6
Estudos de Caso

Com o intuito de experimentar a abordagem deste trabalho em cenários mais realistas, foram desenvolvidos exemplos de implantações distribuídas. Duas aplicações foram experimentadas: *CosEventService* e *MapReduce*. A primeira é mais simples e facilita a comparação entre os trechos de código com uso direto da infraestrutura de execução do SCS e os trechos necessários considerando o uso dos serviços de implantação propostos neste trabalho. A segunda aplicação é mais complexa, portanto, só serão apresentados os códigos que já usam os serviços de implantação. A partir da análise dos roteiros (do Inglês, *script*) de implantação, torna-se possível identificar o ganho de simplicidade na programação da implantação de aplicações complexas e amplamente distribuídas.

É importante destacar que não é objetivo deste capítulo avaliar quantitativamente o desempenho das aplicações, uma vez que não foram realizadas mudanças nelas. O impacto do uso deste trabalho só está relacionado à padronização dos procedimentos de implantação e não afeta a execução das aplicações, pois os componentes implantados, através da infraestrutura proposta neste trabalho, não precisam manipular as entidades virtuais, nem os planos e nem mesmo os serviços de implantação.

A versão componentizada do sistema de eventos de CORBA (*CosEventService*) foi desenvolvida por Augusto [42] e a aplicação *MapReduce* foi desenvolvida por Fonseca [61] com base no modelo arquitetural proposto por Dean e Ghemawat [62]. A análise da implantação do sistema de eventos componentizado é apresentada na Seção 6.1 e da aplicação *MapReduce* na Seção 6.2.

6.1
Serviço de Eventos CORBA

A Figura 6.1 ilustra a arquitetura do Serviço de Eventos CORBA encapsulado como componentes SCS concebida no trabalho de Augusto [42]. Nessa figura omite-se as três facetas básicas do modelo SCS para efeitos de clareza. Para efeitos didáticos, destaca-se o caminho do fluxo de eventos desde o produtor até o consumidor. As facetas e receptáculos *PushSupplier* e

PushConsumer servem para cadastrar o produtor e o consumidor de eventos, respectivamente. O produtor usa a faceta *ProxyPushConsumer* para publicar os eventos. O consumidor recebe o evento através de sua faceta *PushConsumer*.

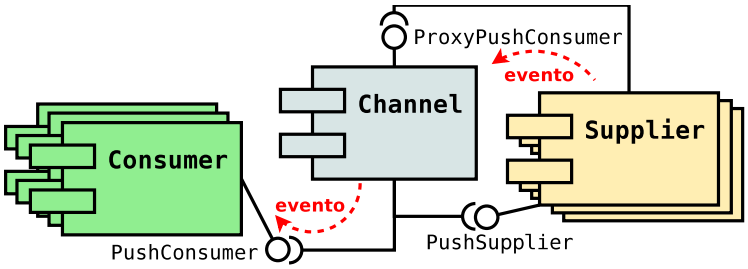

Figura 6.1: Arquitetura do Serviço de Eventos CORBA encapsulado como componentes SCS

Os exemplos dos códigos necessários para implantar uma configuração simples do serviço de eventos antes da existência dos serviços de implantação são comentados nos próximos parágrafos. Consideramos uma configuração com apenas um único componente de cada tipo (canal, consumidor e produtor) em uma única máquina e em um único contêiner, visto que o uso de vários contêineres ou várias máquinas complica ainda mais o roteiro. É importante observar que, ao usar diretamente a infraestrutura de execução do SCS, além de ser obrigatório definir o mapeamento físico, todos os passos do roteiro são executados em tempo de execução sem a abstração da fase de planejamento. Portanto, em casos de erro, o ator deve fazer toda a limpeza do ambiente, a descarga dos componentes, entre outras tarefas onerosas.

A primeira atividade é descobrir as referências remotas para os componentes da infraestrutura de execução do SCS que precisam estar executando. Especificamente para essa configuração basta encontrar um *ExecutionNode* e um *Repository* disponíveis. A partir de então, obtém-se as facetas que serão usadas para a execução das próximas atividades. O Código 6.1 exemplifica um roteiro escrito em Lua.

Código 6.1: Descoberta das referências aos componentes da infraestrutura

```
1  EN   = orb:newproxy("corbaloc::n1.tecgraf:3001/ExecutionNode")
2  REPO = orb:newproxy("corbaloc::head.tecgraf:3000/Repository")
3  enFacet        = orb:narrow(EN:getFacetByName("ExecutionNode"),
4                              "IDL:scs/execution_node/ExecutionNode:1.0")
5  installFacet   = orb:narrow(EN:getFacetByName("Installer"),
6                              "IDL:scs/execution_node/Installer:1.0")
7  repoFacet      = orb:narrow(REPO:getFacetByName("ComponentRepository"),
8                              "IDL:scs/repository/ComponentRepository:1.0")
9  -- conexão entre o ExecutionNode e Repository
10 enReceptacles  = orb:narrow(EN:getFacetByName("IReceptacles"),
11                             "IDL:scs/core/IReceptacles:1.0")
12 enReceptacles:connect( "Repository", repoFacet )
```

A segunda atividade é publicar suas implementações de componentes no *Repository*. Originalmente a infraestrutura de execução não prevê nenhum suporte a empacotamento. Considera-se apenas o envio de um arquivo comprimido com todos os artefatos necessários, conforme exemplificado no Código 6.2.

Código 6.2: Publicação das implementações dos componentes

```
13 channelId = { name = "EventChannel",
14                major_version = 1, minor_version = 0, patch_version = 0 }
15 entry_point = "scs.demos.event.EventChannel"
16 file = io.open("EventChannel.zip", "r"):read("*a")
17 helperInfo = "This component externs the EventChannel entity!"
18 repoFacet:upload(channelId, entry_point, true, file, helperInfo, "lua")
19 -- publicando componente do consumidor
20 consumerId = { name = "Consumer",
21                 major_version = 1, minor_version = 0, patch_version = 0 }
22 entry_point = "scs.demos.event.Consumer"
23 file = io.open("Consumer.zip", "r"):read("*a")
24 helperInfo = "This component externs the Consumer entity!"
25 repoFacet:upload(consumerId, entry_point, true, file, helperInfo, "lua")
26 -- publicando componente do produtor
27 supplierId = { name = "Supplier",
28                 major_version = 1, minor_version = 0, patch_version = 0 }
29 entry_point = "scs.demos.event.Supplier"
30 file = io.open("Supplier.zip", "r"):read("*a")
31 helperInfo = "This component externs the Supplier entity!"
32 repoFacet:upload(supplierId, entry_point, true, file, helperInfo, "lua")
```

Com as implementações publicadas o usuário pode criar um *Container* em um *ExecutionNode* e configurá-lo de forma a poder instanciar seus componentes. O Código 6.3 ilustra a carga dos componentes.

Código 6.3: Carga dos componentes do usuário

```
33 -- criação de um contêiner
34 propertySeq = {{ name = "language" , value = "lua" , read_only = true},
35                 { name = "machine" , value = "lua5.1" , read_only = true}}
36 container    = enFacet:startContainer("ChannelContainer", propertySeq)
37 --- conexão entre o contêiner e o nó de execução na faceta Installer
38 chanReceptacles = orb:narrow(container:getFacetByName("IReceptacles"),
39                              "IDL:scs/core/IReceptacles:1.0")
40 chanReceptacles:connect("Installer", installFacet)
41 --- faceta do contêiner por onde solicita-se o carregamento dos componentes
42 loaderFacet = orb:narrow(container:getFacetByName("ComponentLoader"),
43                          "IDL:scs/container/ComponentLoader:1.0")
44 -- carga dos três componentes num mesmo contêiner
45 channelInstance  = loaderFacet:load( channelId,  { "EventChannel_1" } )
46 consumerInstance = loaderFacet:load( consumerId, { "Consumer_1" } )
47 supplierInstance = loaderFacet:load( supplierId, { "Supplier_1" } )
```

A partir das instâncias dos componentes só resta definir a configuração inicial entre esses componentes antes de ativá-los. O Código 6.4 exemplifica a configuração e ativação dos componentes do usuário. Só depois da ativação é que se pode iniciar as ações específicas da aplicação.

Através do uso dos serviços de implantação propostos neste trabalho, a mesma configuração pode ser feita de forma mais simples, sem que o usuário precise lidar com: (i) a descoberta das instâncias remotas, (ii) o mapeamento físico ou ainda (iii) a configuração entre os componentes da infraestrutura de

Código 6.4: Configuração e ativação entre os componentes do usuário

```
48 -- configuração entre facetas e receptáculos desses componentes
49 obj        = channelInstance.cmp:getFacetByName("ProxyPushConsumer")
50 channFacet=orb:narrow(obj,"IDL:org/omg/CosEventComm/ProxyPushConsumer:1.0")
51 -- produtor conecta-se no canal para envio de eventos
52 obj             = supplierInstance.cmp:getFacetByName("IReceptacles")
53 supplierRecept = orb:narrow(obj,"IDL:scs/core/IReceptacles:1.0")
54 supplierRecept:connect("ProxyPushConsumer", channFacet)
55 obj          = channelInstance.cmp:getFacetByName("IReceptacles")
56 channRecept = orb:narrow(obj,"IDL:scs/core/IReceptacles:1.0")
57 -- produtor é conectado no canal para receber avisos de desconexão
58 obj            = supplierInstance.cmp:getFacetByName("PushSupplier")
59 supplierFacet = orb:narrow(obj,"IDL:org/omg/CosEventComm/PushSupplier:1.0")
60 channRecept:connect("PushSupplier", supplierFacet)
61 -- consumidor é conectado no canal
62 obj            = consumerInstance.cmp:getFacetByName("PushConsumer")
63 consumerFacet = orb:narrow(obj,"IDL:org/omg/CosEventComm/PushConsumer:1.0")
64 channRecept:connect("PushConsumer", consumerFacet)
65 -- ativação
66 channelInstance.cmp:startup()
67 consumerInstance.cmp:startup()
68 -- o início do produtor fará a aplicação realmente começar
69 supplierInstance.cmp:startup()
```

execução. Os códigos usando os serviços de implantação são apresentados a seguir de forma modular e serão citados na próxima Seção deste Capítulo.

O Código 6.5 exemplifica o empacotamento e a publicação. Nesse código usa-se uma instância local do serviço *Packager* (linha 1) e uma instância remota do *DeployManager* (linha 2). Usa-se o *Packager* para criar os pacotes (linhas 5–11) que são enviados a um repositório público (linhas 14–16) previamente conhecido pelo serviço *DeployManager*.

O Código 6.6 exemplifica o planejamento, a implantação e a ativação usando o mapeamento automático. Nesse código cria-se um plano único (linha 18) para todos os componentes e especifica-se, para cada componente, o seu tipo e a lista de argumentos a ser usada durante a carga (linhas 20, 21, 23, 24, 26 e 27). Em seguida, conecta-se (linhas 29–31) os componentes respeitando a arquitetura da Figura 6.1 e, por fim, procede-se a implantação e a ativação (linhas 33 e 35). É fácil perceber que ambos os Códigos 6.5 e 6.6 são mais concisos e têm foco na aplicação, sem precisar detalhar a configuração da infraestrutura.

Alternativamente, o usuário pode controlar manualmente o mapeamento. O Código 6.7 exemplifica o detalhamento sobre o uso da infraestrutura de execução e das máquinas. Esse mapeamento manual é apresentado nas linhas 29–39. Esse exemplo explicita o uso de um nó de execução (linha 33) relacionado a uma determinada máquina (linha 34) onde é criado um único contêiner (linha 35) para todos os componentes (linhas 37–39). O restante do código é idêntico àquele Código 6.6.

Ainda no Código 6.7, as linhas 29–31 mostram a importação de um arquivo com a lista de máquinas. As máquinas precisam ser descritas conforme a estrutura *HostDescription* definida no Código A.6. Um exemplo desse descritor

Código 6.5: Empacotamento e publicação usando os serviços para implantação

```
local packager = require("scs.deployer.Packager")()
local manager = orb:newproxy("corbaloc::head.tecgraf:2501/Deployer_1",
                             "IDL:scs/deployer/Manager:1.0")
-- empacotamento
dir  = "/home/amadeu/sources/events-dev/"
rockspec = dir.."/channel-1.0-0.rockspec"
chann_pkg, chann_tmpl = packager:create_from_dir( rockspec, dir )
rockspec = dir.."/supplier-1.0-0.rockspec"
suppl_pkg, suppl_tmpl = packager:create_from_dir( rockspec, dir )
rockspec = dir.."/consumer-1.0-0.rockspec"
consu_pkg, consu_tmpl = packager:create_from_dir( rockspec, dir )
-- publicação
repoList = manager:get_public_repositories()
repoList[1]:publish_pkg(chann_pkg); repoList[1]:publish_desc(chann_tmpl)
repoList[1]:publish_pkg(consu_pkg); repoList[1]:publish_desc(consu_tmpl)
repoList[1]:publish_pkg(suppl_pkg); repoList[1]:publish_desc(suppl_tmpl)
```

Código 6.6: Continuação do Código 6.5 usando o mapeamento **automático**

```
-- início do planejamento
plan = manager:create_plan()
channel = plan:create_component()  -- 1 componente canal
channel:set_id( chann_tmpl.id )
channel:set_args( "EventChannel_1" )
supplier = plan:create_component() -- 1 componente produtor
supplier:set_id( suppl_tmpl.id )
supplier:set_args( "Supplier_1" )
consumer = plan:create_component() -- 1 componente consumidor
consumer:set_id( consu_tmpl.id )
consumer:set_args( "Consumer_1" )
--- configuração entre os componentes da aplicação
channel:add_connection("PushSupplier", supplier, "PushSupplier")
channel:add_connection("PushConsumer", consumer, "PushConsumer")
supplier:add_connection("ProxyPushConsumer", channel, "ProxyPushConsumer")
-- implantação de todos componentes do plano
plan:deploy()
-- ativação de todos componentes do plano
plan:activate()
```

é apresentado no Código 6.8 e nele existem informações sobre os recursos e a localização da máquina.

Código 6.7: Continuação do Código 6.5 usando o mapeamento **manual**

```
17 -- início do planejamento
18 plan = manager:create_plan()
19 --- componentes do usuário
20 channel = plan:create_component()  -- 1 componente canal
21 channel:set_id( chann_tmpl.id )
22 channel:set_args( "EventChannel_1" )
23 supplier = plan:create_component() -- 1 componente produtor
24 supplier:set_id( suppl_tmpl.id )
25 supplier:set_args( "Supplier_1" )
26 consumer = plan:create_component() -- 1 componente consumidor
27 consumer:set_id( consu_tmpl.id )
28 consumer:set_args( "Consumer_1" )
29 --- importação da lista de descrições de máquinas
30 local importer = require("scs.deployer.descriptorhelper")()
31 machines = importer.search("hosts","cluster-tecgraf.txt")
32 --- mapeamento físico nas máquinas e na infra-estrutura de execução
33 node = plan:create_exnode()
34 node:set_host( machines["n1"] ) -- associando a uma máquina física
35 container = plan:create_container()
36 container:set_node( node )      -- associando a um nó de execução
37 channel:set_container( container )  -- associando a um contêiner
38 supplier:set_container( container ) -- associando a um contêiner
39 consumer:set_container( container ) -- associando a um contêiner
40 --- configuração entre os componentes da aplicação
41 channel:add_connection("PushSupplier", supplier, "PushSupplier")
42 channel:add_connection("PushConsumer", consumer, "PushConsumer")
43 supplier:add_connection("ProxyPushConsumer", channel, "ProxyPushConsumer")
44 -- implantação de todos componentes do plano
45 plan:deploy()
46 -- ativação de todos componentes do plano
47 plan:activate()
```

Código 6.8: Exemplo em Lua da descrição de uma máquina

```
1 { name = "n1", ip = "192.168.1.1", port = 2049,
2   languages = { "lua","java" },
3   resources = {
4     cpu = 1800, mem = 512, bandwith = 100, os = "linux", arch = "x86"
5   },
6 }
```

6.2 Aplicação MapReduce

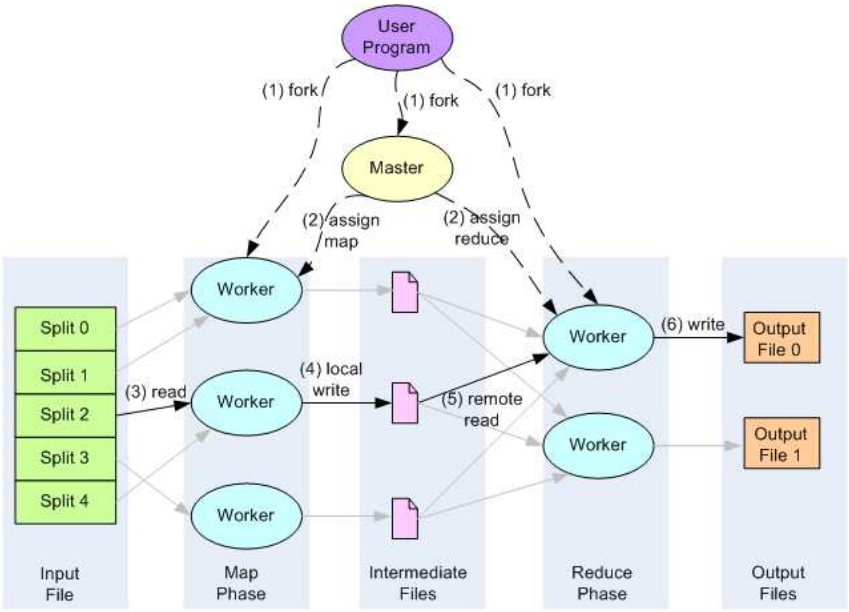

Figura 6.2: Arquitetura da aplicação MapReduce (fonte: [62])

O estilo arquitetural conhecido como *MapReduce* é usado na distribuição do processamento de grandes massas de dados. A Figura 6.2 ilustra uma visão geral dos seus constituintes. Destacam-se dois componentes: *master* e *worker*. O *master* é o componente que controla todo o fluxo de tratamento dos dados, usando os componentes *worker* ociosos e atribuindo-lhes tarefas de *map* (mapeamento) ou *reduce* (redução). As tarefas *map* processam um arquivo de entrada e geram tuplas {chave,valor} intermediárias. Durante a tarefa *reduce*, essas tuplas são combinadas com base na similaridade das chaves. Um dos usos dessa arquitetura é na contagem de ocorrências de palavras (ou conjunto delas) em grandes arquivos de *log* gerados em servidores *web*.

A aplicação *MapReduce* usada neste trabalho foi implementada por Fonseca [61] para validação do seu trabalho no monitoramento de componentes SCS. A Figura 6.3 ilustra a arquitetura dessa implementação baseada em componentes. Em particular, o componente *reporter* é usado para registrar mensagens de depuração e o canal de eventos serve para que o *master* consuma os eventos publicados pelos *workers*, aumentando o desacoplamento entre os componentes. Dessa forma, é interessante reusar o plano de implantação já apresentado nos Códigos 6.5 e 6.6. Assim, os próximos códigos são complementares e constituem um novo plano de implantação para componentes *master, worker* e *reporter*.

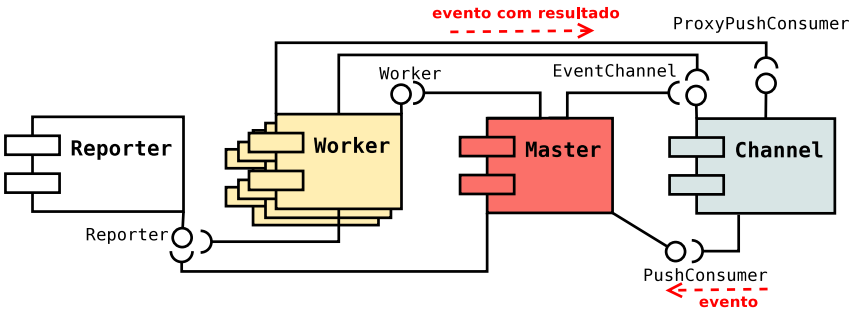

Figura 6.3: Diagrama dos componentes da aplicação MapReduce

Como parte do roteiro de implantação da aplicação *MapReduce*, primeiro é preciso empacotar (linhas 37–43) e publicar (linhas 46–48) as implementações dos componentes *master*, *worker* e *reporter* em um repositório disponível (linha 45), conforme ilustrado no Código 6.9.

Código 6.9: Continuação do Código 6.6, empacotamento e publicação dos componentes do MapReduce

```
36 -- empacotamento do mapreduce
37 dirn = "/home/amadeu/scs/src/java/scs/demos/mapreduce/"
38 master_rockspec    = dirn.."/master-1.0-0.rockspec"
39 master_pkg, master_tmpl = packager:create_from_dir(master_rockspec, dirn)
40 worker_rockspec    = dirn.."/worker-1.0-0.rockspec"
41 worker_pkg, worker_tmpl = packager:create_from_dir(worker_rockspec, dirn)
42 report_rockspec = dirn.."/reporter-1.0-0.rockspec"
43 report_pkg, report_tmpl = packager:create_from_dir(report_rockspec,dirn)
44 -- publicação do mapreduce
45 repoList = manager:get_public_repositories()
46 repoList[1]:publish_pkg(master_pkg); repoList[1]:publish_desc(master_tmpl)
47 repoList[1]:publish_pkg(worker_pkg); repoList[1]:publish_desc(worker_tmpl)
48 repoList[1]:publish_pkg(report_pkg); repoList[1]:publish_desc(report_tmpl)
```

Em seguida, pode-se criar um novo plano de implantação (linha 50) para gerir os componentes a serem implantados, conforme exemplificado no Código 6.10. Os componentes contidos nesse plano podem se conectar a componentes de outros planos previamente implantados, como o serviço de eventos (linhas 59, 60, 67 e 68). Assim, o *DeployManager* será o responsável por implantar os componentes dependentes. Ainda no Código 6.10 pratica-se o mapeamento manual em alto nível, definindo um máximo de 1 componente por contêiner (linha 51) e 10 contêineres por máquina (linha 52). Por fim, solicita-se a implantação de 200 componentes do tipo *worker* (linhas 62–70) e assim serão implantados 10 componentes por máquina (cada um com seu próprio contêiner) demandando uso de um total de 20 máquinas.

Após concluir o planejamento, pode-se concretizar a implantação do novo plano (linha 71) e ativar os componentes (linha 72) conforme exemplificado no Código 6.11. Assim a aplicação entrará em execução e será possível executar as ações específicas da aplicação como exemplificado no Código 6.12. Nesse código exemplifica-se: a busca da faceta *Worker* nos componentes do tipo *worker*

Código 6.10: Criando um novo plano para implantar 200 componentes *workers* do MapReduce

```
49 -- planejamento do mapreduce
50 other_plan = manager:create_plan()
51 other_plan:set_max_container_usage( 1 ) -- 1 componente por contêiner
52 other_plan:set_max_node_usage( 10 )     -- 10 contêineres por nó
53 reporter = other_plan:create_component()-- 1 componente reporter
54 reporter:set_id( report_tmpl.id )
55 master = other_plan:create_component()  -- 1 componente master
56 master:set_id( master_tmpl.id )
57 master:add_connection( "Reporter", reporter, "Reporter" )
58 -- configuração do master com o canal de eventos implantado anteriormente
59 master:add_connection( "EventChannel", channel, "EventChannel" )
60 channel:add_connection( "PushConsumer", master, "PushConsumer" )
61 local workers = {}  -- demanda por 200 componentes distribuídos
62 for i=1,200 do      -- em 10 contêiners, resultará em 20 máquinas usadas
63   workers[i] = other_plan:create_component()
64   workers[i]:set_id( worker_tmpl.id )
65   -- configuração entre componentes do usuário
66   workers[i]:add_connection("Reporter",reporter,"Reporter")
67   workers[i]:add_connection("EventChannel",channel,"EventChannel")
68   workers[i]:add_connection("ProxyPushConsumer",channel,"ProxyPushConsumer")
69   master:add_connection( "Worker", workers[i], "Worker" )
70 end
```

(linha 81) para definir propriedades específicas desse tipo de componente (linha 86); e a busca da faceta *Master* no componente do tipo *master* (linha 90) para solicitar o início do processamento da aplicação (linha 92). Ao término da execução da aplicação, pode-se desativar (linhas 94 e 96) e remover (linhas 95 e 97) todos componentes conforme exemplifcado no Código 6.13.

Código 6.11: Implantação e ativação dos componentes do MapReduce

```
71 other_plan:deploy()   -- implantação dos componentes do plano do mapreduce
72 other_plan:activate() -- ativação
```

6.3 Considerações finais

A partir dos experimentos apresentados entendemos que os serviços para implantação distribuída permitem ganhos de simplicidade na confecção dos roteiros de implantação motivando inclusive seu reuso. Além disso, o fato de não obrigar a gerência manual da infraestrutura de execução desonera o administrador de tarefas repetitivas e propensas a erros. Por outro lado, quando da ocorrência de erros na execução ou mesmo na implantação, a interface de programação disponibilizada pelos serviços facilita a limpeza do ambiente de execução e a restauração da configuração anterior.

Esses experimentos também demonstram que apesar do usuário não mais lidar diretamente com a infraestrutura de execução, isso não causa prejuízo nem mudanças na arquitetura dos componentes da sua aplicação. Isso é possível pois preserva-se as abstrações dos componentes do usuário e flexibiliza-se o

Código 6.12: Ações específicas da aplicação do MapReduce

```
73 -- ações específicas da aplicação do usuário
74 local propFile = "/home/amadeu/scs/scripts/execute/mapReduce.properties"
75 local logName = "/tmp/reporter.debug"
76 reporter_facet = orb:narrow(reporter:get_facet("Reporter"),
77                              "IDL:scs/demos/mapreduce/Reporter:1.0")
78 reporter_facet:open( logName, 2 )
79 reporter_facet:report( 2, "DeploymentManager :: iniciando mapreduce")
80 for i=1,200 do
81   worker_facet = orb:narrow(workers[i]:get_facet("Worker"),
82                             "IDL:scs/demos/mapreduce/Worker:1.0")
83   -- nome de host e contêiner para ajudar no debug
84   hostname      = workers[i]:get_container():get_node():get_host().name
85   containername = workers[i]:get_container():get_nickname()
86   worker_facet:start(propFile, hostname, containername)
87 end
88 -- envio de propriedades sobre:
89 -- nome de arquivo de entrada, tamanho, divisões, funções map e reduce
90 master_facet = orb:narrow(master:get_facet("Master"),
91                           "IDL:scs/demos/mapreduce/Master:1.0")
92 master_facet:submitJob(propFile)
93 reporter_facet:report( 2,"MapReduce executou com sucesso!!")
```

Código 6.13: Desativação e remoção dos planos de implantação

```
94 other_plan:deactivate() -- desativação do plano do mapreduce
95 other_plan:undeploy()   -- remoção  do plano do mapreduce
96 plan:deactivate() -- desativação do plano do serviço de eventos
97 plan:undeploy()   -- remoção do plano do serviço de eventos
```

seu mapeamento na infraestrutura de execução e nas máquinas. Portanto, os serviços aqui propostos não oneram o desempenho das aplicações que são configuradas por eles, já que não interferem em sua execução.