda-se que as exceções sejam tratadas uma a uma.

Código 2.6: Aplicação de exemplo SCS

```
1 /**
2  * Classe que implementa a aplicação do exemplo HelloWorld, responsável por
3  * carregar os componentes no contêiner, realizar a amarração entre facetas
       e
4  * receptáculos e por fim fazer a chamada remota ao método do componente
5  * HelloPrinter.
6  */
7 public class HelloApp {
8
9   /**
10   * Função main.
11   * @param args Contém o IOR do componente em args[0]
12   */
13   public static void main(String[] args) throws Exception {
```

```
14      ORB orb = ORB.init(args, null);
15
16      // Obtém o componente que representa o contêiner através do IOR.
17      IComponent container = IComponentHelper
18          .narrow(orb.string_to_object(args[0]));
19
20      container.startup();
21
22      ComponentLoader loader = ComponentLoaderHelper.narrow(container
23          .getFacet("IDL:scs/container/ComponentLoader:1.0"));
24      if (loader == null) {
25        System.out.println("component loader retornado == null !!");
26        return;
27      }
28
29      // Constrói o ComponentId relativo ao componente Hello
30      ComponentId helloCompId = new ComponentId("Hello", (byte) 1, (byte) 0,
31          (byte) 0, "none");
32
33      // Constrói o ComponentId relativo ao componente HelloPrinter
34      ComponentId printerCompId = new ComponentId("HelloPrinter", (byte) 1,
35          (byte) 0, (byte) 0, "none");
36
37      ComponentHandle helloHandle = null;
38      ComponentHandle printerHandle = null;
39
40      // Carrega o componente Hello no Contêiner
41      helloHandle = loader.load(helloCompId, new String[] { "" });
42      System.out.println("HelloComponent carregado com sucesso.");
43
44      // Carrega o componente HelloPrinter no Contêiner
45      printerHandle = loader.load(printerCompId, new String[] { "" });
46      System.out.println("HelloPrinterComponent carregado com sucesso.");
47
48      helloHandle.cmp.startup();
49      printerHandle.cmp.startup();
50
51      // Obtém o objeto remoto relativo à faceta do componente HelloPrinter
52      HelloPrinter printer = HelloPrinterHelper.narrow(printerHandle.cmp
53          .getFacetByName(HelloPrinterServant.class.getSimpleName()));
54
55      IReceptacles rcpt = IReceptaclesHelper.narrow(printerHandle.cmp
56              .getFacetByName(IReceptacles.class.getSimpleName()));
57
58      org.omg.CORBA.Object o = HelloHelper.narrow(helloHandle.cmp
59          .getFacetByName(HelloServant.class.getSimpleName()));
60
61      // Conecta o receptáculo do componente HelloPrinter à faceta do
            componente Hello.
62      rcpt.connect(HelloPrinterServant.class.getSimpleName(), o);
63
64      // Realiza a chamada remota do método printHello()
65      printer.printHello();
66    }
67  }
```