

5

Exemplos de Uso dos Conectores Coletivos

Esse capítulo descreve a paralelização de dois problemas com o *middleware* para Computação Paralela SCS-COLLECTIVE. Para facilitar o entendimento do processo de utilização dos conectores, foram escolhidos exemplos simples que somente fazem uso de um conector em separado.

5.1

Conector *MulticastReceptacle*

Nesta seção são apresentados os passos necessários para a construção de uma aplicação que faz uso do conector *MulticastReceptacle*. Para facilitar o entendimento, primeiro é descrito o problema a ser paralelizado e seu algoritmo serial, sendo, após isso, apresentada uma proposta de paralelização procedural (pseudo-código) no estilo do MPI. Neste ponto, a proposta de paralelização procedural é adaptada para uma proposta de paralelização com componentes, onde são identificados os componentes e a arquitetura de comunicação entre eles com o conector *MulticastReceptacle*.

Uma vez que a arquitetura de paralelização com componentes está definida, são definidas as interfaces dos conectores paralelos necessários e suas respectivas implementações.

5.1.1

Demo *PI_Dartboard*

Para exemplificar o uso do conector *MulticastReceptacle*, foi escolhido um algoritmo simples cuja paralelização se adéqua muito bem a esse conector: Cálculo do número π . A forma de realizar o cálculo do número π utilizada é a que contabiliza os pontos lançados aleatoriamente em uma circunferência da seguinte forma:

1. Inscreve-se um círculo em um quadrado
2. Geram-se pontos randomicamente no quadrado
3. Determina-se o número de pontos no quadrado que também estão no círculo

4. Realiza-se a divisão do número de pontos dentro do círculo pelo número de pontos dentro do quadrado¹
5. π , então, é dado por: $\pi = 4.0 * \text{pontos_circulo} / \text{pontos_quadrado}$

A implementação utilizada (Código 5.1) seguiu o algoritmo descrito em [35].

Código 5.1: Pseudo algoritmo para aproximação de π

```

1 npoints = 10000
2 circle_count = 0
3
4 for i := 1 step until npoints do
5
6   generate 2 random numbers between 0 and 1
7   xcoordinate = random1
8   ycoordinate = random2
9
10  if (xcoordinate, ycoordinate) inside circle
11  then circle_count = circle_count + 1
12
13 end (i-loop)
14
15 PI = 4.0*circle_count/npoints

```

Proposta de paralelização Master-Worker

Uma estratégia de paralelização trivial pode ser definida com a partição dos pontos entre os *workers*. Os *workers* calcularão o número π com uma parte dos pontos e no *master* será feita uma média do resultado de cada *worker* para a obtenção da aproximação final do número do π .

As operações coletivas utilizadas serão o *broadcast* do número de *darts* (pontos aleatórios) que cada *worker* deverá calcular e a *redução* dos resultados gerados por cada *worker*, como se pode verificar no Código 5.2.

Arquitetura de componentes SCS-Collective proposta

A Figura 5.1 mostra como foi estruturada a arquitetura *Master-Worker* com componentes, e como as facetas são conectadas ao conector *MulticastReceptacle* para a realização da invocação paralela.

O Código 5.3 exemplifica a definição do Conector em IDL *MulticastReceptacle* para a arquitetura de componentes paralela proposta, onde a anotação **@CollectiveFacetConfig** define qual interface representa a *sub-interface* interna e qual representa a *sub-interface* externa do conector (interface requerida). Na *sub-interface* interna, a anotação **@MethodDispatchConfig** define a estratégia de envio dos parâmetros

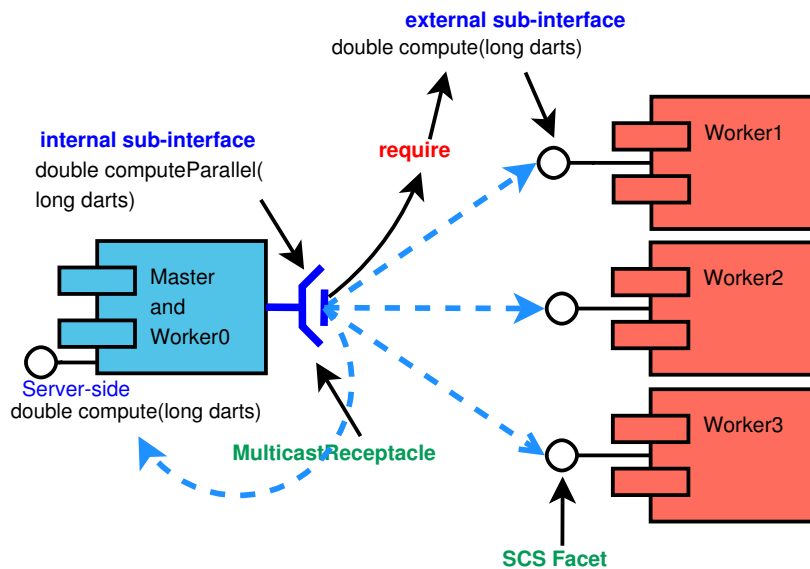
¹Quanto mais pontos são gerados, melhor a aproximação para o número π .

Código 5.2: Pseudo algoritmo paralelo para aproximação de PI

```

1 npoints = 10000
2 circle_count = 0
3 p = number of tasks
4 {number of points to generate are divided by number of tasks}
5 darts = npoints/p
6
7 find out if I am MASTER or WORKER
8 do j = 1,darts
9   generate 2 random numbers between 0 and 1
10  xcoordinate = random1
11  ycoordinate = random2
12  if (xcoordinate, ycoordinate) inside circle
13  then circle_count = circle_count + 1
14 } end do
15
16 if I am MASTER
17   receive from WORKERS their circle_counts
18   compute PI (use MASTER and WORKER calculations)
19 else if I am WORKER
20   send to MASTER circle_count
21 end if

```

Figura 5.1: Arquitetura de paralelização *Master-Worker*

(`@ParamDispatch (Strategy.BROADCAST)`) e a estratégia de recebimento do retorno (`@ResultGathering (Strategy.REDUCE)`). A implementação desta interface é obtida através de geração de código. Na *sub-interface* externa não há necessidade de anotações nos métodos, dado que é somente a interface requerida implementada por outros componentes.

De acordo com a definição das ICs, é possível configurar individualmente cada método de um dado conector paralelo, mas no estado atual da implementação a primeira configuração é aplicada a todos os métodos definidos na interface do conector.

Código 5.3: Definindo o conector *MulticastReceptacle*

```

1 module scs {
2   module demos {
3     module Parallel_PI_Dartboard {
4       //Defining MulticastReceptacle
5       //=====
6       /**
7        * @CollectiveFacetConfig (
8        *   @MulticastMode ( MulticastMode.INTERNAL ) )
9        */
10      interface PI_DartboardMulticast
11      {
12        /**
13         * @MethodDispatchConfig (
14         *   @ParamDispatch ( Strategy.BROADCAST ),
15         *   @ResultGathering ( Strategy.REDUCE ) )
16         */
17        double computeParallel( in long darts );
18      };
19      /**
20       * @CollectiveFacetConfig (
21       *   @MulticastMode ( MulticastMode.EXTERNAL ) )
22       */
23      interface PI_DartboardWorkers
24      {
25        double compute ( in long darts ) ;
26      };
27    };
28  };
29 };

```

Construindo o código da sub-interface interna

A implementação da *sub-interface* interna do *MulticastReceptacle* seria gerada automaticamente pelo comando seguinte:

```
./scscodegen --all -v Parallel_PI_Dartboard.idl
```

Todavia, no estado atual da implementação, a construção ainda feita pela montagem manual dos modelos de códigos para cada política de distribuição de dados, sendo necessários os devidos ajustes para a adaptação aos tipos da aplicação definidos nas *sub-interfaces*.

Após esse processo, tem-se a implementação (arquivos .h e .cpp) do objeto que implementa a *sub-interface* interna do *MulticastReceptacle* definido. Esse objeto é capaz de realizar a redistribuição dos dados e gerenciar as referências para os objetos remotos das facetadas conectadas, realizando a invocação paralela dos mesmos quando requisitado.

Implementando os Workers (sub-interface externa)

A implementação da *sub-interface* externa (ou interface requerida do *MulticastReceptacle*) é feita da forma padrão do SCS, onde se fazem necessários basicamente a respectiva classe POA gerada nos stubs, a implementação dos métodos definidos na interface IDL, além da definição os métodos estáticos

instantiate e **destruct**. A implementação da *sub-interface* externa é mostrado nos Códigos 5.4 e 5.5.

Código 5.4: Definição dos Workers

```

1 #ifndef PI_DARTBOARD_WORKERSIMPL.H
2 #define PI_DARTBOARD_WORKERSIMPL.H
3
4 #include <stubs/mico/Parallel_PI_Dartboard.h>
5 #include <ComponentContext.h>
6 using namespace scs::demos::Parallel_PI_Dartboard;
7 extern double dboard(int DARTS);
8
9 class PI_DartboardWorkersImpl : virtual public
10 POA_scs::demos::Parallel_PI_Dartboard::PI_DartboardWorkers {
11 private:
12     scs::core::ComponentContext* myComponent;
13 public:
14     static PortableServer::ServantBase* instantiate (
15         scs::core::ComponentContext* componentContext);
16     static void destruct( void* obj );
17
18     PI_DartboardWorkersImpl(scs::core::ComponentContext* context) ;
19     ~PI_DartboardWorkersImpl();
20
21     CORBA::Double compute(const CORBA::Long data) ;
22 };
23 #endif

```

Código 5.5: Implementação das funções dos Workers

```

1 #include "PI_DartboardWorkersImpl.h"
2 //C++ constructor
3 PI_DartboardWorkersImpl::PI_DartboardWorkersImpl(
4     scs::core::ComponentContext* myComponent)
5 { this->myComponent=myComponent; }
6 //scs constructor
7 PortableServer::ServantBase* PI_DartboardWorkersImpl::instantiate (
8     scs::core::ComponentContext* componentContext)
9 { return (PortableServer::ServantBase*)
10     new PI_DartboardWorkersImpl(componentContext); }
11 //scs destructor
12 void PI_DartboardWorkersImpl::destruct( void* obj )
13 { delete (PortableServer::ServantBase*) obj; }
14 //C++ destructor
15 PI_DartboardWorkersImpl::~PI_DartboardWorkersImpl( ) { }
16 //functional code
17 CORBA::Double PI_DartboardWorkersImpl::compute (const CORBA::Long darts)
18 {
19     //all functional code stay here
20     std::cout << "Component " << this->myComponent <<
21         " has started calculating PI with " << darts << " darts..." << std::endl;
22     CORBA::Double local_pi = dboard((int) darts);
23     std::cout << "Component " << this->myComponent <<
24         " has finished calculating PI!" << std::endl;
25     return local_pi;
26 }

```

Descrevendo e compilando o componente

Uma vez que todos os artefatos de software foram implementados e gerados, é necessário descrever o componente para que o mesmo possa ser carregado pelo container C++ do SCS. No Código 5.6 é feita a descrição

da *sub-interface* externa (PI_DartboardWorkers), a descrição do conector *MulticastReceptacle* e a descrição da implementação gerada para a *sub-interface* externa do receptáculo multicast (PI_DartboardMulticast), sendo estes os três itens básicos para a definição de um *MulticastReceptacle*.

Código 5.6: Descrição das facetas necessárias

```

1 #include <ComponentBuilder.h>
2 #include <ComponentContext.h>
3 #include "stubs/mico/deployment.h"
4 #include "PI_DartboardWorkersImpl.h"
5 #include "PI_DartboardMulticastClientGen.h"
6
7 extern "C" scs::core::ComponentContext * createComponent(
8     scs::core::ComponentBuilder& componentBuilder,
9     const scs::container::StringSeq& args)
10 {
11     /** criação de descrições de facetas */
12     std::list<scs::core::ExtendedFacetDescription> extFacets;
13     scs::core::ExtendedFacetDescription solverDesc_client;
14     solverDesc_client.name = "PI_DartboardWorkers";
15     solverDesc_client.interface_name =
16         "IDL:scs::demos::Parallel_PI_Dartboard::PI_DartboardWorkers:1.0";
17     solverDesc_client.instantiate = PI_DartboardWorkersImpl::instantiate;
18     solverDesc_client.destructor = PI_DartboardWorkersImpl::destruct;
19     extFacets.push_back(solverDesc_client);
20
21     /** criação de descrições de receptaculo */
22     std::list<scs::core::ReceptacleDescription> receptacleDescs;
23     scs::core::ReceptacleDescription multicastReceptDesc;
24     multicastReceptDesc.name = "PI_DartboardWorkers";
25     multicastReceptDesc.interface_name =
26         "IDL:scs/demos/Parallel_PI_Dartboard/PI_DartboardWorkers:1.0";
27     multicastReceptDesc.is_multiplex = true;
28     /** definição dos campos relativos ao MulticastReceptacle */
29     multicastReceptDesc.server_interface_name =
30         multicastReceptDesc.interface_name;
31     multicastReceptDesc.client_interface_name =
32         "IDL:scs::demos::Parallel_PI_Dartboard::PI_DartboardMulticast:1.0";
33     receptacleDescs.push_back(multicastReceptDesc);
34
35     /** adicionando o objeto faixada do receptáculo multicast */
36     scs::core::ExtendedFacetDescription controlDesc;
37     controlDesc.name = "PI_DartboardMulticast";
38     controlDesc.interface_name =
39         "IDL:scs::demos::Parallel_PI_Dartboard::PI_DartboardMulticast:1.0";
40     controlDesc.instantiate = PI_DartboardMulticastClientGen::instantiate;
41     controlDesc.destructor = PI_DartboardMulticastClientGen::destruct;
42     extFacets.push_back(controlDesc);
43
44     /** criação do ComponentId */
45     scs::core::ComponentId componentId;
46     componentId.name = "Parallel_PI_Dartboard";
47     componentId.major_version = '1';
48     componentId.minor_version = '0';
49     componentId.patch_version = '0';
50     componentId.platform_spec = "none";
51     return
52         componentBuilder.newComponent(extFacets, receptacleDescs, componentId);
53 }

```

Vale ressaltar que os componentes *Workers* e *Master* implementam as mesmas facetas (e possuem mesma descrição - Código 5.6), se diferenciando apenas no fato de que nos componentes *Workers* a faceta utilizada será *PI_DartboardWorkers* e no *Master* será utilizado o *MulticastReceptacle*

PI.DartboardMulticast (além da faceta PI.DartboardWorkers, pois o componente *Master* também faz o papel de *Worker*).

Implantação e Execução

Com o binário do componente devidamente gerado (*Parallel_PI_Dartboard.so*), é necessário criar um configurador, ou seja, o programa/script que vai criar várias instâncias do referido componente no contêiner e realizar as respectivas conexões entre os mesmos de acordo com a arquitetura definida na seção 5.1.1. No tutorial de uso do SCS-COLLECTIVE em [36] tem-se a implementação desse configurador que realiza a carga, configuração e execução da aplicação.

A compilação desse configurador pode ser feita utilizando o utilitário TecMake com o script makefile presente no no tutorial supracitado.

A execução da aplicação construída, se dá através da instanciação da infraestrutura de execução do SCS apresentada no Capítulo 3 nos *hosts* que serão utilizados. Após isso se faz necessária a implantação do componente nos locais adequados em cada *host* onde o contêiner possa localizá-lo (basicamente a cópia do arquivo **.so* para o diretório do executável do contêiner).

Uma vez que a infraestrutura está preparada e os artefatos da aplicação (arquivos **.so*, de configuração, de entrada/saída) foram devidamente implantados, pode-se iniciar a aplicação executando o configurador.

5.2

Conector GatherFacet

A *GatherFacet* possui uma semântica de barreira, ou seja, uma chamada a um método definido em uma *GatherFacet*, somente executa quando todos os clientes realizarem a invocação, o que, de forma global, realiza a sincronização da aplicação.

Dada essa semântica, não é possível construir uma aplicação paralela somente com *GatherFacet*, pois não há como realizar nenhuma distribuição de dados para processamento em paralelo sem o *MulticastReceptacle*. Então, para exemplificar seu uso, foi decido por um demo simples que apresenta um ponto de sincronização explícito, onde será utilizada a *GatherFacet*.

5.2.1

Demo FindMAXValue

Para exemplificar os requisitos para a construção de uma *GatherFacet*, foi construída a aplicação componentizada FindMAXValues que encontra o valor

máximo em uma lista de números distribuída entre componentes. As próximas seções descrevem os passos para a construção da mesma.

Arquitetura de Componentes

A arquitetura de componentes definida é constituída de componentes *workers* e um componente *Gather*. Componentes *workers* encontram os máximos locais de suas listas e realizam uma chamada ao componente da *GatherFacet* passando os máximos locais encontrados. A *GatherFacet* realiza o *gathering* dos parâmetros e encontra o máximo global, realizando o *broadcasting* do resultado para os *workers*. A Figura 5.2 mostra a arquitetura de componentes utilizada e suas respectivas conexões.

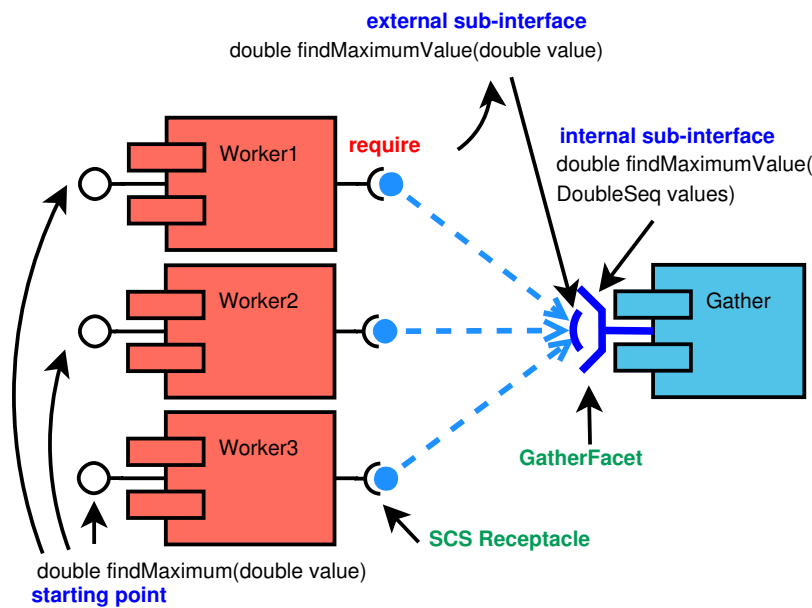


Figura 5.2: Arquitetura de componentes do demo FindMAXValues

O esquema de execução desse exemplo inicia na invocação do método `double findMaximum (double value)` nos componentes *workers* (no *Starting point*). Esse método encontra os máximos locais e faz a invocação, através do receptáculo, do método da *sub-interface* externa `double findMaximumValue (double value)` da *GatherFacet*. A *sub-interface* externa realiza as devidas operações de redistribuição de dados e bloqueia até que todos os componentes *Workers* realizem a invocação, para finalmente executar o método `findMaximumValue(DoubleSeq values)` na *sub-interface* interna da *GatherFacet* que finalmente encontrará o valor máximo global e retornará o resultado para os *workers* usando a política de distribuição *broadcasting*.

A definição das interfaces referentes a esse exemplo estão descritas no Código 5.7, onde estão definidas a *sub-interface* externa (interface exposta) da *GatherFacet* `FindMAXValueReceiver` e a *sub-interface* interna (local) `FindMAX` que definem o componente *Gather* e a faceta `FindMAXWorkers` que define os componentes *Workers*.

Código 5.7: Definição das interfaces da aplicação `FindMAXValues`

```

1  module scs {
2    module demos {
3      module FindMAXValue {
4
5        typedef sequence<double> DoubleSeq;
6        //Defining GatherFacet
7        //*****
8        /**
9         * @CollectiveFacetConfig (
10         *   @GatherMode ( GathercastMode.INTERNAL ) )
11         */
12        interface FindMAX
13        {
14          double findMaximumValue ( in DoubleSeq values ) ;
15        };
16        /**
17         * @CollectiveFacetConfig (
18         *   @GatherMode ( GathercastMode.EXTERNAL ) )
19         */
20        interface FindMAXReceiver
21        {
22          /**
23           * @MethodDispatchConfig (
24           *   @ParamGathering ( Strategy.GATHER ),
25           *   @ResultDispatch ( Strategy.BROADCAST ) )
26           */
27          double findMaximumValue ( in double value ) ;
28        };
29        //END GatherFacet
30        //*****
31        //Defining workers
32        interface FindMAXWorkers
33        {
34          double findMaximum( in DoubleSeq vector ) ;
35        };
36      };
37    };
38 };

```

A anotação `@CollectiveFacetConfig` define qual interface representa a *sub-interface* externa e qual representa a *sub-interface* interna do conector. Na *sub-interface* externa, a anotação `@MethodDispatchConfig` define a estratégia de recebimento dos parâmetros para `@ParamGathering (Strategy.GATHER)` e a estratégia de envio do valor de retorno para `@ResultGathering (Strategy.BROADCAST)`. A implementação desta *sub-interface* (externa) será produto de geração de código. Na *sub-interface* interna não há necessidade de anotações nos métodos, e a implementação será feita pelo programador do componente.

Implementando a sub-interface interna

Apesar de estar definido em IDL, a *sub-interface* interna da *GatherFacet* não precisa ter uma implementação seguindo estritamente as regras de implementação de facetas do SCS (estender a respectiva classe POA, implementar os métodos **instantiate** e **destruct**), precisando somente da definição do nome da classe como o nome da respectiva interface com o sufixo Impl, e a implementação dos métodos seguindo a assinatura definida em IDL (padrão utilizado pelo gerador de código para identificar o arquivo de implementação). Os Códigos 5.8 e 5.9 ilustram a implementação da *sub-interface* interna da *GatherFacet* definida para a aplicação FindMAXValue.

Código 5.8: Header da *sub-interface* interna da *GatherFacet*

```

1 #ifndef FindMAXSERVANTIMPL_H_
2 #define FindMAXSERVANTIMPL_H_
3
4 #ifdef SCS_MICO
5 #include <stubs/mico/FindMAXValue.h>
6 #else
7 #include <stubs/orbix/FindMAXValueS.hh>
8 #endif
9
10 #include <ComponentContext.h>
11
12 using namespace scs::demos::FindMAXValue;
13
14 class FindMAXServantImpl {
15 private:
16     scs::core::ComponentContext* myComponent ;
17
18 public :
19     FindMAXServantImpl( scs::core::ComponentContext* context ) ;
20     ~FindMAXServantImpl() ;
21     //user fuction
22     CORBA::Double findMaximumValue ( DoubleSeq &values ) ;
23 };
24 #endif

```

A implementação dos Workers foi feita seguindo as regras de implementação de facetas do SCS, como se pode verificar nos Códigos 5.10 e 5.11. Da linha 18 até a 23 do Código 5.11 é realizada a busca pelo valor máximo local. Nas linhas 24 a 28 é obtida a referência para a *GatherFacet* conectada e na linha 30 é realizada a invocação a mesma, onde é passado o valor máximo local para a *GatheFacet*. Uma vez que esta tenha recebido todos os máximos locais, será executada a função definida entre as linhas 8 e 22 do Código 5.9, onde é encontrado o máximo global.

Construindo a sub-interface externa

Assim como na geração de código para o conector *MulticastReceptacle*, somente seria preciso utilizar o utilitário *scscodegen*:

```
./scscodegen --all -v FindMAXValues.idl
```

Código 5.9: Implementação da *sub-interface* interna da *GatherFacet*

```

1 #include "FindMAXServant.h"
2 //C++ constructor
3 FindMAXServantImpl::FindMAXServantImpl(scs::core::ComponentContext*
   myComponent){
4     this->myComponent=myComponent;
5 }
6
7 //functional code
8 CORBA::Double FindMAXServantImpl::findMaximumValue ( DoubleSeq &values )
9 {
10     cout << "Executando SERVANT..." << values.length() << endl;
11
12     CORBA::Double result = values[0];
13
14     for (int i = 1; i < values.length(); i++ )
15         if(result < values[i]) result = values[i];
16
17     std::cout << "SERVANT: Máximo encontrado: " << result << std::endl;
18     return result;
19 }

```

Código 5.10: Definição da implementação da faceta FindMAXWorkers

```

1 #ifndef FINDMAXWORKERSIMPL_H
2 #define FINDMAXWORKERSIMPL_H
3
4 #ifdef SCS_MICO
5     #include <CORBA.h>
6     #include <stubs/mico/FindMAXValue.h>
7 #endif
8 #include <ComponentBuilder.h>
9 using namespace scs::demos::FindMAXValue;
10
11 class FindMAXWorkersImpl : virtual public
12     POA_scs::demos::FindMAXValue::FindMAXWorkers {
13
14 private :
15     scs::core::ComponentContext* componentContext;
16     FindMAXWorkersImpl (scs::core::ComponentContext* componentContext);
17
18 public :
19     static PortableServer::ServantBase* instantiate (
20         scs::core::ComponentContext* componentContext);
21     static void destruct( void* obj );
22     //idl function definition
23     CORBA::Double findMaximum (const DoubleSeq& vector)
24         throw(CORBA::SystemException);
25 };
26 #endif

```

Todavia, no estado atual da implementação, a construção ainda feita pela montagem manual dos modelos de códigos para cada política de distribuição de dados, sendo necessários os devidos ajustes para a adaptação aos tipos da aplicação definidos nas *sub-interfaces*.

Após esse processo, tem-se a implementação do objeto que implementa a *sub-interface* externa da *GatherFacet* definido. Esse objeto é capaz de realizar a redistribuição dos dados e aplicar as políticas de agrupamento para a realização da chamada o objeto que implementa a *sub-interface* interna.

Código 5.11: Implementação da faceta FindMAXWorkers

```

1 #include "FindMAXWorkers.h"
2
3 //functional code implementation
4 CORBA::Double FindMAXWorkersImpl::findMaximum (
5     const DoubleSeq& vector ) throw ( CORBA::SystemException ){
6
7     cout << "Executando WORKERS: " << vector.length() << std::endl;
8     CORBA::Double result = vector[0];
9     //find the biggest local value
10    for (int i = 1; i < vector.length(); i++ )
11        if(result < vector[i]) result = vector[i];
12    std::cout << "WORKERS: Máximo local encontrado: " << result << std::endl;
13    //retrieves gatherfacet reference
14    std::map<std::string , scs::core::Receptacle*>::const_iterator it =
15        this->componentContext->getReceptacles().find("FindMAXReceiver");
16    FindMAXReceiver_var gather = FindMAXReceiver::_narrow(
17        ((scs::core::Receptacle*)it->second)->getConnections()->front().objref);
18    //calls gatherfacet to find the global biggest value
19    CORBA::Double result_global = gather->findMaximumValue(result);
20    std::cout << "WORKERS: Máximo GLOBAL: " << result_global << std::endl;
21
22    return result_global;
23 }

```

Descrevendo e compilando o componente

Agora que todos os artefatos de software necessários para a implementação do componente foram criados, faz-se necessário gerar o componente e para tal é necessário criar a descrição das interfaces (FindMAXReceiver, FindMAXWorkers) e receptáculos (FindMAXReceiver).

No Código 5.12 pode-se verificar a descrição da *sub-interface* interna (*FindMAX*), a descrição da implementação gerada da *sub-interface* externa do *FindMAXReceiver*, sendo estes os dois itens básicos para a definição de uma *GatherFacet*.

Implantação e Execução

Após a geração do binário do componente (*libFindMAXValue-server_mico.so*), é necessário criar um configurador, ou seja, a aplicação que vai criar instâncias do componente no contêiner e realizar as respectivas conexões entre os mesmos de acordo com a arquitetura definida na seção 5.2.1. Após carregamento e conexão, esse mesmo configurador deve iniciar a execução da aplicação.

No tutorial de uso do SCS-COLLECTIVE descrito em [36] tem-se a implementação desse configurador que realiza a carga, configuração e execução da aplicação.

Após a compilação, serão obtidos os dois binários necessários para a execução da aplicação: **libFindMAXValue-server_mico.so** (componente) e **FindMAXValue-client_mico** (configurador).

Código 5.12: Descrição de facetas e receptáculos do comp. FindMAXValue

```

1 #include <ComponentBuilder.h>
2 #include <ComponentContext.h>
3 #include "stubs/mico/deployment.h"
4 #include "FindMAXReceiverGEN.h"
5 #include "FindMAXWorkers.h"
6 extern "C" scs::core::ComponentContext * createComponent(
7     scs::core::ComponentBuilder& componentBuilder,
8     const scs::container::StringSeq& args) {
9     // criação de descrições de facetas
10    std::list<scs::core::ExtendedFacetDescription> extFacets;
11    scs::core::ExtendedFacetDescription gatherDesc_client;
12    gatherDesc_client.name = "FindMAXReceiver";
13    gatherDesc_client.interface_name =
14        "IDL:scs::demos::FindMAXValue::FindMAXReceiver:1.0";
15    gatherDesc_client.instantiator = FindMAXReceiverImpl::instantiate;
16    gatherDesc_client.destructor = FindMAXReceiverImpl::destruct;
17    extFacets.push_back(gatherDesc_client);
18    scs::core::ExtendedFacetDescription workersDesc;
19    workersDesc.name = "FindMAXWorkers";
20    workersDesc.interface_name =
21        "IDL:scs::demos::FindMAXValue::FindMAXWorkers:1.0";
22    workersDesc.instantiator = FindMAXWorkersImpl::instantiate;
23    workersDesc.destructor = FindMAXWorkersImpl::destruct;
24    extFacets.push_back(workersDesc);
25
26    // criação de descrições de receptaculo
27    std::list<scs::core::ReceptacleDescription> receptacleDescs;
28    scs::core::ReceptacleDescription receptDesc;
29    receptDesc.name = "FindMAXReceiver";
30    receptDesc.interface_name =
31        "IDL:scs/demos/FindMAXValue/FindMAXReceiver:1.0";
32    receptDesc.is_multiplex = true;
33    receptacleDescs.push_back(receptDesc);
34
35    // criação do ComponentId
36    scs::core::ComponentId componentId;
37    componentId.name = "FindMAX";
38    componentId.major_version = '1';
39    componentId.minor_version = '0';
40    componentId.patch_version = '0';
41    componentId.platform_spec = "none";
42    return
43    componentBuilder.newComponent(extFacets, receptacleDescs, componentId);
44 }

```

A execução da aplicação construída, se dá através da instanciação da infraestrutura de execução do SCS nos *hosts* que serão utilizados. Diferentemente da aplicação do *MulticastReceptacle*, os componentes *Workers* (faceta *FindMAXValueWorkers*) e o componente *Gather* (faceta *FindMAXValueReceiver*) serão carregados no mesmo contêiner, para facilitar do configurador, todavia não existe impedimento para a implantação distribuída destes. Os clientes (configuradores) poderão executar em outros *hosts* e, assim iniciar os componentes *Workers* e conectá-los ao componente *Gather*.

Vale ressaltar que a diferença entre os componentes *Workers* e o *Gather* se dá meramente pelo uso das facetas *FindMAXValueWorkers* nos *Workers* e *FindMAXValueReceiver* do componente *Gather*, pois ambos componentes expõem as duas facetas (e possuem mesmo binário).