3
Conceitos Básicos

Este capítulo apresenta as ferramentas e conceitos importantes para o entendimento deste trabalho. Será dada uma breve descrição do middleware de componentes SCS que foi utilizado como base para o desenvolvimento e estudo apresentados nesse trabalho e uma descrição mais completa sobre o conceito de *Interfaces Coletivas* e suas políticas de distribuição e agrupamento de dados.

3.1
Middleware de Componentes SCS

Uma ferramenta de grande importância neste trabalho foi o *middleware* de componentes *Sistema de Componentes de Software* - SCS [30]. Este foi concebido como uma ferramenta de experimentação e desenvolvimento de técnicas em sistemas de *componentes de software*, características tais que o tornaram apropriado para o estudo aqui descrito.

3.1.1
Modelo de Componentes

O Modelo de Componentes SCS teve inspiração nos modelos COM e *CORBA Component Model* - CCM [19], duas tecnologias bem estabelecidas no cenário de desenvolvimento atual. 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.

Uma aplicação *componentizada* consiste da interligação de peças de software que interagem entre si através de conectores para a realização de uma determinada tarefa. O modelo de interação do SCS define dois tipos de conectores: *Facetas* (provedores de serviços) e *Receptáculos* (portas de requisição de serviços).

Facetas

Todo serviço disponibilizado por um componente SCS é definido através de uma *Faceta*. Uma *Faceta* é constituída de uma interface em CORBA IDL (*Interface Definition Language*), um nome simbólico atribuído pelo desenvolvedor e o objeto CORBA que implementa essa interface, como pode-se ver no Código 3.2 (linhas 1-5). Como utiliza o padrão CORBA para a comunicação, os serviços de um componente SCS podem ser acessados de quaisquer clientes CORBA, o que torna o SCS compatível com diversas outras ferramentas existentes. O Código 3.1 mostra um exemplo de interface CORBA IDL.

Código 3.1: Exemplo de interface CORBA IDL

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

Um componente pode implementar um número arbitrário de *Facetas*, porém, três *Facetas* específicas são oferecidas pelo modelo (ver Figura 3.1):

- **IComponent**: define as operações básicas de um componente SCS para ativação e desativação do componente, bem como operações para obtenção de suas demais facetas;
- **IReceptacles**: define operações para gerenciar conexões de receptáculos, como por exemplo, métodos para conectar e desconectar um objeto a uma receptáculo e métodos para listar todos os objetos a ele conectados;
- **IMetaInterface**: define operações básicas para introspecção de facetas e receptáculos do componente.

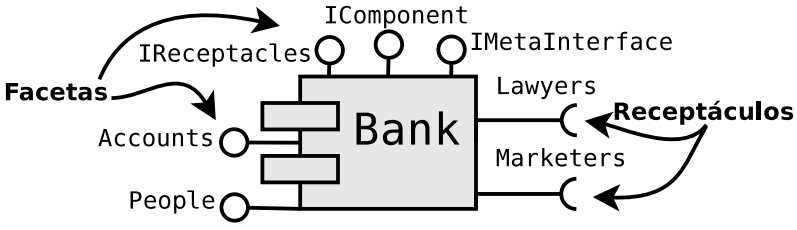

Figura 3.1: Modelo de componentes SCS (fonte [2])

Vale ressaltar que esses conectores possuem semântica síncrona, ou seja, não disponibilizam nenhum meio para a realização de operações não bloqueantes.

Receptáculos

Um componente SCS define os serviços dos quais ele depende para o seu funcionamento através dos *Receptáculos*. Conexões entre componentes são estabelecidas conectando-se a *Faceta* de um componente provedor ao *Receptáculo* de um componente requisitante do serviço.

Um *Receptáculo* é definido por um nome simbólico atribuído pelo desenvolvedor, o nome da interface que pode se conectar ao mesmo, os objetos CORBA que a implementam e uma flag para indicar se o receptáculo é múltiplo (pode ser conectada mais de uma *Faceta* ao mesmo) ou simples (permite somente uma conexão), como pode-se verificar no Código 3.2 (linhas 14-19).

Código 3.2: Estruturas que definem uma *Faceta* e um *Receptáculo*

```
1  struct FacetDescription {
2    string name;
3    string interface_name;
4    Object facet_ref;
5  };
6  typedef sequence<FacetDescription > FacetDescriptions ;
7
8  struct ConnectionDescription {
9    ConnectionId id;
10   Object objref;
11 };
12 typedef sequence<ConnectionDescription > ConnectionDescriptions ;
13
14 struct ReceptacleDescription {
15   string name;
16   string interface_name;
17   boolean is_multiplex;
18   ConnectionDescriptions connections ;
19 } ;
20 typedef sequence<ReceptacleDescripition> ReceptacleDescriptions ;
```


3.1.2 Infraestrutura de Execução

O SCS dispõe de um mecanismo padrão para instanciação, configuração e execução dos componentes. Esse mecanismo foi proposto por Augusto et al. [31], sendo composto basicamente de dois serviços principais:

- **Contêiner**: Disponibiliza um ambiente onde implementações de componentes são implantadas, criadas e executadas, sendo representado por um componente chamado `Container`.
- **Nó de Execução**: Corresponde a um dispositivo físico (*host*) onde contêineres executam. Sua função básica está em gerenciar os contêineres instanciados.

Esses serviços se conectam como mostrado na Figura 3.2.

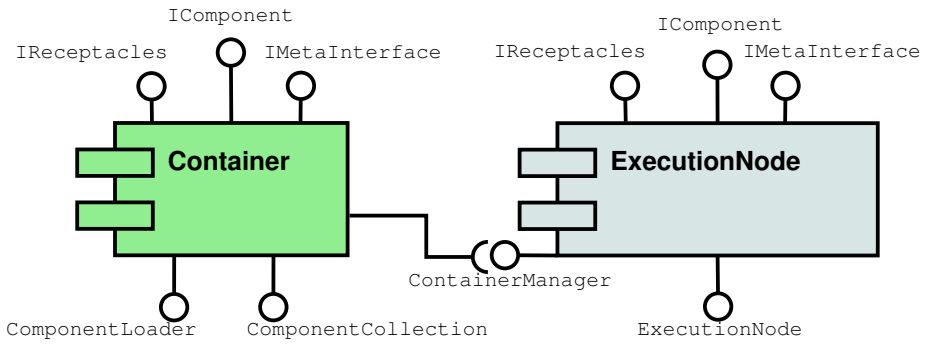

Figura 3.2: Infraestrutura de Execução do SCS (fonte [2])

Como complementação desses serviços, um ambiente de implantação foi proposto por Barbosa Jr [2], proporcionando melhorias como implantação remota e descentralizada de componentes de software distribuídos, multi-linguagem e multi-plataforma, todavia somente a primeira infraestrutura foi utilizada neste trabalho.

3.1.3 Suporte a Programação Paralela

O SCS, em sua concepção, não teve HPC como objetivo primário, por tanto não dispõe de recursos que suportem os requisitos e restrições de computação paralela, como mecanismos de comunicação coletiva e meios eficientes de redistribuição de dados.

Poder-se-ia ver o receptáculo múltiplo, como uma possível forma de expressar, por exemplo, uma arquitetura paralela *Master-Worker* onde os componentes *workers* estariam conectados no receptáculo múltiplo do componente *master*. Todavia, a redistribuição dos dados teria que ser feita manualmente e, para que a invocação aos *workers* fossem verdadeiramente paralela, *threads* também precisariam ser criadas manualmente, pois o SCS não possui suporte a invocações síncronas e o programador precisaria lidar com a conexão de cada *worker* individualmente.

3.2 Interfaces Coletivas

Através de suas interfaces, um componente define seu comportamento e funcionalidades. Usualmente um modelo de componentes define algum padrão de interação desses componentes através dessas interfaces. O padrão mais utilizado é o *Provides/Uses* [21], que consiste da disponibilização de uma coleção de recursos para outros componentes interessados. Normalmente esses

recursos são uma coleção de sub-rotinas que poderão ser utilizadas pelo componente que os importar, seguindo um modelo síncrono de chamadas.

Esses padrões podem até expressar comunicação coletiva, como o SCMD, mas não definem estratégias mais elaboradas de redistribuição de dados durante a comunicação. Com o objetivo de definir em nível de interface o comportamento coletivo e as estratégias de redistribuição de dados, foi definido por Baude et al. [12] o conceito de *Interfaces Coletivas* - ICs. Essa abordagem evita entidades intermediárias e proporciona uma maior eficiência e escalabilidade à comunicação, possibilitando a utilização de variados padrões de programação paralela.

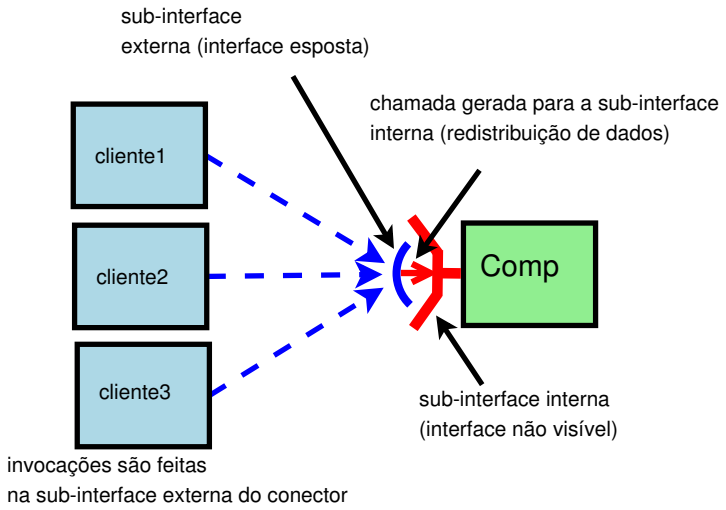

Figura 3.3: Arquitetura de *sub-interfaces* das ICs.

Cada *Interface Coletiva* é definida por duas *sub-interfaces*: uma interna e outra externa. Essa organização torna-se necessária para que o motor de redistribuição de parâmetros e invocação de métodos possa fazer a ligação entre essas *sub-interfaces*, realizando nesse ponto as redistribuições de dados configuradas. A Figura 3.3 ilustra o uso de uma interface coletiva, conhecida como *Gathercast*, onde clientes invocam um método com uma determinada assinatura na sub-interface externa, que é a interface oferecida pela faceta do componente. Essa faceta, já no lado do processo que hospeda o componente servidor, repassa a chamada ao motor de redistribuição, para que o mesmo possa realizar a invocação do método correspondente na sub-interface interna, oferecida pela implementação do componente. Nesse exemplo, o motor de redistribuição é responsável por agrupar os parâmetros recebidos pelas chamadas dos diversos clientes e por redistribuir os resultados para os mesmos, de acordo com as estratégias de redistribuição especificadas na definição da interface coletiva.

As estratégias de redistribuição básicas utilizadas pelas ICs são a *gather*, *scatter*, *reduce* e *broadcast*.

3.2.1 Interface Multicast

Uma interface *Multicast* possibilita que uma única invocação seja transformada em uma lista de invocações. Estratégias de distribuição de dados como o *broadcast* ou *scattering* precisam ser definidas para que o processo se complete. Essa interface pode ser síncrona ou assíncrona, sendo que as invocações são realizadas em paralelo, ou seja, o componente invoca um determinado método na interface interna do conector *Multicast* e o motor de redistribuição realiza a invocação paralela do método correspondente em todas as interfaces (externas) conectadas.

A Figura 3.4 exemplifica os três esquemas básicos de redistribuição de dados que são possibilitados pela interface *Multicast*. No primeiro, os parâmetros da invocação paralela são distribuídos através da estratégia de *broadcast*, ou seja, a sequência de *longs* é copiada para cada invocação que faz parte da invocação paralela, sendo que os resultados desta invocação são agrupados através da estratégia *gather*. No segundo, os parâmetros são distribuídos através da estratégia *scatter*, onde a sequência de *longs* é divida igualmente entre cada invocação que faz parte da invocação paralela. Os resultados são agrupados com *gather* onde *double* é mapeado para uma sequência de *doubles*. No terceiro esquema é feito o *scatter* dos parâmetros, e os resultados (retorno) sofrem uma operação de *reduce* (soma, máximo, mínimo). É perfeitamente possível também a combinação de *broadcast* dos parâmetros da invocação com *reduce* dos resultados.

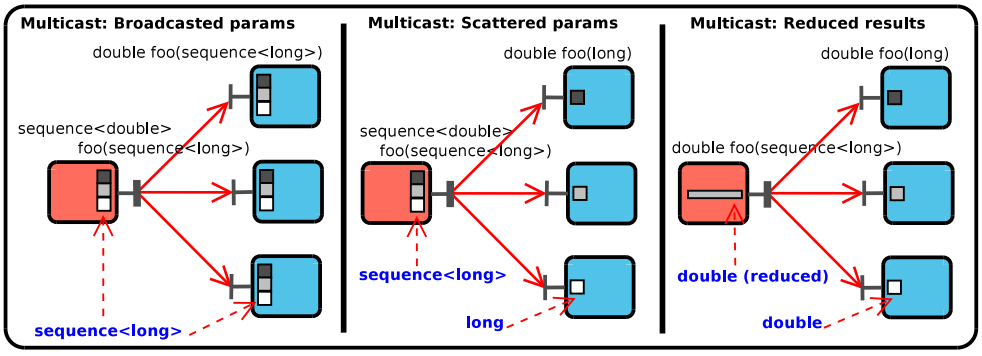

Figura 3.4: Estratégias de redistribuição com a interface *Multicast*

3.2.2 Interface Gathercast

Uma interface *Gathercast* realiza o trabalho oposto da interface *Multicast*: Ela agrupa um conjunto de chamadas em uma única chamada, realizando as devidas operações de *gathering* ou *reduce* dos parâmetros na invocação. Essa interface possui a semântica de uma barreira, pois somente invoca o método correspondente na interface interna do conector *Gathercast* depois que todos os clientes conectados invocaram o método correspondente na interface externa (Figura 3.3).

A Figura 3.5 exemplifica os três esquemas básicos de redistribuição de dados que são possibilitados pela interface *Gathercast*. No primeiro, os resultados da invocação paralela são distribuídos através da estratégia de *broadcast*, ou seja, a sequência de *doubles* é copiada para cada cliente conectado que realizou a invocação, sendo que os parâmetros desta invocação são agrupados através da estratégia de *gather* resultando no mapeamento *long* para uma sequência de *longs*. No segundo, os resultados são distribuídos através da estratégia *scatter*, onde a sequência de *doubles* é divida igualmente entre cada cliente conectado que participou da invocação, os parâmetros são agrupados com *gather*. No terceiro esquema os parâmetros sofrem uma operação de *reduce*, e os resultados são redistribuídos pela estratégia de *scatter*.

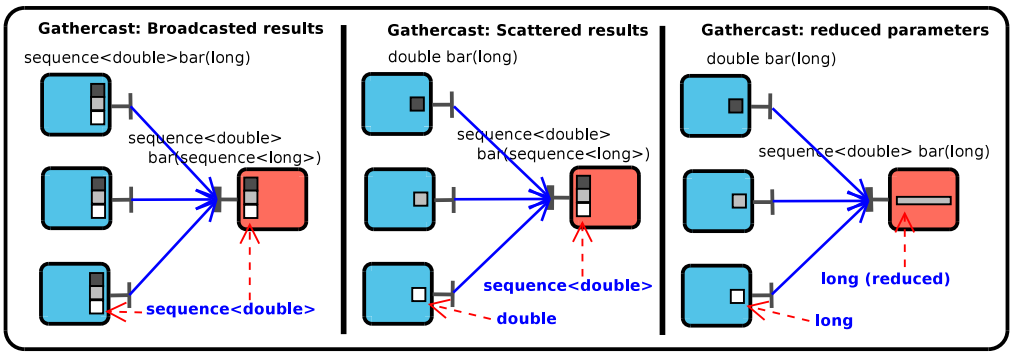

Figura 3.5: Exemplos de estratégias de redistribuição com *Gathercast*

O mecanismo de *Interfaces Coletivas*, além de permitir que o comportamento coletivo do componentes seja definido na interface e não precisar fazer uso de entidades intermediárias para a comunicação coletiva (evitando problemas como degradação de escalabilidade, causados por redirecionamentos desnecessários de chamadas), possui um outro aspecto relevante: Comunicação tipada. Outros ambientes como MPI e derivados possuem uma API fixa de comunicação em grupo, onde cada operação coletiva é realizada por uma determinada função. Os tipo dos parâmetros são genéricos no sentido de que

qualquer tipo de dado é tratado como um grande bloco de memória, e a estratégias de redistribuição são definidas em estruturas de dados não triviais[1], deixando o processo muito susceptível a erros.

Com o mecanismo de ICs, a comunicação em grupo é definida em termos de duas interfaces básicas (*Gathercast* e *Multicast*), e das estratégias de redistribuição que são anotadas para cada método definido nessas interfaces. Essas interfaces e todos os seus itens (nome dos métodos, tipos de parâmetros e de valores de retorno) são definidos pelo usuário e expressam uma semântica própria da aplicação.

3.2.3 Padrões de programação paralela suportados

Aqui estão listados alguns padrões de programação paralela que podem ser utilizados com as *Interfaces Coletivas*. Esses padrões são parte de uma coletânea de padrões de programação paralela definida por Mattson et al. [32]:

- **SPMD**: Padrão onde os dados são divididos e processados de forma simétrica por um mesmo processo instanciado várias vezes. O demo `PI_Dartboard` que está descrito no Capítulo 5 é um exemplo desse padrão.
- **MPMD**: Pode ser alcançado com interligação de vários componentes paralelos distintos de uma mesma aplicação. O demo `FindMAXValue` pode ser considerado um exemplo desse padrão por possuir dois componentes distintos interligados.
- **Master-Worker**: É inerente à semântica da interface *Multicast*, dado que, esta, quando requisitada, realiza um grupo de invocações paralelas para componentes que exerceriam o papel de *workers*. O demo PI_Dartboard no Capítulo 5 ilustra bem esse padrão de paralelização.
- **Divide and Conquer**: Como a paralelização com as ICs é baseada em funções, esse padrão de paralelização é suportado por definição [12].
- **Recursão**: Conceito inerente a programação funcional ou baseada em funções, assim como as ICs.

[1] É possível a construção de tipos customizados em MPI, todavia o programador precisa definir os detalhes de todos campos (tamanho, tipo, quantidade) da mesma forma.

3.2.4 Implementação de Referência

A implementação de referência do *Grid Component Model*, conhecida como *Component Framework Implementation* - CFI [33], foi desenvolvida tendo como base o *middleware* para computação em grade ProActive.

Descrito por Baduel et al. [14], o ProActive, é uma biblioteca Java de código aberto para computação em grade, que suporta programação concorrente e paralela, proporcionando comunicação distribuída e assíncrona. Com um conjunto reduzido de primitivas, ProActive fornece uma API que possibilita o desenvolvimento de aplicações paralelas e sua implantação em sistemas distribuídos e grades computacionais.

O modelo de objetos ativos (do Inglês, *Active Objects Model*) é base do ProActive, e onde as *Interfaces Coletivas* foram aplicadas. Um objeto ativo é sempre um proxy para um objeto (possivelmente) remoto com sua própria thread de controle e possui as seguintes semânticas de invocação:

- **Síncrona**: Execução bloqueia até que um resultado (valor de retorno) ou exceção sejam retornados.
- **Assíncrona sem retorno**: Invocação não bloqueante para métodos que não possuem valor de retorno.
- **Assíncrona com resultado futuro**: Um objeto futuro é devolvido, e o chamador continua seu fluxo de execução. O objeto ativo irá processar a invocação de acordo com sua política de recebimento, e o objeto futuro será atualizado com o valor do resultado da execução do método.

Apesar de ser uma implementação de referência, alguns pontos do modelo GCM ainda não são suportados. A implementação de interfaces *Gathercast* é restrita ao gerenciamento de sincronização básica, onde um limite de tempo que pode ser especificado se o método retorna um resultado. Políticas de redistribuição de dados para resultados não são configuráveis, ocorrendo somente da forma padrão.

Interfaces *Multicast* podem utilizar as redistribuições de dados *broadcast* (todos os elementos da lista são enviados para todas as interfaces conetadas), *one-to-one* (envia o *ith* elemento da lista para a *ith* interface conetada - é a forma padrão), *round-robin* (distribui os elementos por rodadas, onde, em cada rodada, todos recebem um elemento até que todos tenham sido distribuídos), *unicast* (envia um determinado elemento da lista para somente uma das interfaces conetadas) e *random* (distribui cada elemento de forma randômica).

O Código 3.3 mostra um exemplo da definição de uma interface coletiva *Multicast*, onde é usado o mecanismo de anotações da API reflexiva da linguagem Java para a configuração das políticas de redistribuição de dados (linhas 19 e 20).

Código 3.3: Exemplo de definição da interface *Multicast* no ProActive.

```
1  package org.objectweb.proactive.examples.pi;
2
3  import java.util.List;
4
5  import org.objectweb.proactive.core.component.type.
6    annotations.multicast.ClassDispatchMetadata;
7  import org.objectweb.proactive.core.component.type.
8    annotations.multicast.ParamDispatchMetadata;
9  import org.objectweb.proactive.core.component.type.
10   annotations.multicast.ParamDispatchMode;
11
12 /**
13  * This interface represents the client multicast
14  * interface of the master component in the component
15  * version of the application.
16  * @author The ProActive Team
17  *
18  */
19 @ClassDispatchMetadata(mode =
20   @ParamDispatchMetadata(mode = ParamDispatchMode.ONE_TO_ONE)
21 )
22 public interface PiCompMultiCast {
23
24     /**
25      * Initiates the computation of pi on all the
26      * workers bound to the client multicast interface
27      * @param msg the list of intervals that have to be
28      * distributed to the workers
29      * @return The list of partial results of pi computation
30      * that have to be gathered into the final pi value
31      */
32     public List<Result> compute(List<Interval> msg);
33
34     /**
35      * Sets scale for several pi computers
36      * @param scale The scale to set
37      */
38     public void setScale(List<Integer> scale);
39 }
```