6
Avaliação

O principal objetivo deste trabalho é avaliar o uso da linguagem Java na implementação de mecanismos de adaptação dinâmica em sistemas baseados em componentes, comparando uma solução desenvolvida nesta linguagem com outras soluções de modo a gerar dados que ajudem aos desenvolvedores a optar por uma linguagem de programação e um conjunto de ferramentas adequado às suas necessidades. Três aspectos são especialmente importantes na avaliação de mecanismos desse tipo: desempenho, flexibilidade e facilidade de uso. Recursos de adaptação geralmente impõem alguma sobrecarga no desempenho das aplicações, de forma que, ao considerar a sua adoção, o desenvolvedor precisa avaliar se o impacto desta sobrecarga é aceitável. Caso a sobrecarga de desempenho não represente uma barreira, cabe ao desenvolvedor selecionar uma ferramenta que ofereça mecanismos de adaptação simples de usar e ao mesmo tempo flexíveis o suficiente para atender suas necessidades.

No restante deste capítulo buscamos comparar a solução desenvolvida com outros trabalhos para avaliar alguns de seus aspectos, tentando sempre extrair indicadores de desempenho, flexibilidade e facilidade de uso, além de identificar sempre que possível os obstáculos encontrados durante o seu desenvolvimento. Na seção 6.1 avaliamos o mecanismo de *binding* dinâmico implementado, usando como referência para a comparação a versão convencional do SCS Java e o sistema *LuaORB*. Inicialmente, mensuramos o impacto deste mecanismo sobre o desempenho das aplicações, comparando o tempo de execução de aplicações idênticas implementadas com as versões adaptável e convencional do SCS. Em seguida, avaliamos a facilidade de uso dos mecanismos de criação de componentes e manipulação de conexões, utilizando como referência a versão convencional do SCS, implementada com *binding* estático entre os componentes, e o *LuaORB*, que assim como a versão adaptável do SCS, implementa esse *binding* de maneira dinâmica. Buscamos também extrair indicadores de flexibilidade da solução, comparando as possibilidades oferecidas

pela versão adaptável do SCS com as oferecidas pelo *LuaORB*. Finalmente, analisamos os desafios encontrados na implementação do *binding* dinâmico em Java tomando novamente como referência o *LuaORB*, que implementou o mesmo mecanismo em Lua.

Na seção 6.2 avaliamos o contêiner dinâmico e os mecanismos de adaptação implementados, tendo como referência o sistema *LuaCCM*. Comparamos inicialmente a facilidade de uso dos mecanismos de adaptação nos dois sistemas, avaliamos a flexibilidade das duas soluções, comparando suas possibilidades, e por fim identificamos os desafios encontrados na implementação desses mecanismos. Na seção 6.3 comparamos o uso das linguagens Lua e Java para efetuar adaptações através da versão adaptável do sistema SCS desenvolvida neste trabalho.

6.1
Mecanismo de binding dinâmico

Conforme visto no capítulo 4, o primeiro passo para a implementação dos mecanismo de adaptação dinâmica no sistema SCS foi a implementação de uma versão desse sistema baseada em *esqueletos dinâmicos*, onde a ligação entre os componentes e a verificação de suas interfaces são efetuadas em tempo de execução. Os principais objetivos dessa implementação eram diminuir ao máximo o grau de acoplamento entre a implementação do componente e a sua estrutura e criar um nível de indireção entre os componentes que permitisse a introdução de mecanismos de adaptação que pudessem interceptar e manipular as mensagens trocadas entre eles. Alguns dos recursos de adaptação propostos, como a adição de facetas, receptáculos e interceptadores, poderiam ser implementados sem o mecanismo de *binding* dinâmico, utilizando ao invés dele um esquema de geração de código de esqueleto que disponibilizasse pontos de extensão. Esta saída porém, acarretaria em uma grande perda de flexibilidade pela solução. Em primeiro lugar, não seria possível alterar as interfaces das portas em tempo de execução, pois elas estariam ligadas estaticamente ao código gerado. Além disso, a ligação estática dificultaria a implementação dos mecanismos de verificação de tipos baseados em compatibilidade estrutural.

O mecanismo de *binding* dinâmico simplificou bastante o processo de criação dos componentes mas, conforme esperado, implicou em uma sobrecarga no desempenho das aplicações. Nas próximas sessões utilizamos a implementação convencional do SCS Java e o *LuaORB* como referências para

avaliar o desempenho, a facilidade de uso e a flexibilidade desta solução. Na seção 6.1.4, descrevemos uma série de pontos importantes identificados, relacionados à utilização da linguagem Java na implementação de um mecanismo de *esqueletos dinâmicos*, traçando um paralelo com o uso da linguagem Lua na implementação de um mecanismo similar no *LuaORB*.

6.1.1 Desempenho

Para avaliar o impacto no desempenho causado pelo uso do SCS com *binding* dinâmico, implementamos aplicações idênticas utilizando esta implementação e a implementação convencional do SCS. Em seguida, testamos as aplicações, medindo o tempo gasto por suas principais operações diversas vezes até obter um volume considerável de medições. Os dados foram tratados estatisticamente para eliminar os valores que se afastaram do intervalo de confiança e assim gerar médias válidas. Para extrair informações mais precisas destas medições, repetimos esses testes em três diferentes cenários: componentes instanciados em um único processo de uma máquina, componentes instanciados em processos separados de uma mesma máquina e componentes instanciados em máquinas distribuídas em uma mesma rede local. No primeiro cenário, buscamos medir a sobrecarga da maneira mais isolada possível, reduzindo o tempo gasto com a comunicação entre processos e, consequentemente, aumentando a representatividade da sobrecarga frente o tempo total. No segundo cenário criamos um ambiente próximo do ambiente típico das aplicações nas quais o sistema é focado, evitando porém a interferência da variação de disponibilidade de recursos da rede. O terceiro cenário representa um cenário típico de uma aplicação distribuída. Os testes locais foram realizados em um notebook Apple Macbook com processador de dois núcleos Intel Core 2 Duo de 2GHz, 1GB de memória RAM e sistema operacional Mac OS X versão 10.4 *Tiger*. Nos testes distribuídos, além do notebook, foi utilizado um computador de mesa Apple Mac mini com processador de dois núcleos Intel Core Duo de 1.83 GHz, 1GB de memória RAM e e sistema operacional Mac OS X versão 10.4 *Tiger*, ambos conectados através de cabos a uma rede local de velocidade 100 Mbps. Em cada cenário de cada aplicação efetuamos em torno de 1000 execuções de cada método medido. Um dos problemas encontrados na medição do tempo de execução dos exemplos foi a falta de mecanismos que permitam efetuar essa medição desconsiderando o tempo gasto pela máquina virtual de Java e demais processos da máquina. Para contornar essa limitação,

utilizamos a interface nativa de Java (JNI) para acessar operações do sistema operacional que permitam consultar o tempo de processamento gasto por um processo de maneira isolada.

Os exemplos utilizados nos testes foram escolhidos para tentar medir de forma isolada o impacto causado pelo sistema em diversos aspectos das aplicações. O primeiro exemplo, chamado de *PingPong*, é composto por um conjunto de componentes conectados que, após iniciados, efetuam chamadas *ping* entre si, como ilustra a figura 6.1. Ao receber uma chamada *ping*, um componente efetua uma chamada *pong* em cada um dos outros componentes conectados em seu receptáculo, que decrementam um contador interno e reiniciam o processo com novas chamadas *ping*. Para avaliar o desempenho dessa aplicação, medimos o tempo decorrido entre o envio da chamada *ping*, através do receptáculo do componente, e o recebimento da chamada *pong* correspondente em sua faceta. Essa aplicação foi selecionada para medir o impacto no desempenho causado nas conexões entre facetas e receptáculos pelo contêiner dinâmico, pois como as operações não recebem parâmetros nem retornam nenhum tipo, não há sobrecarga de conversão e tráfego de tipos. Ela faz parte dos exemplos de uso da implementação Java padrão do sistema SCS.

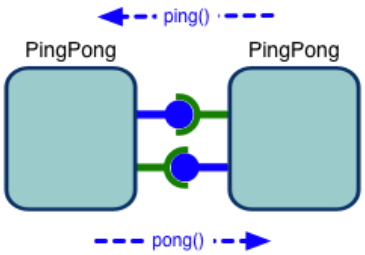

Figura 6.1: Aplicação *PingPong*

A segunda aplicação, chamada de *PhoneBook*, é uma implementação simplificada de um serviço de catálogo de telefones que usa um componente externo para efetuar a autenticação de seus clientes. As operações *add*, que adiciona um novo telefone, e *look*, que consulta um telefone existente, foram testadas, sendo que o tempo foi medido entre o momento que antecede a invocação das operações e o que sucede o seu retorno. Esse exemplo é similar em complexidade ao *PingPong*, porém adiciona o tráfego de tipos básicos e mais um nível de conexões. A figura 6.2 ilustra o funcionamento da aplicação *PhoneBook*.

A aplicação *Echo* por sua vez, é composta de um componente com uma única faceta, que declara um série de operações que recebem como parâmetros todos os tipos IDL suportados e retornam o mesmo tipo recebido. A

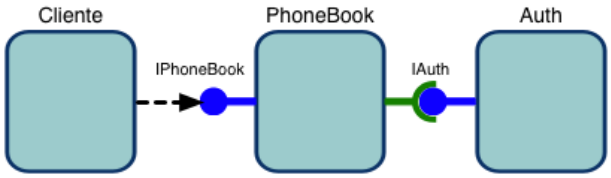

Figura 6.2: Aplicação *PhoneBook*

implementação dessa aplicação apenas "ecoa"os valores recebidos, retornando-os imediatamente. O objetivo deste exemplo é mensurar a sobrecarga imposta no tráfego e conversão dinâmica dos diversos tipos IDL suportados. Mais uma vez, o tempo foi medido entre o momento que antecede a invocação das operações e o que sucede o seu retorno. A listagem 6.1 exibe a interface do componente *Echo*.

```
1  module tests {
2          module echo {
3                  exception EmptyExcept {};
4                  exception FullExcept {
5                          unsigned short ushort_value;
6                          string string_value;
7                          octet octet_value;
8                          short short_value;
9                          Object objref_value;
10                         long long_value;
11                         any any_value;
12                         double double_value;
13                         char char_value;
14                         float float_value;
15                         boolean boolean_value;
16                         unsigned long ulong_value;
17                 };
18
19                 struct FullStruct {
20                         unsigned short ushort_value;
21                         string string_value;
22                         octet octet_value;
23                         short short_value;
24                         Object objref_value;
25                         long long_value;
26                         any any_value;
27                         double double_value;
28                         char char_value;
29                         float float_value;
30                         boolean boolean_value;
31                         unsigned long ulong_value;
32                 };
```

```
33
34              typedef sequence<unsigned short> ushortSeq;
35              typedef sequence<string> stringSeq;
36              typedef sequence<octet> octetSeq;
37              typedef sequence<short> shortSeq;
38              typedef sequence<FullStruct> structSeq;
39              typedef sequence<Object> objrefSeq;
40              typedef sequence<long> longSeq;
41              typedef sequence<any> anySeq;
42              typedef sequence<double> doubleSeq;
43              typedef sequence<char> charSeq;
44              typedef sequence<float> floatSeq;
45              typedef sequence<boolean> booleanSeq;
46
47              interface Echo {
48                      void echo_void();
49                      unsigned short echo_ushort(
50                              in unsigned short value);
51                      ushortSeq echo_ushorts(
52                              in ushortSeq value);
53                      string echo_string(in string value);
54                      stringSeq echo_strings(
55                              in stringSeq value);
56                      octet echo_octet(in octet value);
57                      octetSeq echo_octets(in octetSeq value);
58                      short echo_short(in short value);
59                      shortSeq echo_shorts(in shortSeq value);
60                      FullStruct echo_struct(
61                              in FullStruct value);
62                      structSeq echo_structs(
63                              in structSeq value);
64                      Object echo_objref(in Object value);
65                      objrefSeq echo_objrefs(
66                              in objrefSeq value);
67                      long echo_long(in long value);
68                      longSeq echo_longs(in longSeq value);
69                      any echo_any(in any value);
70                      anySeq echo_anys(in anySeq value);
71                      double echo_double(in double value);
72                      doubleSeq echo_doubles(
73                              in doubleSeq value);
74                      char echo_char(in char value);
75                      charSeq echo_chars(in charSeq value);
76                      float echo_float(in float value);
77                      floatSeq echo_floats(in floatSeq value);
78                      boolean echo_boolean(in boolean value);
79                      booleanSeq echo_booleans(
```

```
80                                  in booleanSeq value);
81                          unsigned long echo_ulong(
82                                  in unsigned long value);
83                          void raise_empty() raises (EmptyExcept);
84                          void raise_full() raises (FullExcept);
85                  };
86          };
87  };
```

Listing 6.1: Interface do componente *Echo*

6.1.2

Resultados dos testes de desempenho

A figura 6.3 exibe uma tabela com os resultados dos testes realizados. Para cada aplicação são exibidos os tempos médios em milisegundos das chamadas na implementação padrão do SCS e na versão adaptável. Em seguida é exibida a diferença percentual entre as duas médias, sendo os valores negativos - onde a versão adaptável foi mais rápida que a padrão - exibidos em verde, os valores entre 0 e 5% exibidos em azul e os superiores a 5% em vermelho e negrito. A aplicação *Echo* não conta com medidas em dois processos por ser composta por apenas um componente e ter o cliente conectado diretamente à faceta dele, sem utilizar receptáculos para isso.

Ao contrário do esperado, os valores auferidos nos cenários de processo único e de múltiplos processos não apresentaram diferenças significativas. Isso ocorreu porque nenhuma das implementações do SCS utilizadas nos testes conta com otimizações para evitar os mecanismos de comunicação distribuída no caso de componentes que estejam hospedados no mesmo processo. Por outro lado, a diferença de desempenho das duas implementações ficou substancialmente reduzida no cenário de máquinas distribuídas, conforme o esperado. Isso se deve ao fato de que a sobrecarga imposta pelo sistema representa uma parcela reduzida do tempo total de uma invocação típica em um ambiente distribuído. Com a introdução de processamento mais complexo na implementação dos métodos, essa proporção tende a diminuir ainda mais.

A partir dos resultados das aplicações *PingPong* e *PhoneBook*, podemos concluir que a sobrecarga imposta pelos mecanismos de conexão do sistema não impõem um obstáculo representativo para a adoção do sistema. Nas medidas efetuadas dentro de uma mesma máquina a sobrecarga variou entre 3% e 8%,

PingPong

1 Processo			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
ping	933,85	1006,20	7,75

2 Processos			
Operação	SCS	SCSAdapt	Delta
ping	1227,95	1310,56	6,73

2 Maquinas			
Operação	SCS	SCSAdapt	Delta
ping	1183,56	1238,26	4,62

PhoneBook

1 Processo			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
add	482,96	521,24	7,93
look	384,51	396,62	3,15

2 Processos			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
add	536,86	562,04	4,69
look	396,05	415,57	4,93

2 Maquinas			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
add	820,79	837,98	2,09
look	604,15	621,44	2,86

Echo

1 Processo			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
echo_ushort	197,63	197,08	-0,28
echo_ushorts	206,37	207,43	0,51
echo_string	208,45	209,37	0,44
echo_strings	211,82	214,03	1,04
echo_octet	196,38	196,68	0,15
echo_octets	203,29	204,21	0,46
echo_short	200,44	200,41	-0,02
echo_shorts	205,53	207,18	0,80
echo_struct	365,49	392,18	7,30
echo_structs	363,79	393,75	8,23
echo_objref	307,63	321,42	4,48
echo_objrefs	307,47	319,83	4,02
echo_long	207,93	214,28	3,05
echo_longs	203,52	205,67	1,05
echo_any	223,58	228,37	2,14
echo_anys	242,66	245,62	1,22
echo_double	205,10	208,43	1,62
echo_doubles	208,81	211,02	1,06
echo_char	210,32	212,47	1,02
echo_chars	215,47	218,65	1,47
echo_float	201,77	203,97	1,09
echo_floats	204,63	205,91	0,63
echo_boolean	198,92	198,78	-0,07
echo_booleans	202,94	204,39	0,71
echo_ulong	202,39	203,31	0,45

2 Maquinas			
Operação	SCS (ms)	SCSAdapt (ms)	Variação (%)
echo_ushort	300,12	295,60	-1,51
echo_ushorts	316,70	307,21	-3,00
echo_string	314,42	308,28	-1,95
echo_strings	321,69	330,33	2,69
echo_octet	307,08	307,79	0,23
echo_octets	309,68	302,59	-2,29
echo_short	318,34	315,61	-0,86
echo_shorts	327,98	304,18	-7,26
echo_struct	552,33	546,45	-1,06
echo_structs	565,74	554,11	-2,05
echo_objref	485,20	481,81	-0,70
echo_objrefs	487,01	474,31	-2,61
echo_long	334,36	332,75	-0,48
echo_longs	313,05	300,94	-3,87
echo_any	349,06	348,07	-0,28
echo_anys	386,68	381,83	-1,25
echo_double	310,54	308,23	-0,74
echo_doubles	325,02	315,48	-2,93
echo_char	331,80	342,46	3,21
echo_chars	352,23	359,27	2,00
echo_float	309,89	323,19	4,29
echo_floats	306,96	318,85	3,88
echo_boolean	300,17	303,67	1,16
echo_booleans	304,26	306,83	0,84
echo_ulong	325,57	327,36	0,55

Figura 6.3: Resultados dos testes de desempenho

um intervalo tolerável que estava dentro do esperado. No cenário distribuído, essa sobrecarga foi reduzida a um intervalo de 2,1% a 4,6%, o que mostra que ela tem pouca representatividade frente a sobrecarga normalmente imposta por mecanismos de comunicação distribuída. As medidas obtidas na aplicação *Echo* por sua vez, mostram que o tráfego de tipos de maneira geral introduz um sobrecarga muito pequena, que tende a desaparecer em cenários distribuídos, como ilustra a figura 6.4. A exceção ficou por conta dos tipos estruturados que no cenário local apresentou uma sobrecarga próxima a 8%. Esse valor elevado se deve ao uso intensivo dos mecanismos reflexivos de Java para a conversão dos tipos e interpretação das assinaturas dos métodos.

É importante ressaltar que apesar de termos tomado como referência de desempenho a implementação em Java do SCS, podemos considerar que os tempos de execução de aplicações desenvolvidas com ela são equivalente ao de aplicações desenvolvidas com o ORB Java desenvolvido pela Sun, a partir do

qual ela foi implementada. Essa conclusão se deve ao fato de que depois que os componentes são conectados, nenhum mecanismo do SCS interfere na sua comunicação. O ORB da Sun é uma implementação da arquitetura CORBA em Java que é distribuída juntamente com a plataforma Java desde a versão 1.2 de 1998, o que faz dela uma implementação relativamente madura e confiável.

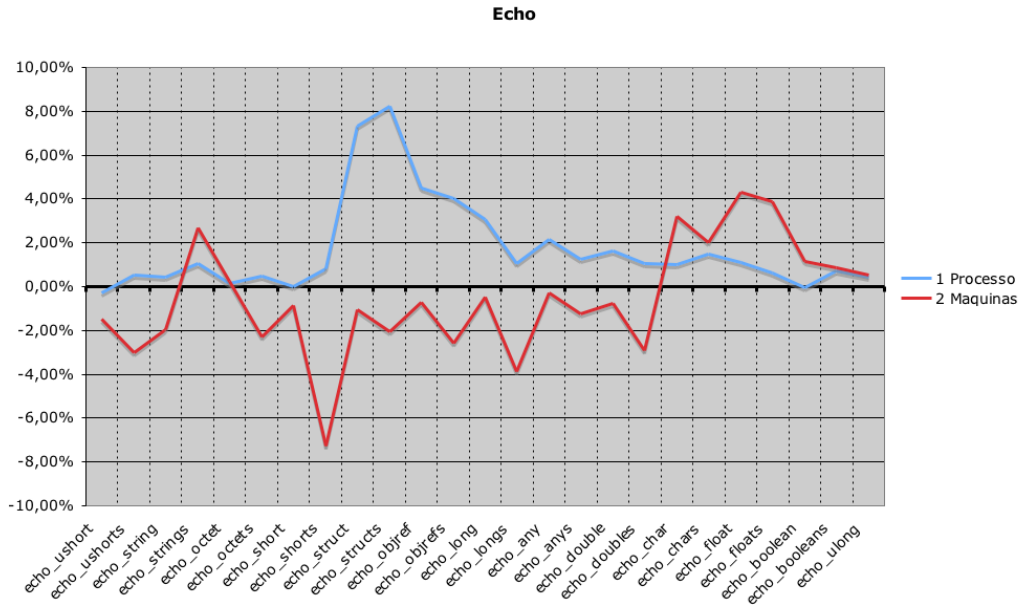

Figura 6.4: Variação percentual do tempo de execução médio da aplicação *Echo* em dois cenários

6.1.3

Facilidade de uso

Um requisito que consideramos fundamental na implementação do SCS com *binding* dinâmico é que os mecanismos de criação de componentes fossem amigáveis do ponto de vista do desenvolvedor, de forma que o uso deste sistema não represente uma barreira para usuários iniciantes. Para isso, partimos da implementação padrão do SCS Java, procurando identificar e eliminar os aspectos que pudessem aumentar a complexidade do desenvolvimento ou impor restrições às implementações dos componentes.

A implementação Java padrão do SCS segue aproximadamente o modelo de desenvolvimento definido por implementações da arquitetura CORBA nesta linguagem, que consiste em especificar as interfaces dos objetos em IDL, utilizar ferramentas para gerar código que encapsule os mecanismos de comunicação dos objetos definidos e estender o código gerado com a implementação desses

objetos. O cliente dos objetos também utiliza código gerado a partir da definição dos objetos para estabelecer a conexão com eles e consumir os seus serviços. A primeira limitação identificada neste modelo é a necessidade de que a implementação dos componentes estenda uma classe gerada para o objeto. Como Java não aceita heranças múltiplas, essa obrigatoriedade elimina a possibilidade das implementações estenderem outras classes que façam sentido para o desenvolvedor. Na implementação adaptável decidimos por eliminar essa limitação, aceitando implementações Java sem nenhuma restrição. A compatibilidade entre a implementação e a definição do componente pode ser garantida no momento de sua carga através da simples verificação das operações definidas, feita com o auxílio da API reflexiva da linguagem Java. Uma outra dificuldade que eliminamos neste processo é a necessidade de gerar código para implementar os componentes. Como o SCS é um sistema independente de linguagem de programação, não é possível abrir mão de definições IDL para descrever seus componentes, mas através da incorporação de um *parser* desta linguagem ao sistema, foi possível substituir esse passo do processo de desenvolvimento por uma chamada ao contêiner do componente passando a definição do componente em IDL como parâmetro. A geração de código para o cliente ainda não foi eliminada pois não consideramos razoável supor que todos os clientes conheçam interfaces equivalentes aos componentes oferecidos.

A flexibilidade proporcionada pelo mecanismo de verificações de tipo da linguagem Lua permite que a utilização de componentes externo seja bastante simplificada quando comparada com o mesmo processo utilizando o SCS Java com *esqueletos dinâmicos*. Enquanto no *LuaORB* é possível criar um *proxy* para um componente externo a partir da sua interface e do IOR que o localiza, no SCS Java é necessário também que exista uma interface Java equivalente carregada no *classloader* do componente. Essa restrição é particularmente trabalhosa de ser atendida no caso de tipos descobertos em tempo de execução, onde a interface equivalente tem que ser escrita ou gerada e carregada para que seja possível utilizar o novo componente.

6.1.4 Flexibilidade

A introdução de um nível de indireção capaz de interceptar e manipular as mensagens trocadas entre a implementação do componente e os demais com-

ponentes externos, possibilitou o desenvolvimento de uma série de mecanismos que flexibilizaram o processo de implementação de componentes. Uma vez que a implementação do componente não está mais conectada diretamente aos componentes externos, é possível que ela seja substituída mais facilmente em tempo de execução. As interfaces das facetas do componente também podem ser alteradas a qualquer momento, bastando apenas que exista uma implementação equivalente para a qual o *esqueleto dinâmico* encaminhe as chamadas recebidas. Também foi viabilizada a implementação de mecanismos de interceptação, que processam e manipulam as mensagens antes que elas sejam encaminhadas para a implementação do componente e depois de seu retorno.

Um dos pontos críticos para o elevado grau de flexibilidade atingido por esta solução foi o mecanismo de verificação de compatibilidade ente as implementações e a estrutura dos componentes. Assim como no *LuaORB*, esse mecanismo é baseado apenas na compatibilidade estrutural dos tipos, isto é, para uma implementação de uma faceta ser aceita, basta que para toda operação declarada na interface dessa faceta, exista uma operação com assinatura equivalente na implementação fornecida. Duas assinaturas de operações são equivalentes se tem o mesmo nome e o mesmo número, tipo e ordem de parâmetros de entrada e de saída. Apesar dos mecanismos de verificação similares, o *LuaORB* conta com um nível adicional de flexibilidade derivado da linguagem Lua. Como em Lua as funções são tratadas como valores de primeira classe, elas podem ser manipuladas como qualquer outro tipo de valor da linguagem. Desta forma a implementação do componente pode ser manipulada diretamente, com a adição ou substituição de funções. Um dos benefícios derivados dessa característica da linguagem é a possibilidade de se fornecer implementações parciais para os componentes, que podem ser completadas posteriormente. Essa possibilidade é bastante útil na fase de implementação e teste das aplicações, pois permite testar um subconjunto das funcionalidades de um componente antes de implementá-lo como um todo.

6.1.5
Desafios de implementação

Os dois pontos mais críticos da implementação dos *esqueletos dinâmicos* no SCS foram os mecanismos de invocação dinâmica das facetas e dos receptáculos do componente. No caso das facetas, o mecanismo de invocação dinâmica recebe chamadas arbitrárias de objetos CORBA remotos e deve ser

capaz de, em tempo de execução, consultar a interface da faceta, decidir se está apto a responder a chamada recebida, efetuar uma chamada equivalente à implementação local da faceta de maneira reflexiva e retornar ao objeto CORBA o resultado da chamada local efetuada. Instanciar um objeto CORBA para receber as chamadas da faceta sem especificar a interface que ele implementa e inspecionar a assinatura das chamadas remotas recebidas são operações relativamente simples de se implementar, pois são contempladas pelo mecanismo de *Dynamic Skeleton Interface* (DSI) de CORBA. Para decidir se a faceta está apta a responder as chamadas recebidas, os *esqueletos dinâmicos* mantêm estruturas de dados com a descrição detalhada das interfaces que eles implementam. Ao receber uma chamada, o esqueleto consulta um mapa associativo, que liga o nome da operação a uma estrutura com o seu tipo de retorno e com o tipo de cada um de seus argumentos. A chamada à implementação local da faceta é efetuada com a ajuda da API reflexiva da linguagem Java, que permite consultar e invocar os métodos implementados por um objeto sem conhecer a sua classe. É importante ressaltar que ainda não foram implementados mecanismos de verificação que garantam que a implementação da faceta tenha todos os métodos declarados por sua interface, essa responsabilidade atualmente é do desenvolvedor do componente.

Para permitir que a implementação das facetas dos componentes utilizem componentes externos conectados a seus receptáculos, criamos um mecanismo similar ao das facetas, porém no sentido oposto. A implementação do componente efetua uma chamada a um *proxy* local fornecido pelo esqueleto que, a partir da assinatura da chamada recebida, efetua uma chamada equivalente ao objeto remoto conectado no receptáculo e devolve o resultado recebido. Para criar um *proxy* local equivalente a um objeto remoto é necessário fornecer uma interface Java equivalente à interface desse objeto. Consideramos razoável assumir que, no momento da criação do componente, essa interface seja conhecida pelo desenvolvedor. Desta forma, uma das exigências para a adição de um receptáculo ao componente é informar uma interface Java equivalente à sua interface IDL. Com essa interface disponível, é possível criar *proxies* para os objetos remotos utilizando o mecanismo *Invocation Handler* de Java. Os *proxies* constroem uma chamada remota equivalente a partir da assinatura da chamada recebida utilizando o mecanismo *Dynamic Invocation Interface* (DII) de CORBA, que permite construir chamadas dinâmicas de maneira similar à API reflexiva de Java. Por fim, o retorno da chamada remota efetuada é encaminhado para o objeto local que invocou o *proxy*.

A conversão de tipos Java para tipos CORBA é um trabalho relativa-

mente complexo e bastante trabalhoso. Felizmente, boa parte deste trabalho já foi realizado por implementações de CORBA em Java, restando apenas o trabalho de decomposição de tipos estruturados e complexos, como o tipo genérico *Any* de CORBA.

6.2 Contêiner Dinâmico

A implementação do sistema SCS com *binding* dinâmico serviu de base para o desenvolvimento de uma nova versão desse sistema que incorpora as ferramentas de adaptação propostas neste trabalho. Nesta seção utilizamos o *LuaCCM* como referência para avaliar a facilidade de uso e a flexibilidade da solução desenvolvida. Inicialmente, procuramos comparar a facilidade de uso das duas soluções, identificando deficiências derivadas da abordagem adotada e possíveis melhorias para trabalhos futuros.

Em seguida, na seção 6.2.2, descrevemos a implementação com a versão adaptável do sistema SCS de uma série de exemplos de uso do *LuaCCM*, que foi realizada com o intuito de identificar possíveis limitações nos mecanismos de adaptação desse sistema decorrentes da adoção da linguagem Java. Aproveitamos esses exemplos para demonstrar os principais problemas encontrados na implementação dos mecanismos de adaptação e justificar as decisões de projeto adotadas. Por fim, na seção 6.2.4, discutimos os principais desafios encontrados na implementação da solução, dando atenção especial aos que acreditamos estar relacionados à linguagem utilizada.

6.2.1 Facilidade de uso

Comparar a facilidade de uso de mecanismos de adaptação implementados em Java com outros implementados em Lua é uma tarefa difícil, uma vez que os recursos dessas linguagens são bastante diferentes e induzem a estilos de programação distintos. Ainda assim, durante a implementação de exemplos originalmente implementados com o *LuaCCM* no SCS Java adaptável, identificamos alguns pontos de comparação importantes. A API do SCS Java adaptável foi criada a partir da API do *LuaCCM* e adaptada para oferecer suporte aos diversos tipos de implementação contemplados. De maneira geral,

as operações que recebem implementações (*addFacet*, *updateFacet* e *addInterceptor*) passaram a receber como argumento um *array* de *bytes*, um nome de classe e um indicador de tipo de implementação, que indica se o *array* de *bytes* contém código fonte Lua, código fonte Java ou *bytecode* Java compilado.

O *LuaCCM* implementa duas abstrações que permitem aplicar adaptações a um sistema de forma estruturada. Do ponto de vista do usuário, essas adaptações são de extrema valia, pois permitem que adaptações relacionadas sejam agrupadas e aplicadas de forma organizada a uma aplicação, além de viabilizar o reuso dessas adaptações em diferentes componente e sistemas. No capítulo 7.1 discutimos a possibilidade de implementar abstrações semelhantes no SCS Java adaptável, através da utilização de uma linguagem declarativa que permita representar as adaptações desejadas. Um outro aspecto que tem um impacto relevante na facilidade de uso dos mecanismos de adaptação do SCS Java são as restrições impostas pelo mecanismo de verificação estática de tipos de Java. Todas as operações que permitem acessar tipos possivelmente desconhecidos, como por exemplo o mecanismo de geração de *proxies* para componentes conectados a receptáculos, tiveram de ser alteradas para permitir a parametrização dos tipo, o que acarretou em um aumento na complexidade do processo de adaptação. Esse problema não é sentido no *LuaCCM*, pois o mecanismo flexível de verificação de tipos de Lua permite utilizar tipos recém descobertos sem maiores problemas.

As demais questões identificadas são decorrentes das diferenças entre as linguagens Lua e Java. A seção 6.3 explora essas diferenças para avaliar de que forma elas influenciam o processo de adaptação.

6.2.2 Flexibilidade

A principal questão que pretendemos responder com a avaliação de flexibilidade é se existe alguma limitação no SCS Java adaptável que impeça a implementação de adaptações contempladas pelo *LuaCCM*. Acreditamos que a verificação estática de tipos de Java é um fator de alto impacto na implementação de mecanismos de adaptação dinâmica, uma vez que impõe sérias restrições à introdução de novos tipos às aplicações em tempo de execução.

Para tentar identificar possíveis limitações no SCS Java adaptável, im-

plementamos com ele todos os exemplos de uso utilizados pelo *LuaCCM*. A implementação desses exemplos também serviu para comparar as abordagens das duas soluções e identificar vantagens e desvantagens da adoção de cada uma delas. Nos capítulos a seguir descrevemos os exemplos e a sua implementação com o SCS, indicando as adaptações e pontos de atenção identificados.

Fluxo de dados

A base para os exemplos de uso do *LuaCCM* é uma aplicação simples de fluxo de pacotes de dados numerados, composta por três componentes: um produtor e dois processadores de dados. A aplicação é iniciada com o componente produtor gerando o pacote de dados de número um e o enviando aos dois processadores. A partir deste momento, inicia-se um ciclo onde os processadores recebem um pacote de dados, o consomem e enviam ao produtor a solicitação pelo próximo pacote. Ao receber uma solicitação, o componente produz o pacote de dados seguinte e o encaminha aos processadores, dando continuidade ao processo. Para não gerar pacotes de dados repetidos, o produtor responde apenas a primeira solicitação recebida com um mesmo número, ignorando as demais. Tanto os pacotes de dados quanto as solicitações são trafegados sob a forma de eventos do mesmo tipo. O componente produtor oferece uma interface de controle, através da qual é possível iniciar, encerrar e suspender a produção de dados, e todos os componentes oferecem funcionalidades de instrumentação, que permitem nomear os componentes e alterar a sua velocidade de produção / processamento.

Apesar de simples, esse exemplo é bastante relevante, pois representa uma aplicação distribuída na qual um recurso é compartilhado por clientes remotos. Através da funcionalidade de instrumentação dos componentes podemos simular situações comuns neste tipo de sistema, como variações significativas no desempenho dos clientes causadas por problemas na rede, indisponibilidade dos recursos compartilhados, entre outros. A figura 6.5 ilustra a estrutura da aplicação de fluxo de dados.

O componente produtor envia os eventos de dados através do publicador de eventos *produced*, recebe eventos de solicitação através do receptor de eventos *request* e disponibiliza as funções de controle através da faceta *controller*. Os processadores recebem os eventos do produtor através do receptor de eventos *raw* e enviam as solicitações através do emissor de eventos *done*. Todos os componentes *disponibilizam* as funcionalidades de instrumentação através da

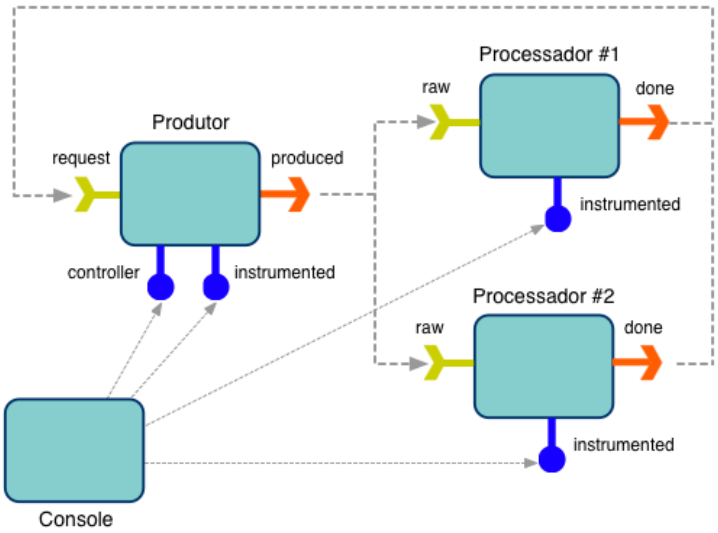

Figura 6.5: Aplicação de Fluxo de Dados

faceta instrumented.

O primeiro problema que surgiu na implementação deste exemplo, foi que o modelo do SCS não suporta comunicação baseada em eventos. Foram necessárias algumas alterações na estrutura original da aplicação para que ela fosse implementada utilizando apenas facetas e receptáculos. Os receptores de eventos foram implementados como facetas que oferecem a interface *EventConsumer*. Esta interface é composta apenas pela operação *push*, usada para receber eventos. Publicadores de eventos, que são fontes que enviam eventos a mais de um receptor simultaneamente, foram implementados como receptáculos de conexões múltiplas, enquanto emissores de eventos, que são fontes que enviam evento a um único receptor simultaneamente, foram implementados como receptáculos de conexões únicas. Todos os receptáculos de fontes de eventos aceitam apenas facetas do tipo *EventConsumer*.

Outro aspecto que precisou ser tratado para que o modelo de comunicação baseado em eventos fosse implementado corretamente é o fluxo de controle dos componentes entre as trocas de mensagens. Neste modelo, os eventos são trocados de maneira assíncrona, diferentemente das chamadas entre receptáculos e facetas do SCS. Para implementar o exemplo, utilizamos artifícios na implementação de ambos os componentes para garantir a assincronicidade da comunicação entre eles. Ao enviar um novo evento, o componente produtor executa a chamada do método *push* de cada processador em uma linha de execução (*thread*) independente, de forma que um processador não tenha que esperar o processador anterior responder para receber o seu evento. Já o processador, ao receber um novo evento, adiciona ele a uma fila de even-

tos que é processada paralelamente. Desta forma, nenhum evento é perdido, o processamento dos eventos respeita a ordem de chegada e o método *push* da faceta receptora de eventos não precisa aguardar o processamento do evento para retornar. A listagem 6.2 exibe a definição das portas dos componentes da aplicação adaptadas para o modelo SCS.

```
1  module DataFlow {
2
3          interface Instrumented {
4                  string getName();
5                  void setName(in string name);
6                  float getSpeed();
7                  void setSpeed(in float speed);
8          };
9
10         interface Controlable {
11                 void start();
12                 void stop();
13                 void pause();
14                 void _continue();
15         };
16
17         struct SerialEvent {
18                 long seq_no;
19         };
20
21         interface EventConsumer {
22                 void push(in SerialEvent event);
23         };
24 };
```

Listing 6.2: Definição dos componentes da aplicação Fluxo de Dados adaptada para o modelo SCS

Sincronização de fluxo

Conforme mencionamos anteriormente, a aplicação de fluxo de dados pode ser utilizada para simular aplicações distribuídas que compartilham recursos em situações adversas. Um problema comum em aplicações deste tipo ocorre quando um dos clientes que utiliza o recurso compartilhado reduz a sua capacidade de processamento por algum motivo e fica defasado em relação aos demais clientes, acumulando as solicitações que ele não conseguiu processar. Caso a aplicação não seja ajustada a tempo, o cliente defasado pode começar a

perder os recursos enviados devido a um possível estouro em seu acumulador. Um ajuste que poderia ser aplicado para evitar este tipo de problema seria a regulagem da produção dos recursos, de forma manter a sincronicidade entre os clientes, mesmo no caso de eventuais variações de desempenho. Para simular esta situação, utilizamos a faceta de instrumentação de um dos componentes processadores para diminuir a sua velocidade de processamento. Como o outro processador continua com a velocidade original, o ritmo de produção de eventos continua o mesmo, e a fila de eventos do componente mais lento cresce em um ritmo maior do que a sua capacidade de processamento. Como a fila deste componente é limitada, em algum momento o limite será atingido e a aplicação apresentará um erro de perda de dados.

Para solucionar este problema adaptamos a aplicação de forma que o tempo total de processamento dos eventos em cada processador seja medido e, caso esse tempo ultrapasse um limite pré-definido, o fluxo de produção de eventos seja limitado. Para tanto, definimos os seguintes mecanismos, ilustrados na figura 6.6 (a listagem 6.3 exibe a interface das portas adicionadas ao sistema):

- A Faceta *Limited* (1) foi instalada nos processadores para permitir que seja definido um limite de vazão que, caso ultrapassado, deve disparar a regulação de fluxo. Essa faceta é manipulada pelo console da aplicação.
- Um par de interceptadores (2a e 2b) foi instalado em cada um dos processadores para medir o tempo total de processamento dos eventos.
- Outro interceptador (3) foi instalado no componente produtor para introduzir pausas antes do envio dos eventos, de modo a manter uma taxa de envio de eventos definida.
- Faceta *Rateable* (4) foi instalada no produtor para permitir que componentes externos definam a taxa de envio de eventos.
- O receptáculo *Regulator* (5) foi instalado nos processadores para permitir que a implementação da faceta *Limited* dispare a regulagem do fluxo de produção de eventos no produtor.

```
1 module FlowSync {
2
3         interface Rateable {
4                 double getrate();
5                 void setrate(in double rate);
6         };
7
```

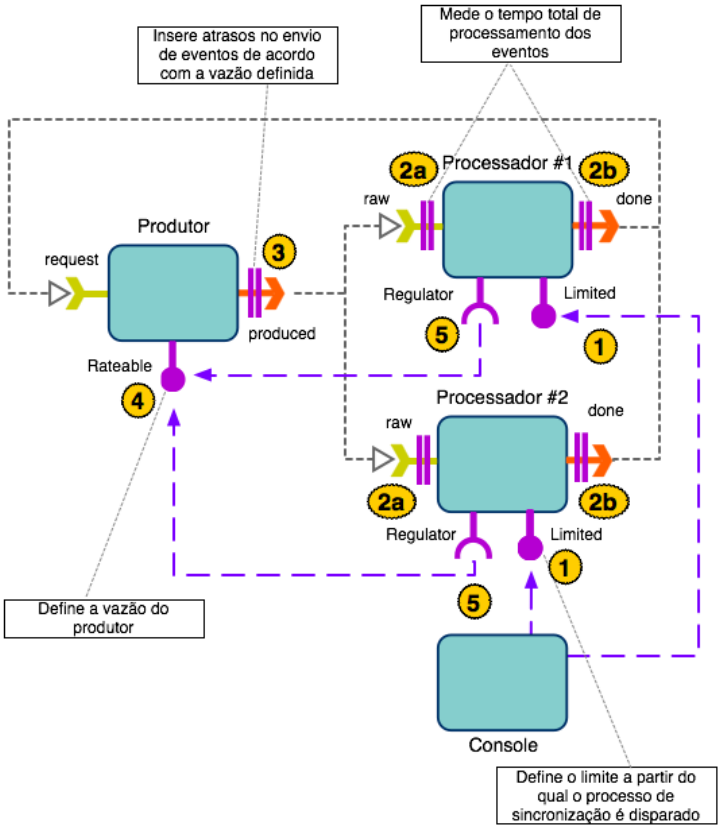

Figura 6.6: Aplicação de Fluxo de Dados após a adaptação para regulagem de fluxo

```
8           interface Limited {
9                   double getValue();
10                  void setValue(in double value);
11          };
12  };
```

Listing 6.3: Definição das portas adicionadas para a sincronização de fluxo

Em decorrência das mudanças efetuadas na aplicação devido à falta de suporte a comunicação baseada em eventos pelo SCS, a implementação das adaptações de controle de fluxo descritas acima também teve de ser alterada. Como o envio de eventos é efetuado de forma síncrona, o seu processamento teve de ser realizado em uma linha de execução a parte para que não bloqueasse o produtor. Desta forma, a única maneira de medir o tempo total de processamento de um evento é entre o seu recebimento e a solicitação do evento seguinte. Como estas ações ocorrem em portas distintas, precisaríamos de dois interceptadores para efetuar esta medição, além de alguma porta que permitisse compartilhar estado entre eles. Para simplificar o desenvolvimento, instalamos uma única classe de interceptadores nas duas

portas e compartilhamos o estado entre elas através de um atributo estático.

Depuração Distribuída

Outro experimento interessante realizado pelo *LuaCCM* sobre a aplicação de fluxo de dados foi o uso dos recursos de adaptação dinâmica para a introdução de funcionalidades que permitam a depuração remota da aplicação. O processo de depuração implementado no exemplo do *LuaCCM* consiste na introdução de pontos de parada (*break-points*) em operações específicas das portas dos componentes e na utilização de um mecanismo de inspeção para analisar o estado interno do componente quando algum ponto de parada é atingido.

O mecanismo de pontos de parada é composto por um receptáculo e um interceptador. O interceptador consulta o receptáculo para determinar se o estado de *debug* está ativado e, em caso afirmativo, interrompe a execução do método para que possa ser realizada a inspeção. O receptáculo deve ser conectado a uma faceta do tipo *Pauser* que declara apenas a operação de consulta a respeito do estado de *debug*. A inspeção de estado interno do componente é realizada através da faceta *Inspectable*, que define operações que permitem executar trechos de código Lua dentro do componente e retornam o resultado sob a forma de uma *string*. A listagem 6.4 exibe a definição das facetas *Pauser* e *Inspectable*.

```
1  module distdebug {
2          interface Inspectable {
3                  string evaluate(in string expression);
4                  string execute(in string code);
5          };
6
7          interface Pauser {
8                  boolean locked();
9          };
10 };
```

Listing 6.4: Definição das portas adicionadas para a depuração distribuída

A versão do exemplo da depuração distribuída foi bastante alterada para que pudesse ser implementado com o SCS Java dinamicamente adaptável. Neste caso, além dos problemas decorrentes da falta de suporte à comunicação baseada em eventos, adaptamos diversos recursos que foram projetados tendo em vista características da linguagem Lua e adicionamos algumas melhorias.

A figura 6.7 ilustra a estrutura da aplicação após a introdução dos mecanismos de depuração adaptados.

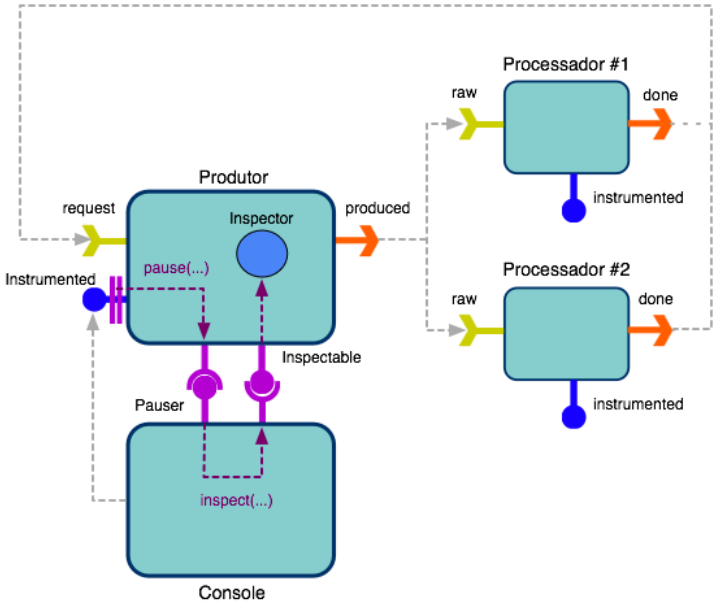

Figura 6.7: Aplicação de fluxo de dados após a adaptação de depuração distribuída

A primeira mudança se deu no interceptador do ponto de parada que, ao invés de consultar o receptáculo *Pauser* para ver se o estado de *debug* está ativado e interromper a execução, apenas chama o método pause deste receptáculo e retorna, dando continuidade à execução da operação. Como a chamada é realizada de maneira síncrona, a execução da faceta inspecionada é interrompida até que este método retorne, permitindo que o processo de inspeção seja realizado neste intervalo. O mecanismo de inspeção também foi alterado para refletir as decisões adotadas ao longo do projeto. Em vez de passar para a faceta de inspeção um trecho de código para ser executado no componente, como no caso do *LuaCCM*, optamos por passar a implementação de uma interface de inspeção, composta por um único método (*inspect*) que recebe um objeto Java com a implementação da faceta a ser inspecionada e retorna o resultado dessa inspeção sob a forma de uma *String*. A faceta *Inspectable* se encarrega de instanciar o inspetor, localizar a implementação da faceta a ser inspecionada e invocar o método de inspeção, retornando a *String* resultante. A passagem da implementação do inspetor é feita, como nos demais mecanismos de adaptação, através de três parâmetro: um *array* de *bytes* com a implementação, uma *string* com o nome da classe, para o caso de implementações Java, e um indicador do tipo de implementação (código lua, código Java ou *bytecode* Java). A listagem 6.5 exibe a definição das facetas *Pauser* e *Inspectable*, após as alterações mencionadas.

```
module distdebug {
        interface Inspectable {
                string inspect(in string facetName,
                        in string className,
                        in OctetSeq impl,
                        in string impl_type);
        };

        interface Pauser {
                void pause(in string facetName,
                        in string operation);
        };
};
```

Listing 6.5: Definição das portas adicionadas para a depuração distribuída adaptadas ao SCS Java

Replicação Passiva

O último exemplo apresentado pelo *LuaCCM* é utilizado para demonstrar a sua capacidade de efetuar adaptações no nível da definição dos componentes, aplicando assim as alterações a todas as instâncias do componente adaptado. O exemplo sugere a criação de réplicas dos componentes processadores que recebam os mesmos eventos recebidos pelos componentes originais, de forma que as réplicas possam ser utilizadas para substituir os originais caso ocorra alguma falha que impossibilite o seu funcionamento. Para implementar este exemplo um publicador de eventos foi adicionado ao componente processador, de forma que a réplica possa se conectar para receber copias dos eventos enviados a ele. Um interceptador foi então adicionado ao receptor de eventos deste componente para capturar os eventos recebidos e retransmiti-los através do novo publicador. Como essas adaptações foram aplicadas no nível da definição do componente processador em um contêiner específico, todas as instâncias deste componente criadas no contêiner serão automaticamente adaptadas. A figura 6.8 ilustra o estado da aplicação depois da adaptação e da conexão das réplicas.

No SCS adaptável não foi implementado o mecanismo de adaptações no nível da definição do componente, então as adaptações tiveram de ser aplicadas individualmente em cada um dos componentes, no caso os dois processadores. Fora isso, o exemplo foi implementado sem maiores dificuldades

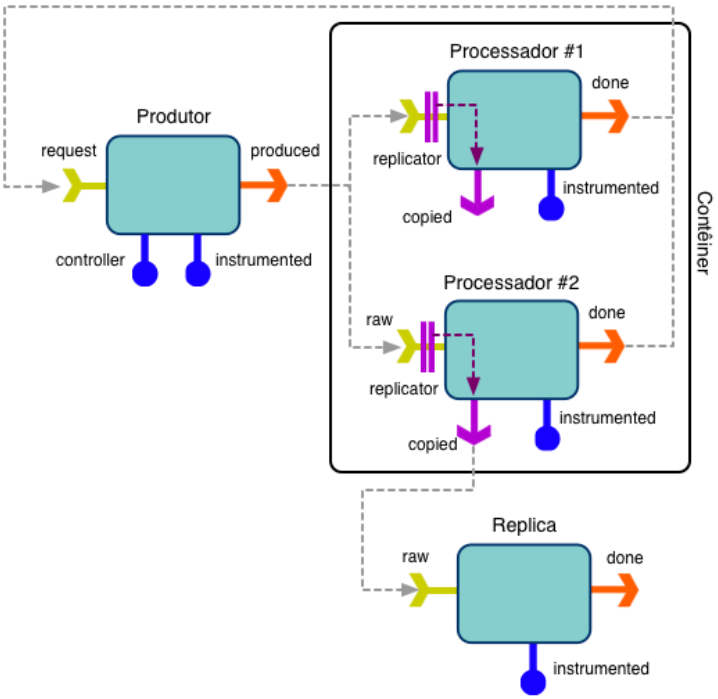

Figura 6.8: Aplicação de fluxo de dados adaptada para comportar replicas dos processadores

ou modificações merecedoras de nota.

6.2.3 Comparação

De modo geral, não nos deparamos com nenhuma limitação que impedisse a implementação de nenhum dos exemplos de adaptação testados com o *LuaCCM*. Todos eles foram implementados com sucesso no SCS Java e as poucas restrições encontradas foram decorrentes de diferenças no escopo das soluções. Analisando o código da implementações dos exemplos fica claro que cada solução atende melhor a uma situação específica, apresentando vantagens que fazem sentido neste contexto.

Um aspecto aonde o *LuaCCM* claramente leva vantagem é a agilidade da implementação das adaptações. A flexibilidade proporcionada pela linguagem Lua permite que o processo de adaptação seja simplificado e o volume de código necessário para que sejam realizadas as adaptações seja bem menor. Por exemplo, para utilizar uma nova faceta descoberta através de mecanismos de introspecção no *LuaCCM*, basta conhecer a assinatura de suas operações, enquanto no SCS é necessário obter a definição da faceta em IDL, gerar

as classes *stub* equivalentes e carrega-las. Ao analisar a implementação dos exemplos essa diferença fica clara pela quantidade de código e distribuição da lógica das adaptações: enquanto no *LuaCCM* elas são expressas em pequenos trechos de código Lua, no SCS Java adaptável foram criadas uma série de pacotes, classes e interfaces Java. O tamanho reduzido da API de adaptação do *LuaCCM* em comparação à do SCS Java adaptável reflete a simplicidade do *LuaCCM*.

Apesar da maior agilidade proporcionada, é uma expectativa comum que a ausência de um mecanismo de verificação de tipos torne o código das adaptações mais propenso a erros no *LuaCCM* do que no SCS Java. Em primeiro lugar, o mecanismo de verificação estático de tipos de Java permite uma checagem de erros bem mais abrangente do que a validação sintática de Lua, de forma que grande parte dos erros são detectados na compilação do código. A organização do código em classes e a possibilidade do uso de interfaces para garantir a assinatura das operações impõe uma maior organização no código Java, enquanto a organização do código Lua varia de acordo com a organização dos desenvolvedores que o escrevem. Entretanto, o uso de Lua facilita a definição de novas abstrações, como o uso de papeis e protocolos [23, 9].

6.2.4
Desafios de implementação

Alterar a implementação do SCS baseada em *esqueletos dinâmicos* para permitir adaptações nos componentes em tempo de execução trouxe uma série de desafios. O primeiro deles surgiu com a necessidade de trafegar as interfaces das portas de uma maneira mais eficiente. Como inicialmente as portas eram manipuladas apenas localmente, em tempo de desenvolvimento, era razoável deixar para o desenvolvedor a tarefa de instanciar as meta-classes do sistema para descrever as interfaces dessas portas. Com a introdução dos mecanismos de adaptação essa manipulação passou a ser efetuada também em tempo de execução e de forma remota. Optamos por trafegá-las sob a forma de *strings* com sua descrição em IDL, mas para isso foi necessário incorporar um *parser* desta linguagem ao sistema. Uma opção que nos pareceu adequada foi a utilização do *LuaIDL*, um *parser* de IDL desenvolvido em Lua por integrantes de nosso grupo de pesquisa. Como já havia a intenção de oferecer suporte para adaptações em Lua, a utilização de um *binding* entre Lua e Java já estava

prevista e pudemos usar o *LuaIDL* sem problemas.

A manipulação da interface das portas em tempo de execução não foi difícil, uma vez que estas interfaces já eram armazenadas em estruturas de dados que eram consultadas apenas em tempo de execução. A adição de receptáculo, por outro lado, trouxe dificuldades, pois para gerar *proxies* para um receptáculo, o *contêiner dinâmico* precisa manter uma interface Java equivalente a interface implementada por ele. Ao invés de exigir que essa interface fosse fornecida no momento da adição do receptáculo, como ocorria na versão com *binding dinâmico* inicial, optamos por permitir que o desenvolvedor informe essa interface apenas quando desejar obter uma referência de um componentes externo conectado ao receptáculo, passando a interface para o contêiner através de sua API interna para que ele gere um *proxy* para o componente desejado. Este tipo de limitação, derivado do esquema estático de verificação de tipos da linguagem Java, torna o desenvolvimento de mecanismos de adaptação dinâmica bem mais difícil quando comparado com o desenvolvimento de mecanismos similares em linguagens com tipagem dinâmica como Lua. Nos exemplos do *LuaCCM*, por exemplo, notamos que portas com interfaces recém descobertas podem ser utilizadas imediatamente sem a necessidade de um mecanismo complexo de geração de *proxies* e chamadas reflexivas como o desenvolvido neste trabalho.

Para permitir que a implementação das facetas dos componentes fosse fornecida em tempo de execução sob a forma de código Java, utilizamos a interface programática do compilador Java (*javac*) para compilar o código e as funcionalidades de manipulação de *classloader* para carregar as novas classes. No caso de implementações fornecidas como código Lua, utilizamos o *binding LuaJava* para carregar o código fornecido e efetuar chamadas equivalentes às recebidas pela porta do componente. Para oferecer suporte aos dois tipos de implementação, alteramos a classe *Facet*, responsável por receber as chamadas externas e efetuar chamadas locais equivalentes, de forma a torná-la uma classe abstrata que apenas declara o método de invocação ao invés de implementá-lo. Essa classe foi então estendida por duas novas classes, *JavaFacet* e *LuaFacet*, que implementam a lógica de invocação para implementações em Java e em Lua, respectivamente. A classe *Component*, responsável pela gerência das portas dos componentes, foi também alterada para instanciar a sub-classe da classe *Facet* adequada de acordo com o tipo de implementação de faceta recebida.

Outra alteração importante nos mecanismos de invocação dos componentes foi necessária para permitir a adição de interceptadores às suas portas.

As classes responsáveis pelas invocações de facetas e receptáculos passaram a manter uma lista ordenada de interceptadores, que são invocados seqüencialmente antes e depois da invocação principal - da implementação, no caso de facetas, ou do componente externo no caso de receptáculos. Uma dificuldade encontrada na implementação desse mecanismo foi que os interceptadores fornecidos devem implementar uma interface conhecida para que possam ser invocados pelo *esqueletos dinâmicos*. Optamos por disponibilizar a interface Java *IInterceptor*, que deve ser implementada pelos interceptadores Java. No caso de interceptadores Lua, os interceptadores devem implementar operações equivalentes às definidas nessa interface.

6.3
Linguagens de adaptação

Conforme visto no capítulo 5, durante o desenvolvimento dos mecanismos de adaptação dinâmica propostos neste trabalho, optamos por oferecer suporte a três maneiras distintas para o fornecimento do código adaptativo, de forma a poder avaliá-las posteriormente. A forma mais natural de trafegar essas implementações seria através de um *array* de bytes contendo o *bytecode* de classes Java compiladas pelo cliente. Para agilizar o processo de adaptação, adicionamos o suporte a implementações fornecidas diretamente sob a forma de código Java, compilado posteriormente pelo contêiner. Incorporamos também adaptações fornecidas sob a forma de código Lua para tentar avaliar as possíveis vantagens dessa linguagem na implementação de código adaptativo.

O primeiro aspecto que avaliamos foi o desempenho dos diversos tipos de adaptações, tanto na execução do código adaptado quanto na aplicação dessas adaptações. Embora não tenhamos efetuado medições precisas desses tempo, as diferenças eram sensíveis e as justificativas bastante óbvias. Comparando os dois tipos de adaptação Java, não existem diferenças no tempo de execução do código adaptado, uma vez que após compilado o código, o processo de carga do *bytecode* é o mesmo. Já o tempo de aplicação das adaptações é significativamente maior no caso de adaptações fornecidas como código Java, já que a compilação é um processo bastante custoso. As adaptações fornecidas sob a forma de código Lua tem um tempo de aplicação similar ao das fornecidas como *bytecode* Java, uma vez que o código não precisar ser compilado, apenas carregado pelo *binding LuaJava*. O tempo de execução do código adaptado em Lua por sua vez é bastante maior que o do código

adaptado em Java por diversos fatores. Em primeiro lugar, em nossos testes o código adaptativo escrito em Java passa por uma série de otimizações devido a utilização do compilador JIT (*just-in-time*). Apesar de existirem compiladores JIT implementados para Lua, como o seu uso ainda não é amplamente difundido optamos por não utilizá-los nos testes. Além disso, o uso do *binding LuaJava* prejudica enormemente o tempo de execução do código adaptado, pois faz uso intenso da interface nativa de Java (JNI).

Em relação à facilidade de uso, a primeira comparação importante é entre as adaptações fornecidas como código, tanto Lua quanto Java, e as fornecidas como *bytecode* Java. Obviamente o fato de não ser necessário compilar o código Java agiliza bastante o processo de adaptação nesta linguagem. Apesar disso, adaptações mais complexas podem ser difíceis de se implementar em um trecho de código e geralmente demandam algum tipo de modularização para manter um nível mínimo de organização. Neste caso, acreditamos que é mais fácil implementar a adaptação em um conjunto de classes Java, que podem ser compiladas, agrupadas e trafegadas em um único arquivo JAR (*Java Archive*). Como foi mencionado na seção 6.1.3, a geração de *proxies* em Lua é bem mais simples do que em Java e essa simplicidade se reflete na utilização dos receptáculos do componente. Enquanto implementações Lua podem solicitar um *proxy* genérico para um receptáculo e utilizá-lo normalmente, as implementações Java precisam fornecer uma interface Java equivalente para a mesma tarefa.

As vantagens de implementações Lua no que diz respeito à flexibilidade das adaptações, que teoricamente seriam as mesmas discutidas na seção 6.1.4, foram bastante limitadas pelo fato de não ser possível acessar diretamente as implementações Lua carregadas. Qualquer manipulação deve ser efetuada através da API do *binding LuaJava* e não implementamos mecanismos para disponibilizar essas funcionalidades para o usuário do sistema.