3
Conceitos Básicos

Este capítulo apresenta conceitos importantes para o entendimento da extensão do middleware proposta mais adiante. O conhecimento dos principais conceitos por trás do SCS e das principais características dos Conectores de Fluxo de Dados é primordial para esse objetivo. Portanto, na próxima seção é apresentada uma visão geral do SCS e das infraestruturas de execução e implantação. Na seção seguinte, é dada uma descrição das características relevantes aos CFDs. Por fim, a última seção apresenta algumas considerações finais sobre o conteúdo mencionado.

3.1
Modelo de Componentes SCS

O SCS [15, 16] é um sistema de componentes de software desenvolvido na PUC-Rio, frequentemente utilizado em pesquisas acadêmicas e na construção de sistemas comerciais [28]. Ele oferece um modelo de componentes distribuídos onde esses interagem através de conectores de software baseados em RPC. Esses conectores são implementados utilizando um middleware de comunicação baseado no padrão CORBA 2.x. A utilização dessa tecnologia viabiliza a interoperabilidade entre diferentes linguagens e plataformas, uma característica importante que nos permite ter implementações do SCS em Java, C#, C++ e Lua.

Segundo a classificação feita por *Mehta et Al* [8], conectores baseados em RPC fornecem serviços de Coordenação, Comunicação e Facilitação. Coordenação porque modelam o fluxo de controle entre componentes através de várias formas de invocação. Comunicação porque dados também são transferidos entre componentes através do uso de parâmetros. Por último, Facilitação porque esses conectores mediam e simplificam a interação entre componentes.

A arquitetura do SCS inspira-se em outros modelos como COM e CCM mas, em diversas ocasiões, tenta evitar a complexidade imposta por esses. As abstrações escolhidas são basicamente semelhantes às de outros modelos: uma unidade de composição oferece serviços através de facetas e especifica suas dependências através de receptáculos. A comunicação entre as unidades de

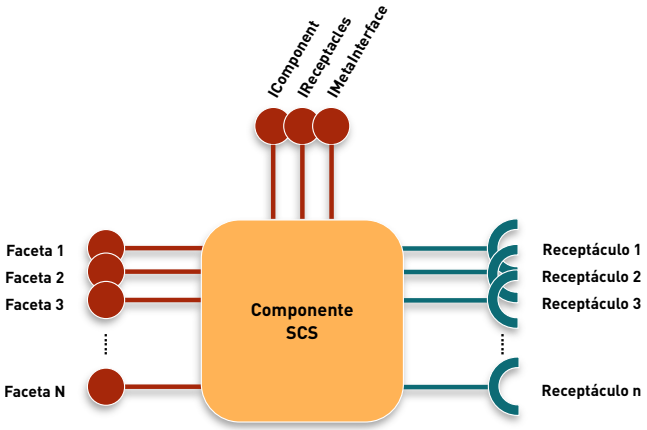

Figura 3.1: Exemplo de Componentes SCS

composição é realizada através desses dois conectores. A figura 3.1 apresenta um exemplo desse modelo.

O modelo define formas de interação, configuração e introspecção através de três facetas básicas: *IComponent*, *IReceptacles* e *IMetaInterface*. A primeira oferece operações de identificação e ativação de um componente. A segunda oferece operações para o gerenciamento de conexões entre receptáculos e facetas, e a última oferece um conjunto de operações para a introspecção.

3.1.1 Componentes SCS

Apesar do modelo SCS definir apenas facetas e receptáculos, a representação local de uma instância de um componente SCS é composta por facetas, receptáculos e um contexto. Um componente pode disponibilizar diversas facetas, e ter zero ou mais receptáculos. No contexto são armazenadas informações de identificação, assim como referencias para facetas, receptáculos e conexões existentes.

O componente é identificado por um atributo chamado *ComponentId*, composto por informações tais como: nome, versão e plataforma. Facetas e receptáculos são identificados por nomes únicos. A obtenção da referencia de uma faceta específica é realizada através da faceta *IComponent*. Esta faceta obrigatória é única e serve como representação do componente.

Na versão atual do middleware, um componente pode ser construído programaticamente através de uma API ou de forma automatizada, através de um arquivo de descrição em XML. O código 3.1 demonstra um arquivo de descrição.

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <scs:component xmlns:scs="tecgraf.scs.core">
```

```
3   <id>
4     <name>Hello</name>
5     <version>1.0.0</version>
6     <platformSpec>Java 1.6</platformSpec>
7   </id>
8   <facets>
9     <facet>
10      <name>Hello</name>
11      <interfaceName>IDL:scs/demos/helloworld/Hello:1.0</interfaceName>
12      <facetImpl>scs.demos.helloworld.servant.HelloServant</facetImpl>
13    </facet>
14  </facets>
15  <receptacles>
16    <receptacle>
17      <name>OtherHello</name>
18      <interfaceName>IDL:scs/demos/helloworld/Hello:1.0</interfaceName>
19      <isMultiplex>false</isMultiplex>
20    </receptacle>
21  </receptacles>
22 </scs:component>
```

Código 3.1: Arquivo de descrição de componente.

Facetas

Como mencionado anteriormente, facetas são serviços fornecidos por um componente. Esses serviços são representados por métodos e atributos definidos em um interface. Pelo fato do middleware ser baseado no padrão CORBA, as interfaces são descritas em IDL seguindo a sintaxe definida pela OMG.

Além das três facetas básicas, o desenvolvedor pode adicionar facetas ao seu componente definindo sua própria interface em IDL. O código 3.2 ilustra a definição da faceta *Hello* utilizada no descritor do componente apresentado no código 3.1.

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

Código 3.2: Interface *Hello*.

Uma possível implementação dessa interface em Java é apresentada no código 3.3. Em Java, uma faceta é mapeada diretamente para uma classe. Portanto, o paradigma de programação baseada em componentes é utilizado simultaneamente com o de programação orientada a objetos. Note que cada faceta recebe uma referência para o contexto do componente durante a sua

instanciação. É através desse objeto de contexto que o componente compartilha um estado entre as facetas. A manipulação das facetas e receptáculos dentro do componente também é feita através desse objeto.

```
1  package scs.demos.helloworld.servant;
2  import scs.core.ComponentContext;
3  import scs.demos.helloworld.HelloPOA;
4
5  public class HelloServant extends HelloPOA {
6    private ComponentContext myComponent;
7    public HelloServant(ComponentContext myComponent) {
8      super();
9      this.myComponent = myComponent;
10   }
11
12   public void sayHello() {
13     System.out.println("Hello World!");
14   }
15
16   @Override
17   public org.omg.CORBA.Object _get_component() {
18     return myComponent.getIComponent();
19   }
20 }
```

Código 3.3: Implementação da faceta Hello.

Receptáculos

Um dos objetivos da programação baseada em componentes é promover o reuso de código. No SCS, a reutilização dos serviços de componentes implementados é uma das formas nas quais se atinge esse reuso. Uma dependência é caracterizada justamente quando um componente necessita de um serviço provido por outro componente. Essas dependências são representadas no SCS através de receptáculos.

Um receptáculo é definido por um nome simbólico e pela interface da faceta da qual o componente depende. Em certas ocasiões a conexão de mais de uma faceta é interessante, por isso um atributo que sinaliza a multiplexação é definido. As linhas 16 a 20 do código 3.1 ilustram a descrição de um receptáculo.

Tipicamente facetas e receptáculos são conectados por ferramentas de configuração e implantação. A faceta *IReceptacles*, também obrigatória, foi definida com o intuito de facilitar as operações de conexão.

Contexto

O contexto é um objeto que representa um estado que é acessível somente no espaço de endereçamento do processo da unidade de composição. Fora

do processo, suas informações só podem ser acessadas através dos serviços providos pelas facetas.

Como dito anteriormente, o contexto é compartilhando entre as facetas. Logo, é um bom candidato para compartilhar informações entre elas. Atualmente os desenvolvedores adotam a prática de estender a classe do contexto quando desejam compartilhar novos dados.

O contexto é representado pela classe *ComponentContext* e suas operações podem ser vistas no código 3.4. Note que algumas operações de introspecção são mapeadas diretamente para a faceta *IMetaInterface*.

```
1  package scs.core;
2  /* ... */
3  public class ComponentContext {
4    public ComponentId getComponentId() {/* ... */}
5    public void addFacet(String name, String interfaceName, Servant servant)
6      throws SCSException {/* ... */}
7    public void updateFacet(String name, Servant servant)
8      throws SCSException {/* ... */}
9    public void addReceptacle(String name, String interfaceName,
10     boolean isMultiplex) throws ReceptacleAlreadyExistsException {/* ... */}
11   public void removeFacet(String name) throws SCSException {/* ... */}
12   public void removeReceptacle(String name)
13     throws ReceptacleDoesNotExistException {/* ... */}
14
15   public Collection<Facet> getFacets() {/* ... */}
16   public Facet getFacetByName(String name) {/* ... */}
17   public Collection<Receptacle> getReceptacles() {/* ... */}
18   public Receptacle getReceptacleByName(String name) {/* ... */}
19   public IComponent getIComponent() {/* ... */}
20
21   public void startup() throws StartupFailed {/* ... */}
22   public void shutdown() throws ShutdownFailed {/* ... */}
23 }
```

Código 3.4: Implementação do ComponentContext em Java

3.1.2 Infraestrutura de Execução

Em conjunto com o SCS, são distribuídos serviços auxiliares que fornecem um mecanismo padrão para a instanciação, configuração e execução de componentes. Esses serviços surgiram como resultado do mecanismo proposto por Augusto [29].

Esse mecanismo define dois serviços com funcionalidades distintas, um responsável por carregar componentes em seu espaço de memória, e o outro, responsável por gerenciar instâncias deste primeiro serviço. Basicamente, esse serviços são representados por dois componentes:

– **Container:** Disponibiliza um ambiente que permita a carga e descarga de componentes em tempo de execução. O *ExecutionNode* instancia esse componente sob demanda.

– **ExecutionNode:** Sua função principal é gerenciar instâncias do componente anterior. Tipicamente apenas uma instancia destes componente é implantada por máquina.

A figura 3.2 ilustra a conexão entre os dois componentes após a instanciação do *Container*. Na versão atual, o *ExecutionNode* pode gerenciar contêineres implementados em qualquer linguagem suportada pelo SCS. No entanto, o *Container* só pode instanciar componentes implementados na mesma linguagem na qual ele foi implementado. Geralmente o *Container* possui uma fábrica que constrói componentes a partir de um arquivo de descrição.

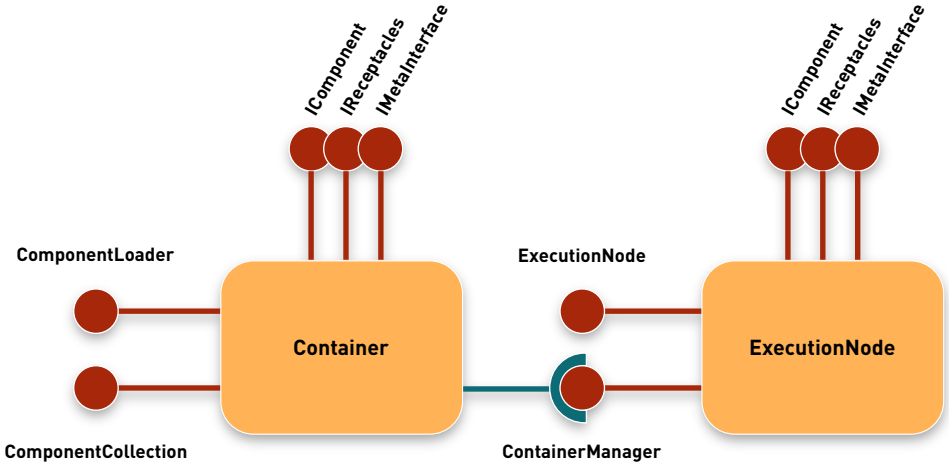

Figura 3.2: *ExecutionNode* e *Container*.

3.1.3 Infraestrutura de Implantação

A utilização da infraestrutura de execução em ambientes heterogêneos gerou demandas por um suporte mais robusto a implantação de componentes. Em seu trabalho, *Barbosa et al.* [17] estudou o problema e propôs uma infraestrutura que permite a implantação remota e descentralizada de componentes de software distribuídos multi-linguagem e multi-plataforma. As principais vantagens da infraestrutura desenvolvida por ele são:

1. Suporte à implantação de componentes com dependências estáticas;

2. Uma interface de programação que oferece ao ator da implantação:

   2.1. A gestão remota a partir de diferentes linguagens e plataformas;

2.2. A implementação de todas as fases de um processo de implantação;

2.3. A manipulação simplificada do ambiente de execução do sistema de componentes;

2.4. O mapeamento dos componentes na infraestrutura de execução e nas máquinas envolvidas através de níveis de detalhamento;

2.5. O planejamento e a implantação automática dos componentes em um parque de máquinas;

2.6. A implementação de novas estratégias automáticas para implantação.

Nessa infraestrutura, *Barbosa* evoluiu a implementação do *Container* e do *ExecutionNode* para suportar a implantação de novos componentes. Os componentes carregados no *Container* são obtidos pelo *ExecutionNode* através do serviço disponibilizado pelo componente *Repository*. Além disso, este último também permite a publicação de unidades de composição em tempo de execução. A figura 3.3 ilustra a composição do sistema em um cenário simplificado.

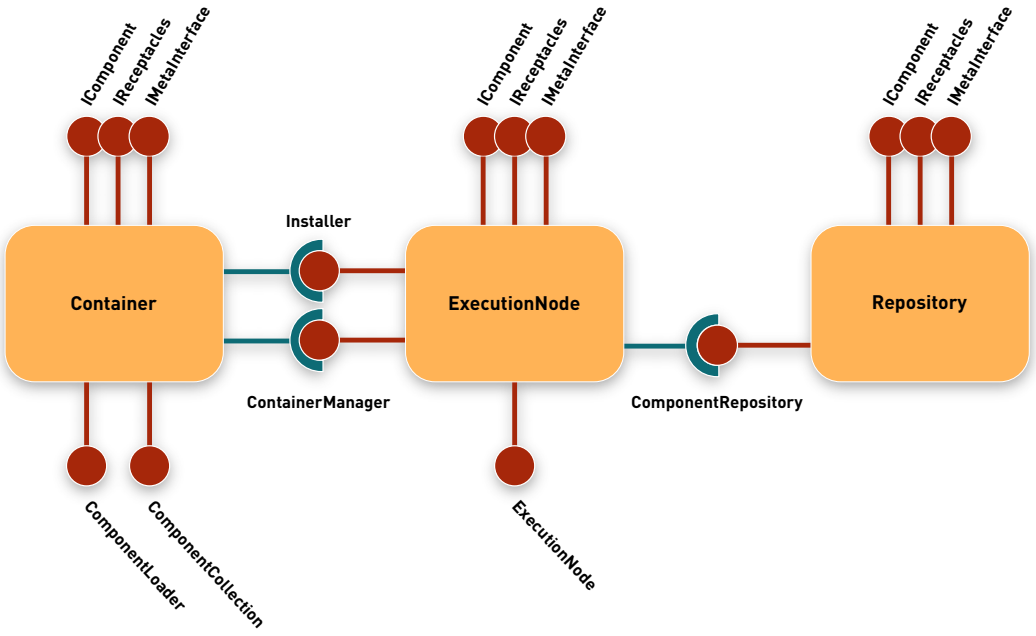

Figura 3.3: *ExecutionNode*, *Container* e *Repository*.

3.2 Conectores de Fluxo de Dados

Diferentemente dos conectores RPC, Conectores de Fluxo de Dados são utilizados para transferir grandes quantidades de dados entre processos

independentes. Desta forma, esse conector tipicamente provê apenas o serviço de Comunicação.

As características desse serviço de Comunicação podem ser diferentes entre aplicações de domínios distintos. Os requisitos de um fluxo de dados de uma aplicação que transmite uma gravação ao vivo não fazem sentido para o fluxo de uma aplicação que transmite o conteúdo de um arquivo em disco. Um fluxo de áudio pode ter uma política de descartar dados que não forem entregues em uma janela de tempo definida. A mesma política pode não ser aplicável durante a cópia de um arquivo.

Logo, a implementação desse tipo de conector em um middleware de componentes genérico precisa considerar as diferentes características de fluxos de dados em sua arquitetura. Portanto, as próxima seções irão revisar as características de Conectores de Fluxo de Dados e alguns conceitos relevantes.

3.2.1 Aplicações com Suporte a Fluxo de Dados

Em *Thies et al* [30], foi realizado um levantamento interessante sobre características encontradas em aplicações do domínio de fluxo de dados. Segundo o texto publicado, independente da implementação, as propriedades comuns a aplicações com suporte a fluxo de dados são:

- **Grandes quantidades de dados** - Um dos aspectos mais fundamentais, as aplicações operam com grandes (virtualmente infinitas) quantidades de dados.
- **Filtros de fluxo independentes** - Conceitualmente, uma computação sobre o fluxo é uma sequência de transformações nos dados do fluxo. Os autores se referem à unidade básica de transformação como filtro. Esses filtros são tipicamente independentes e auto-contidos, sem referências a variáveis globais ou outros filtros. Uma aplicação desse domínio é uma composição de filtros na forma de um grafo, onde as saídas de alguns filtros estão conectadas as entradas de outros.
- **Padrão estável de computação** - A estrutura do grafo de filtros é geralmente constante durante a execução de um programa.
- **Modificações ocasionais na estrutura do fluxo** - Apesar de cada arrumação do grafo de filtros ser executada por longos períodos, existem ocasiões nas quais o grafo é dinamicamente reestruturado. Segundo os autores, tipicamente as possíveis arrumações são conhecidas em tempo de compilação.

– **Comunicação ocasional fora do fluxo** - Além de computar grandes quantidades de dados, os filtros ocasionalmente comunicam pequenas quantidades de informações de controle.

– **Expectativas de alto desempenho** - Tipicamente, existem restrições de comunicação em tempo real, logo, eficiência é uma das principais preocupações.

3.2.2 Variações de Conectores de Fluxo de Dados

Segundo Metha [8], a utilização de Conectores de Fluxo de Dados variam em dez dimensões: *Delivery, Bounds, Buffering, Throughput, State, Identity, Locality, Synchronicity, Format* e *Cardinality.*

Idealmente, a implementação de Conectores de Fluxos de Dados para um modelo de componentes de software deveria oferecer suporte ao maior número de variações sem tornar o código demasiadamente complexo. Para isso é necessário que esses conceitos estejam claros durante o design da arquitetura de um conector desse tipo. Então, é preciso rever os significados dessas dimensões:

– ***Delivery*** - Variações sobre a política de entrega dos dados. Exemplos: Melhor esforço, pelo menos uma vez, até uma vez e exatamente uma vez.

– ***Bounds*** - Se os dados são transferidos com ou sem limites.

– ***Buffering*** - Se é necessária a utilização de *buffers* para conter os dados.

– ***Throughput*** - Unidades de medida utilizadas. Ex: Atômicas (Bytes / segundo) ou de mais alto nível (Operações HTTP / segundo).

– ***State*** - Indica se a comunicação possui ou não um estado.

– ***Identity*** - Se o fluxo possui ou não um nome.

– ***Locality*** - Se o fluxo é local ou remoto.

– ***Synchronicity*** - Se o fluxo é síncrono ou assíncrono.

– ***Format*** - Se os dados são estruturados ou não.

– ***Cardinality*** - Se é fluxo ponto a ponto, múltiplos servidores ou múltiplos clientes.

3.2.3 Conceitos Relevantes

O estudo das soluções utilizadas no trabalhos relacionados auxiliou no levantamento de conceitos relevantes para a implementação do suporte a fluxo de dados. Esses conceitos influenciaram diretamente nas decisões da abordagem seguida. Durante o design da arquitetura de um Conector de Fluxo de Dados, é preciso decidir se os fluxos serão tipados, decidir como será realizada a conexão entre os componentes e, por último, que modelos de execução serão fornecidos para o código cliente do middleware. Para realizar essas escolhas é necessário estar a par da relação custo/benefício de cada uma.

Sistema de Tipos

De forma geral, um sistema de tipos geralmente procura garantir que operações que esperam um determinado valor não sejam realizadas em valores com os quais estas não fazem sentido. A aplicação de um sistema de tipos em um Conector de Fluxo de Dados visa identificar o tipo do fluxo fornecido por um porta. Essa informação pode servir de diversas formas, por exemplo, na proibição de conexões de fluxos de tipos diferentes ou apenas como informação para que as aplicações decidam como interpretar os dados.

Conectividade dos Componentes

O estabelecimento de conexões entre dois componentes distintos pode ser feito de diversas formas. A conexão pode estar descrita na linguagem de descrição arquitetural ou pode ser feita pela aplicação através de uma API. Os conectores podem existir desde o início da vida de um componente ou podem ser criados durante a execução. O meio de transporte (ex.: TCP, UDP) pode ser negociado durante a conexão ou pode ser pré-acordado. Esses aspectos precisam ser considerados de maneira adequada, porque a liberdade em excesso pode tornar tanto a implementação quanto a sua utilização extremamente complexas.

Modelo de Execução

O Modelo de Execução descreve como e quando o código do componente que processa o fluxo de dados será executado. A implementação de diferentes modelos de execução para este tipo de conector é uma característica desejada. Dependendo do domínio da aplicação, pode se desejar escrever o código de processamento de dados sobre um determinado modelo de execução que, se escrito de outra forma, poderia limitar a expressividade do desenvolvedor,

tornando o código mais complexo ou até mesmo impossibilitando certas arquiteturas. Os modelos comumente utilizados em aplicações distribuídas para que possam ser escritas de forma clara e natural são:

- ***Blocking*** - Nesse modelo o código do componente executa uma chamada para consumir ou prover uma certa quantidade de dados. O código fica bloqueado nesse ponto até que a operação seja completada ou até que um tempo pré-determinado tenha passado.
- ***Event-Driven*** - Nesse modelo o código do componente registra funções próprias que serão chamadas pelo código do middleware quando houver dados disponíveis para consumo ou quando houver a necessidade de prover dados.

Neste trabalho optamos por utilizar o termo Modelo de Execução para estabelecer um vocabulário comum para esse conceito, pois o mesmo é utilizado na especificação *Streams for CCM* [12] da OMG. No entanto, esse conceito também pode ser entendido como um Modelo de Interação pelo desenvolvedor do componente.

3.3 Considerações Finais

O objetivo desse capítulo foi revisar os principais conceitos do *middleware* SCS e as principais características de Conectores de Fluxo de Dados e, com isso, prover uma base para o entendimento da extensão proposta no próximo capítulo. A seção 3.1 ilustrou o modelo de componentes SCS, detalhou partes da sua implementação e descreveu as infraestruturas de execução e implantação. Em seguida a seção 3.2 revisou as características de conectores de fluxo de dado e de aplicações que as utilizam.