

# Tutorial Básico do SCS C++ Orbix

Tecgraf

Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

scs-users@tecgraf.puc-rio.br

2011-08-31

## 1 Introdução

Este documento é um tutorial básico sobre a criação de componentes no modelo SCS v1.2, utilizando a versão C++ 1.2.0\_0. Não serão encontradas aqui explicações sobre o modelo, as quais encontram-se em documentos específicos. Também não será abordado o uso de serviços específicos desenvolvidos para o auxílio ao uso do modelo, como a infra-estrutura de execução. Essas informações também podem ser obtidas em outros documentos. A implementação SCS-C++ aqui apresentada baseia-se em CORBA v2.3, representada pelo ORB Orbix v6.3. Este documento assume que o leitor é familiarizado a conceitos de desenvolvimento de *software* baseado em componentes e à terminologia CORBA.

## 2 Inicialização do ORB

Para a criação e execução do código de um componente, é necessária a inicialização prévia de um ORB. A instância de ORB criada será passada por parâmetro posteriormente para o construtor de um componente SCS. Esse procedimento será demonstrado em mais detalhes adiante.

### 3 Contexto de Componente

Todo componente SCS-C++ é representado por seu "contexto". Um Contexto de Componente (classe *ComponentContext*) atua como um envólucro para as facetas e receptáculos de um componente SCS, e fornece acesso interno a essas mesmas facetas e receptáculos. Além disso, concentra também o acesso ao identificador do componente, ao objeto CORBA da faceta *IComponent*. A interface do tipo *ComponentContext* possui funções e tipos para lidar com: Receptáculos, Facetas, *ComponentId* e controle de concorrência.

Iremos mostrar somente as assinaturas das funções de acesso para receptáculos e facetas.

Código 1: *ComponentContext* interface

```
1 class ComponentContext
2 {
3     [...]
4     std::map<std::string, CORBA::Object_var> const& getMapFacets() const;
5
6     std::map<std::string, Facet> const& getMapFacetsByName() const;
7
8     std::map<std::string, Receptacle> const& getMapReceptacles() const;
9
10    std::map<std::string, Receptacle>& getMapReceptacles();
11
12    void addFacet(const char* name, const char* interface_name, PortableServer::ServantBase* servant);
13
14    void addFacet(const char* name, const char* interface_name
15                , std::auto_ptr<PortableServer::ServantBase>& servant);
16
17    void updateFacet(const char* name, PortableServer::ServantBase* servant);
18
19    void updateFacet(const char* name, std::auto_ptr<PortableServer::ServantBase>& servant);
20
21    void addReceptacle(const char* name, const char* interface_name, bool multiplexed = false);
22
23    void removeFacet(const char* name);
24
25    void deactivateFacet(const char* name);
26
27    Facet getFacetByName(const char* name);
28    [...]
29 };
```

Essas funções podem ser vistas com mais detalhe na documentação de referência (Doxygen).

Para fazer controle de concorrência, o *ComponentContext* possui dois tipos-membros: *receptacles\_lock* e *facet\_lock*. E usa-se da seguinte forma:

## Código 2: Primitivas de Lock

---

```
1 void foo(scs::core::ComponentContext& component)
2 {
3     scs::core::ComponentContext::facet_lock lock(component);
4     // usar funções que precisam ser serializadas para o estado
5     // das facetas do componente
6 }
```

Quando o objeto lock sair de escopo, seu destrutor automaticamente fará o unlock do mutex responsável por serializar esse estado. Recomendamos esse método de programação pois garante o destrancamento do mutex mesmo em vista de exceções.

## 4 Passos Necessários à Criação de um Componente

Aqui serão descritos os passos mínimos necessários para a criação de um componente SCS-C++.

### 4.1 Definição do Identificador do Componente

O identificador do componente é uma estrutura definida em IDL (scs.idl) chamada `ComponentId`, e representada em C++ pela classe `ComponentId`. Um identificador de componente conta com os seguintes campos:

- `name`: Nome desejado para o componente.
- `major_version`: Octeto que define o número principal da versão do componente.
- `minor_version`: Octeto que define a versão secundária do componente, possivelmente relacionado a uma sub-versão da versão principal.
- `patch_version`: Octeto que define a versão de revisão do componente.
- `platform_spec`: *String* contendo quaisquer especificações de plataforma necessárias ao funcionamento do componente.

Os números de versão do componente, quando unificados, devem ser separados por pontos. Ou seja, um componente com versão principal 1, versão secundária 0 e versão de revisão 0 deve ser representado como a *String* "1.0.0".

## 4.2 Criação de Facetas

Facetas são interfaces CORBA, e devem ser implementadas por classes definidas pelo usuário, como exigido pelas definições C++ desse padrão.

Código 3: Implementação de uma Faceta MyFacet - Interface

```
1 class MyFacetImpl : virtual public POA_MyFacet {  
2     [...]  
3 };
```

Obviamente, facetas devem ainda implementar seus métodos definidos em IDL.

Nas outras linguagens do SCS (Java e Lua), existe o suporte ao método `_get_component()` de CORBA, que permite acessar a faceta `IComponent` a partir de qualquer outra faceta. Como o Orbix atualmente não dá suporte a esse método, o seu uso não é possível nessa versão do SCS-C++.

## 4.3 Criação do Componente Básico

Para construir um componente básico, que possui as três facetas básicas: `IComponent`, `IReceptacles`, `IMetaInterface`, é só construir o mesmo com o construtor de `ComponentContext`. Esse possui duas sobrecargas. Uma aceita um POA, que pode ser passado o `RootPOA`, e outro que não recebe nenhum POA e usa o `RootPOA` implicitamente.

Todas as duas sobrecargas exigem os parâmetros `ORB` e `ComponentId`.

#### Código 4: Instanciação de um Novo Componente

---

```
1 ORB orb = ...; // referência para o ORB, já inicializado
2 POA poa = ...; // referência para o POA
3
4 // component id
5 scs::core::ComponentId componentId;
6 componentId.name = "MyComponent";
7 componentId.major_version = '1';
8 componentId.minor_version = '0';
9 componentId.patch_version = '0';
10 componentId.platform_spec = "none";
11
12 // utilização da API
13 scs::core::ComponentContext context (orb, poa, componentId);
14 context.addFacet("MyFacet", "IDL:mymodule/MyFacet:1.0", new MyFacetImpl);
15 context.addReceptacle("MyReceptacle", "IDL:expectedmodule/ExpectedFacet:1.0");
```

## 5 Exemplo Completo

Demonstraremos aqui o uso mais simples para um componente: apenas uma faceta além das três facetas básicas. Não será criado nenhum receptáculo, apesar da existência da faceta IReceptacles. Exemplos mais complexos poderão ser encontrados nas *demos* do projeto.

Esta demonstração será baseada na demo *Hello*, que implementa um componente carregável em contêiner (parte da infra-estrutura de execução). O código apresentado a seguir é uma versão modificada dessa demo, para que possa ser carregado manualmente, sem o uso de um contêiner.

O componente Hello tem quatro interfaces: IComponent, IReceptacles, IMetaInterface e apenas uma interface própria, de nome IHello. Sua IDL está disponível no Código 5.

#### Código 5: IDL do Componente Hello

---

```
1 module scs{
2   module demos{
3     module helloworld {
4       interface IHello {
5         void sayHello();
6       };
7     };
8   };
9 };
```

Para implementar a faceta IHello, que conta com apenas um método, *sayHello*,

criamos a classe `HelloImpl`, que pode ser visualizada no Código 6. O código é bastante similar ao apresentado no Código 3.

#### Código 6: A Faceta `IHello`

---

```
1 #include <iostream>
2 #include <omg/orb.hh>
3 #include <scs/core/ComponentContext.h>
4 #include "stubs/helloS.hh"
5
6 class HelloImpl : virtual public POA_demoidl::hello::IHello {
7     void sayHello() IT_THROW_DECL((CORBA::SystemException)) {
8         std::cout << std::endl << "Hello World!" << std::endl;
9     };
10 };
```

Além da implementação da faceta, é necessário um código de criação de componente. Esse código, que tipicamente será incluído na função *main* do programa, é muito similar ao do Código 4 e pode ser conferido no Código 7.

Por fim, temos o código "cliente", que acessa o componente. Note que esse código pode ser CORBA puro, não é necessária a criação de um componente para acessar outro componente. Um exemplo desse tipo de código pode ser visto no Código 8.

Neste exemplo, a mensagem "Hello World!" será exibida somente na máquina servidor. O código cliente apenas terá a chamada *sayHello()* completada corretamente e será finalizado sem erros.

## Código 7: Criação do Componente Hello

---

```
1 #include <omg/orb.hh>
2 #include <it_ts/thread.h>
3 #include <scs/core/ComponentContext.h>
4
5 int main(int argc, char* argv[]) {
6     // inicialização do ORB
7     CORBA::ORB* orb = CORBA::ORB_init(argc, argv);
8     CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
9     PortableServer::POA* poa = PortableServer::POA::_narrow(poa_obj);
10    PortableServer::POAManager_var poa_manager = poa->the_POAManager();
11    poa_manager->activate();
12
13    // criação do ComponentId
14    scs::core::ComponentId componentId;
15    componentId.name = "HelloComponent";
16    componentId.major_version = '1';
17    componentId.minor_version = '0';
18    componentId.patch_version = '0';
19    componentId.platform_spec = "none";
20
21    // instanciação do componente
22    scs::core::ComponentContext component (orb, componentId);
23
24    // adiciona faceta no componente
25    component.addFacet("IHello", "IDL:demoidl/hello/IHello:1.0", new HelloImpl);
26
27    // publicação do IOR para que a faceta IHello do componente possa ser
28    // encontrada. Observação: precisamos exportar a faceta IComponent, pois não
29    // temos o método _get_component para obter a faceta IComponent (esse passo
30    // pode ser substituído por outras formas de publicação, como a publicação em
31    // um serviço de nomes, por exemplo).
32    // Assume-se que writeToFS seja uma função que escreva uma string em
33    // um arquivo.
34    scs::core::IComponent_var icObj = component.getIComponent();
35    std::string icIOR = orb->object_to_string(icObj);
36    writeToFS(icIOR, "hello.ior");
37
38    // instrução ao ORB para que aguarde por chamadas remotas
39    orb->run();
40
41    return 0;
42 }
```

## Código 8: Utilização do Componente Hello

---

```
1 #include <string.h>
2 #include <iostream>
3 #include "stubs/hello.hh"
4
5 int main(int argc, char* argv[]) {
6     // inicialização do ORB
7     CORBA::ORB* orb = CORBA::ORB_init(argc, argv);
8
9     // assume-se que o arquivo que contém o IOR (publicado no código
10    // anterior) esteja disponível. O arquivo pode ter sido criado em
11    // outra máquina e, nesse caso, tem de ser copiado manualmente
12    // (pode-se também utilizar um método diferente de publicação,
13    // como um serviço de nomes).
14    // Assume-se que readFromFile seja uma função que leia o conteúdo
15    // de um arquivo para uma String.
16    CORBA::Object_var objIC = orb->string_to_object(readFromFile("hello.ior"));
17
18    // obtenção das facetas IHello e IComponent
19    // precisamos utilizar o método narrow pois estamos recebendo um
20    // org.omg.CORBA.Object
21    scs::core::IComponent_var iComponentFacet =
22        scs::core::IComponent::_narrow(objIC);
23    CORBA::Object_var objHello =
24        iComponentFacet->getFacet("IDL:demoidl/hello/IHello:1.0");
25    demoidl::hello::IHello* helloFacet = demoidl::hello::IHello::_narrow(objHello);
26
27    // inicialização do componente.
28    iComponentFacet->startup();
29
30    // com o componente inicializado, podemos utilizá-lo à vontade.
31    helloFacet->sayHello();
32
33    return 0;
34 }
```