5
Avaliação Experimental

Neste capítulo, descreveremos alguns experimentos realizados para avaliação da infra-estrutura de execução. Tentaremos primeiramente demonstrar qual o custo em recursos ao utilizá-la, verificando se este se manteve baixo o suficiente para atender a nossos objetivos e validar seu uso. Descreveremos tanto testes básicos como o uso em um cenário comum, a fim de verificar também se seu uso facilita o desenvolvimento, e se acarreta perdas ao desempenho ou escalabilidade. Exibiremos ainda alguns resultados referentes ao uso de diferentes máquinas virtuais, mais especificamente a máquina Lua padrão e a máquina LuaJIT [46, 47].

Para os testes realizados, foram utilizadas sete máquinas de um *cluster*, com processadores Pentium-D (dois núcleos) de 3.4GHz e 2GB de memória RAM. O sistema operacional foi o Linux, com a distribuição CentOS 4.6 (kernel 2.6.9-67) em modo 32-bit. Foram realizadas 10 iterações de cada teste, e retirada a média e o desvio-padrão dos valores. A máquina virtual Lua utilizada foi a versão 5.1.3 e o ORB OiL em sua versão 0.4-beta. A máquina virtual LuaJIT é a da versão 1.1.4.

Não foram utilizados serviços OpenBus, que não os propostos aqui para a infra-estrutura de execução (ex: Serviço de Controle de Acesso), em nenhum teste.

5.1
Sobrecarga dos Serviços

Aqui serão apresentados alguns dados básicos sobre a memória inicial e tempo de carga dos serviços principais da arquitetura. Serão mostrados também os testes de sobrecarga causada pelo uso das facilidades de inter-

ceptação, e as diferenças encontradas ao se utilizar uma máquina virtual diferente, LuaJIT. Estes testes foram realizados em apenas uma máquina.

5.1.1 Memória Inicial

Realizamos testes para obter a memória utilizada pelo nó de execução, contêiner e contêiner com um repositório carregado, em seus estados iniciais. Mais especificamente, a memória em uso ao terminarem suas inicializações e apenas aguardarem por novas requisições, relativa ao processo Lua e medida através do sistema operacional. Não exibiremos valores de desvio-padrão pois foram sempre nulos ou insignificantes (menores que 1%).

Todos os valores são relativos ao processo inteiro, o que engloba a máquina virtual Lua, o ORB OiL e os gastos próprios de cada serviço e bibliotecas carregadas. Não foi feito nenhum tipo de otimização extra para os testes; as distribuições Lua e OiL exibem as configurações padrão. Os valores podem ser diminuídos ao se modificar a configuração de Lua e OiL, retirando por exemplo bibliotecas não utilizadas.

A título de informação, será incluída na Tabela 5.1 a quantidade de memória exigida apenas pela máquina Lua e o ORB OiL nas configurações utilizadas no teste. O ORB foi carregado através dos comandos "*require 'oil'*" e "*oil.init()*"(dentro de um contexto de execução oil.main()).

	Lua + OiL	Nó de Execução	Contêiner	Contêiner + Repositório
Memória (KB)	1872	2536	2944,8	3569,2

Tabela 5.1: Uso de Memória, com Interceptador

Notamos que estes são valores típicos de utilização, medidos pelo sistema operacional, e não a memória alocada efetivamente em uso. Efetuamos um teste para verificar estes valores, inserindo uma chamada ao coletor de lixo Lua como última tarefa de carga e observando o valor reportado pelo coletor após a coleta completa. Os resultados estão na tabela 5.2. A coleta de lixo na versão 5.1 de Lua é incremental e se dá através de um algoritmo *mark and sweep* [48].

Deve-se atentar para o fato de que todos os testes apresentados neste capítulo são afetados por esta diferença. Em nenhum outro caso forçamos a

	Lua + OiL	Nó de Execução	Contêiner	Contêiner + Repositório
Memória (KB)	1241,2	1616,6	1761,4	1786

Tabela 5.2: Uso Efetivo de Memória, com Interceptador

coleta de lixo, a não ser quando especificado, pois isto não seria representativo do uso comum para a maioria das aplicações.

5.1.2 Tempo de Carga

Aqui avaliamos o tempo de carga total demandado pela criação de um contêiner. Para obter este dado, medimos o tempo levado pelo método *startContainer()* da faceta ExecutionNode do nó de execução. Este método engloba a criação do novo processo, a carga do contêiner, o registro deste no nó de execução (última ação feita pelo contêiner em sua inicialização) e uma chamada ao método *startup()* da faceta IComponent do contêiner.

Avaliamos também o tempo de carga de um componente repositório, apesar de não ser necessário um repositório por contêiner, pois apenas um destes já seria capaz de atender a diversos contêineres. Esta parte do script de geração de tempos compreende duas chamadas, uma ao método *load()* da faceta ComponentLoader do contêiner, e outra ao método *startup* da faceta IComponent do repositório, obtida como retorno da chamada ao método *load()*. O repositório já se encontra na *cache* em disco do contêiner, e sua carga não envolve a obtenção de código através da rede.

Por fim, na tabela 5.3 mostramos também o tempo de carga do nó de execução, mas este deve ser visto como um pré-requisito da arquitetura; ou seja, espera-se que já esteja executando na máquina como um *daemon* ou serviço, de forma que não custará tempo da aplicação.

Um detalhe é que todas as chamadas ocorrem na mesma máquina, mas passam pela sobrecarga gerada pelo uso de CORBA, pois mesmo a comunicação entre contêiner e repositório que utilizam o mesmo ORB não conta com otimizações de *collocation* como explicamos na seção 4.2.3.

	Nó de Execução	Contêiner	Repositório
Tempo (s)	0,3537	1,0103	0,2126
Desvio-Padrão (s)	0,0316	0,0027	0,0157

Tabela 5.3: Tempos de Carga Inicial, com Interceptador

5.1.3

Influência do Uso de Interceptadores

Segundo nossas previsões, um dos maiores custos agregados ao se utilizar nossa arquitetura deveria ser causado pelos interceptadores. Não só há o custo causado pelo uso do interceptador CORBA (que é o gerenciador de interceptadores OpenBus) no ORB, como existe um custo causado pelos interceptadores OpenBus em si, caso algum seja instalado.

Como interceptadores OpenBus não precisam necessariamente existir para um dado contêiner, avaliaremos a sobrecarga causada pelo gerenciador de interceptadores. Esta avaliação contemplará apenas seu custo inicial, sem nenhum interceptador OpenBus carregado. Para isso, testamos um contêiner sem o interceptador CORBA instalado, em contraste com os testes das seções anteriores. Como o repositório é um componente comum, carregado em um contêiner, este também é afetado pelo uso de interceptadores e assim também foi incluído nesta seção. Os resultados mostrados serão apenas com coleta de lixo forçada para que possamos realizar uma comparação concreta, e encontram-se na tabela 5.4.

	Contêiner	Repositório
Tempo de Carga (s)	1,0110	0,2070
Desvio-Padrão (s)	0,0010	0,0012
Uso de Memória (KB)	1663,3	1690,3

Tabela 5.4: Sobrecarga sem o Uso do Interceptador CORBA, Com Coleta de Lixo Forçada

Podemos notar que inicialmente há uma pequena diferença em relação à memória, causada pelos interceptadores carregados. Neste caso a maior diferença foi de 5,57%, no uso de memória do contêiner, com os outros valores situando-se em patamares similares. Já em relação ao tempo de carga, estes em geral também são ligeiramente diferentes, com o uso das facilidades de interceptação incorrendo em uma pequena piora nos resultados. A maior diferença ocorreu na carga do repositório: 1,92%.

Avaliaremos também esta influência sobre o uso de componentes de aplicação, em uma situação mais real. Estes testes serão apresentados na seção 5.3.

5.2
Diferenças Entre Máquinas Virtuais Lua

Devido ao suporte oferecido pela infra-estrutura de execução a diferentes máquinas virtuais, incluímos também testes com máquinas virtuais diferentes. Foram realizados testes com a máquina Lua padrão e com a máquina LuaJIT padrão.

5.2.1
Memória, Tempo de Carga e Interceptadores

Cada teste realizado anteriormente foi refeito com a máquina LuaJIT. Com esta, cada trecho de código é compilado apenas na primeira vez e, caso venha a ser chamado novamente, os mesmos binários são reutilizados. Isto traz um ganho de desempenho para tarefas repetitivas, em troca de um aumento na utilização de memória. Nestes testes de carga inicial não esperávamos e não encontramos uma grande diferença, mas mostraremos outros exemplos na seção 5.3 onde a diferença é mais significativa. Os resultados encontram-se nas tabelas 5.5, 5.6 e 5.7.

	Lua + OiL	Nó de Execução	Contêiner	Contêiner + Repositório
Memória (KB)	2688	4054,8	4956,8	5387,2
Memória Efetiva (KB)	1452,7	2070,6	2300,2	2414,6

Tabela 5.5: Uso de Memória, com Interceptador, LuaJIT

	Nó de Execução	Contêiner	Repositório
Tempo (s)	0,2811	1,0117	0,1347
Desvio-Padrão (s)	0,0142	0,0014	0,0014

Tabela 5.6: Tempos de Carga Inicial, sem Interceptador, LuaJIT

	Contêiner	Repositório
Tempo de Carga (s)	1,0101	0,1365
Desvio-Padrão (s)	0,0004	0,0009
Uso de Memória (KB)	2194,5	2309,5

Tabela 5.7: Sobrecarga sem o Uso do Interceptador CORBA, com Coleta de Lixo Forçada, LuaJIT

Com o uso de interceptadores, o tempo de carga inicial do Nó de Execução caiu 20,53%, enquanto que o do contêiner se manteve praticamente o mesmo com um ligeiro aumento de 0,14%. Já o tempo do repositório teve uma queda de 36,64%. O uso de memória efetiva inicial (com coleta de lixo forçada) foi 28,08% maior no nó, 30,59% maior no contêiner e 35,20% maior no conjunto do contêiner com repositório carregado.

Sem o uso de interceptadores, o tempo de carga inicial do contêiner se manteve praticamente o mesmo com uma ligeira queda de 0,09%. Já o tempo de carga do repositório teve uma queda de 34,06%. O uso de memória efetiva inicial (com coleta de lixo forçada) foi 31,94% maior no contêiner e 36,63% maior no conjunto do contêiner com repositório carregado.

A partir destes resultados, podemos concluir que o uso da máquina LuaJIT levou aos resultados esperados, que eram a melhoria dos tempos de carga em troca de um aumento na utilização de memória. Os resultados do contêiner sobre o tempo de carga foram os únicos que se mostraram diferentes, pois praticamente não houve modificações. Podemos atribuir estes resultados ao fato de que este serviço é o único que conta com chamadas remotas em sua inicialização, que têm um peso maior no tempo de carga do que a melhoria que as características do LuaJIT proporcionam. Isso faz com que o tempo de carga do contêiner seja governado pelo tempo dessas chamadas.

5.3 Exemplo: Sistema de Eventos

Esta seção apresentará um exemplo simples de uso da arquitetura. Um dos cenários de uso que consideramos bastante atraente é a construção de ferramentas de apoio ao teste de *software* distribuído. Por isso, escolhemos demonstrar aqui uma aplicação de testes, que será utilizada para testar uma implementação do serviço de eventos de CORBA feita em Lua [49] em ter-

mos de desempenho e escalabilidade. Posteriormente, na Seção 5.4, mostraremos algumas modificações que podem ser feitas na aplicação ao utilizar a infra-estrutura de execução criada, facilitando os processos de implantação e execução remota da aplicação.

O sistema de eventos se caracteriza pela comunicação através de canais, nos quais eventos são publicados por produtores, e em seguida enviados pelo próprio canal a consumidores. Este tipo de funcionamento é chamado de modelo *push*, pois o canal envia os eventos a consumidores cadastrados sem sua requisição explícita. Já no modelo *pull*, o canal armazena os eventos, que são posteriormente recuperados por consumidores quando estes considerarem apropriado. As implementações que serão avaliadas aqui fazem uso apenas do modelo *push*.

Compararemos uma implementação feita pelo nosso grupo do Serviço de Eventos CORBA, em relação a uma implementação que se aproveita do paradigma de orientação a componentes e que usufrui das facilidades da infra-estrutura de execução. Devido a estas facilidades, alguns pontos poderiam ser bastante facilitados. Um exemplo seria a conexão entre componentes, pois o próprio modelo SCS trata conexões através do uso de receptáculos. No entanto, devido às mudanças nas interfaces a compatibilidade com o Serviço de Eventos CORBA padrão seria perdida. Para não haver prejuízo em nenhum dos casos, nossa implementação componentizada permite a comunicação tanto com outras instâncias componentizadas, como com instâncias do Serviço de Eventos CORBA padrão.

Como consumidores e produtores não mantêm nenhum tipo de contato direto, já que ambos comunicam-se apenas com canais, cabe ao canal componentizado manter a compatibilidade tanto com versões padrão como componentizadas. Consumidores e produtores comunicam-se apenas com canais do mesmo tipo, ou seja, um consumidor OpenBus se comunica com um canal OpenBus, mas não com um canal padrão. Canais padrão são compatíveis apenas com consumidores e produtores padrão, mas um canal OpenBus é capaz de comunicar-se com todos.

Componentes OpenBus conectam-se através de receptáculos, enquanto que o código padrão utiliza um conjunto formado por duas entidades administradoras (uma para consumidores e uma para produtores) e objetos *proxies*, também para ambos os casos. Desta forma, este processo é bastante simplificado quando se utiliza a arquitetura OpenBus. Para permitir a compatibilidade, um canal OpenBus mantém as conexões de duas formas diferentes.

Quando envia os eventos a consumidores, primeiro percorre a lista dos objetos padrão, depois a do receptáculo OpenBus. No entanto, todas as chamadas são feitas de forma assíncrona, e assim o prejuízo para os componentes OpenBus é minimizado. Ainda assim os testes não são totalmente justos, com este pequeno prejuízo para a versão componentizada. A versão padrão testará apenas comunicação padrão, enquanto que a versão componentizada testará apenas comunicação entre componentes OpenBus, tendo a fila de objetos padrão vazia.

Para verificarmos a diferença de desempenho, comparamos a quantidade de eventos que passam pelo canal, contabilizando o tempo de manipulação total entre a chegada do primeiro evento e a saída do último. Não são contabilizados os tempos de chamadas remotas referentes a publicação ou consumo dos eventos. Também não foram utilizados os serviços OpenBus que configuram um barramento(controle de acesso e registro), pois este não é conhecido pelo código padrão. É utilizado o serviço de nomes CORBA em ambos os casos.

Já para avaliarmos a utilização de memória no canal, primeiro marcamos a utilização inicial do processo. A partir de então, é verificada a utilização de memória a cada segundo, obtendo-se ao fim um valor razoavelmente descritivo da variação total de memória do processo, bem como os valores mínimo e máximo.

Foi criado um script parametrizado para variar a quantidade de consumidores, produtores e eventos, máquinas virtuais, quantidade de contêineres e nós a serem utilizados (um para o canal, uma lista para consumidores e outra lista para produtores). O script se encarrega de distribuir o número de cada componente entre o número de nós providos. Na máquina escolhida para o canal, quando da execução da versão componentizada, é criado um contêiner e carregado um repositório, que será utilizado por todos os contêineres e no qual são instalados todos os componentes necessários. Em seguida, é criado um outro contêiner no mesmo nó, onde é carregado e inicializado o canal. A partir de então, cada consumidor é carregado em um contêiner próprio em seu nó escolhido, e por último o mesmo ocorre para os produtores, que ao serem inicializados iniciam o envio de eventos.

A partir desta configuração, os testes consistiram de todas as combinações de 3, 6, 9 ou 12 consumidores e produtores, com sempre um canal e, no caso da versão componentizada, um repositório adicional. Cada produtor publica 5000 eventos seqüencialmente, que são repassados pelo canal a todos os consumidores. Os consumidores apenas registram que receberam o evento para confirmação posterior de que todos foram recebidos. As sete máquinas utiliza-

das foram repartidas em três nós para produtores, três para consumidores e um para o canal e repositório. Nos casos de 3 elementos, é então instanciado apenas um por nó. Nos casos de 6 elementos, dois por nó e assim sucessivamente.

Notamos que, para manter a mesma metodologia de obtenção de resultados tanto para a versão padrão quanto para a componentizada, utilizamos *scripts* do sistema operacional nestes testes. Estas medições servirão para validar o uso da arquitetura, avaliando se a perda de desempenho e o aumento no uso da memória são pequenos o suficiente para não se tornarem um empecilho maior que os benefícios para a maioria das aplicações.

A partir desses resultados, avaliaremos na Seção 5.4 modificações na metodologia para o aproveitamento de todas as facilidades da arquitetura, agregando benefícios em termos de facilidade de criação e modificação, e principalmente flexibilidade para a execução da aplicação de testes.

Os resultados obtidos no *cluster* serão apresentados a seguir. Por questões de espaço, serão inclusos apenas as médias finais de alguns resultados mais significativos, e não todas as iterações. A tabela 5.8 refere-se à versão padrão do Serviço de Eventos, a tabela 5.9 refere-se à versão componentizada com o sistema de interceptação funcional e a tabela 5.10 refere-se à versão componentizada sem qualquer tipo de interceptação habilitada. Lembramos que estes dados referem-se ao uso de memória típico e não ao efetivo, pois não foi forçada a coleta de lixo.

Consumidores X Produtores	Média (eventos/s)	Desvio-Padrão (eventos/s)	Memória Inicial (KB)	Variação de Memória (KB)
Lua				
3x3	1622,14	28,26	3857,6	6896,8
6x6	2029,54	91,60	3860	15117,6
9x9	2337,36	65,62	3859,2	23612
12x12	2515,12	109,72	3860	31293,6
LuaJIT				
3x3	2276,79	102	5192	9167,2
6x6	2926,39	136,20	5192	19220
9x9	3134,19	119,04	5190,4	29884,8
12x12	3298,84	132,32	5192	39431,2

Tabela 5.8: Canal - Padrão

Nossa previsão inicial foi de que os resultados se mostrassem piores na arquitetura OpenBus, devido à sobrecarga adicional. Além disso, prevíamos

Consumidores X Produtores	Média (eventos/s)	Desvio-Padrão (eventos/s)	Memória Inicial (KB)	Variação de Memória (KB)
Lua				
3x3	1042,95	16,68	2944	9061,6
6x6	1270,56	54,45	2944,8	17064
9x9	1480,09	23,65	2944	25232
12x12	1580,17	8,55	2944,8	32464
LuaJIT				
3x3	1504,33	21,22	4935,2	10734,4
6x6	1874,90	123,61	4908	20201,6
9x9	2020,91	82,20	4814,4	30445,6
12x12	2197,24	41,56	4876	38836

Tabela 5.9: Canal - Componentizado - com Interceptador

Consumidores X Produtores	Média (eventos/s)	Desvio-Padrão (eventos/s)	Memória Inicial (KB)	Variação de Memória (KB)
Lua				
3x3	1558,01	13,09	3220	8447,2
6x6	1949,82	92,47	3190,4	16739,2
9x9	2206,70	63,34	3219,2	25035,2
12x12	2389,61	73,34	3244,8	32744
LuaJIT				
3x3	2225,69	41,24	4759,2	10348,8
6x6	2983,72	109,54	4701,6	20078,4
9x9	3098,12	198,73	4701,6	30187,2
12x12	3559,37	42,92	4680,8	39337,6

Tabela 5.10: Canal - Componentizado - sem Interceptador

que o maior custo viesse da utilização de interceptadores mas esperávamos também que a arquitetura por si só impusesse alguma perda considerável de desempenho, principalmente neste exemplo com alto processamento de mensagens. Esperávamos, também, algum aumento na utilização de memória, mas nada muito alto.

Estas previsões se mostraram corretas, na maior parte. A maior exceção, em relação ao número de eventos por segundo, foi o custo advindo da interceptação que se mostrou bastante alto, variando entre 32,93% e 34,84% de perda de desempenho na máquina Lua (com resultados similares para a máquina LuaJIT) em relação à versão componentizada sem o uso de intercep-

tadores. Por outro lado, tivemos uma outra fuga do esperado que se mostrou bastante positiva, que foi a baixa sobrecarga causada pela arquitetura sem interceptação. Em relação à implementação padrão não-componentizada, esta perdeu entre 3,93% e 5,59% de desempenho com a máquina Lua. No caso da máquina LuaJIT, as variações foram de 2,24% negativos a 7,90% positivos, ou seja, houve até mesmo ganho de desempenho. Em geral, quanto mais produtores e consumidores, pior parece ser o desempenho (exceto no caso descrito onde houve ganho, que mostra melhor desempenho com mais elementos participantes), apesar da influência destes números não ser muito forte. A arquitetura pode ser utilizada sem suporte à interceptação se desejado, pois apesar de haver perda em funcionalidades, estas talvez não sejam úteis à aplicação em específico. Um trabalho futuro interessante seria a realização de estudos mais precisos para detectar a razão da forte queda de desempenho com interceptadores.

Em relação ao uso de memória, podemos afirmar que este é bastante dependente do algoritmo de coleta de lixo de Lua. O fato da linguagem contar com coleta incremental faz com que os valores reservados de memória (soma da memória inicial com a variação) aumentem bastante, ainda que isto não esteja de acordo com o uso efetivo. Comparando resultados como nas tabelas 5.1 e 5.2, pode-se observar a reserva do dobro de memória do que estava em uso, em certos casos. Isto leva também a alguns resultados inesperados, como o canal sem interceptação exigir mais memória que o canal com interceptação, que ocorreu no caso do uso da máquina Lua padrão nas tabelas 5.9 e 5.10. Como mostramos nas seções iniciais deste capítulo, a coleta de lixo faz com que o uso efetivo de memória diminua, e a medição através do coletor exibe esses valores efetivos, coerentes com a utilização baixa esperada. No entanto, a coleta de lixo incremental de Lua faz com que a quantidade de memória reservada ao processo tenda a não diminuir. Como estas medições foram realizadas através de chamadas do sistema operacional, os valores exibidos são altos. Apesar dos valores sem coleta forçada não serem condizentes com o uso real, decidimos manter esses resultados pelo fato de que, dependendo da plataforma e configurações utilizadas, podem ser os obtidos na prática. Nesta situação, podemos observar valores similares de uso de memória entre todas as versões do Sistema de Eventos.

A máquina LuaJIT mostrou muito melhor sua influência neste exemplo. Em relação à quantidade de eventos por segundo, em comparação à máquina Lua, a implementação padrão obteve entre 31,16% e 44,19% de melhoria. A implementação componentizada sem interceptadores obteve entre 40,40% e

53,02% de melhoria, e a implementação componentizada com interceptadores obteve entre 36,54% e 47,56% de melhoria. No geral, as implementações componentizadas se beneficiaram um pouco mais. Em relação à quantidade de memória total alocada (soma da memória inicial com a variação), a implementação padrão passou a utilizar entre 26,94% e 33,52% mais memória, enquanto que a implementação componentizada sem interceptadores passou a utilizar entre 22,31% e 29,49% a mais e, a implementação componentizada com interceptadores, entre 23,45% e 30,52% a mais. Novamente as implementações componentizadas tiveram ligeira vantagem, mas estas medições de memória também não são representativa do uso efetivo, e sim da quantidade de memória alocada pelo processo Lua.

De acordo com esses resultados, consideramos o uso da arquitetura válido para aplicações distribuídas em geral, devido à baixa sobrecarga. Em especial, a perda relativa de desempenho computacional tende a ser atenuada pelo uso e condições da rede, que podem impor atrasos consideráveis.

5.4
Modificações e Melhorias Relacionadas ao Uso da Infra-Estrutura

Os testes da seção anterior foram executados utilizando a mesma metodologia tanto para a versão do Sistema de Eventos CORBA padrão, como para a versão componentizada. Como mencionamos, isso foi feito para que os resultados de desempenho não fossem comprometidos, e pudéssemos validar o uso da infra-estrutura de execução ao verificar que esta não incorre em uma alta sobrecarga. No entanto, podemos agora demonstrar como a aplicação pode ser modificada para aproveitar melhor o modelo SCS e a infra-estrutura de execução, facilitando os processos de implantação e execução da aplicação.

Para começar, em cenários de testes frequentes, todo o processo de implantação e execução dos componentes pode ser facilmente empacotado num componente "instanciador", e os parâmetros transformados em um arquivo de configuração ou mantidos da mesma forma. Assumindo que códigos a serem executados ainda não se encontram nas máquinas que os executarão, podemos ver abaixo um exemplo de como seria a função de instanciação de um componente instanciador:

Listing 5.1: Exemplo de Componente Instanciador

```
1  function launchApplication(nodeProxy, containerName,
       containerProps, components, args)
2    local handles = {}
3    local errors = {}
4    -- lançamento do contêiner no nó de execução especificado
         pelo proxy recebido
5    local ctr = nodeProxy:startContainer(containerName,
         containerProps)
6    ctr:startup()
7    -- obtenção da faceta loader
8    local loader = ctr:getFacetByName("ComponentLoader"):_narrow
         ()
9    -- carga dos componentes
10   local status = false
11   local err = 0
12   for index, id in ipairs(components) do
13     status, handle = oil.pcall(loader.load, loader, id, args[
           index] or {})
14     if not status then
15       -- erro ao carregar o componente atual
16     handles[index] = {cmp = {}, id = id, instance_id = 0}
17     errors[index] = handle
18     else
19       -- componente atual carregado com êxito
20       handles[index] = handle
21       errors[index] = ""
22     end
23   end
24   -- retorna uma referência para o IComponent do contêiner
         criado, a tabela de handles e a tabela de erros
25   return ctr, handles, errors
26 end
```

Este tipo de funcionalidade poderá ser futuramente inserido em um dos componentes da Infra-Estrutura de Execução, como o Nó de Execução ou o Contêiner de Componentes. Devemos notar que, por ser um componente SCS carregável em contêiner, não precisamos sequer ter acesso direto à máquina que executará o instanciador, nem obviamente às máquinas que executarão de fato os testes. Vejamos um exemplo do código necessário para utilizar a função *launchApplication*, assumindo que tenhamos uma referência para um componente instanciador chamada "instanciador":

Listing 5.2: Exemplo de Uso da launchApplication

```
1  EN = orb:newproxy(assert(oil.readfrom("execution_node.ior")))
2  EN:startup()
3  local enFacet = EN:getFacetByName("ExecutionNode"):_narrow()
4
5  -- Iniciando aplicação
6  -- Definição das propriedades do contêiner
7  local containerName = "ComponentContainer"
8  local containerProperties = { { name = "language" , value = "
       lua", read_only = true },
9    { name = "machine", value = "lua", read_only = true } }
10
11 -- Definição das propriedades dos componentes a serem
       carregados
12 local componentId1 = { name = "MyComponent1", major_version =
       1, minor_version = 0,
13   patch_version = 0, platform_spec = "" }
14 local componentId2 = { name = "MyComponent2", major_version =
       1, minor_version = 0,
15   patch_version = 0, platform_spec = "" }
16 local ids = { componentId1, componentId2 }
17 -- Para fins de exemplo, vamos assumir que o primeiro
       componente espere uma string como argumento
18 local args = { {"a string"}, {} }
19
20 -- Chamada a launchApplication
21 local container, handles, errors = instanciador:
       launchApplication(enFacet, containerName,
22   containerProperties, ids, args)
23
24 -- Checando por erros
25 for index, err in ipairs(errors) do
26   if err ~= "" then
27     print("Erro ao carregar o componente " .. index .. "! Erro:
           " .. err)
28     return
29   end
30 end
```

Em contra-partida, sem o uso de uma infra-estrutura de apoio, alguma forma de acesso remoto à máquina (e possivelmente configurações de contas de usuário e seus direitos) é necessária. Veremos a seguir o código que teve de ser desenvolvido para a execução dos testes da Seção 5.3, para que

mantivéssemos a mesma metodologia de lançamento dos testes em relação à implementação não-componentizada. Alguns dos maiores desafios encontrados nessa implementação, baseada apenas em *scripts* do sistema operacional, foram:

- Necessidade de uso de ferramentas externas como o ssh, rename e ps, que podem ser diferentes em cada plataforma
- Como garantir a finalização dos códigos remotos após o término da execução, em cada máquina
- Como finalizar o *script* de monitoramento dos códigos remotos
- Impossibilidade de evitar código específico de uma plataforma
- Como coletar os dados produzidos
- Configurações dos ambientes remotos, pois pode ser necessário refazer certas configurações a cada ssh
- Grande quantidade de parâmetros do *script* principal para que não seja necessário modificá-lo a cada teste

O código pode ser conferido no Apêndice C.1, por questões de espaço.

Esses exemplos mostram como a quantidade de código e trabalho necessários são muito maiores sem o uso de uma infra-estrutura de apoio à execução, além de resultar em um código mais complexo e sem abstrações que ajudem a entender melhor o que está sendo feito. Mas as vantagens não limitam-se à diminuição do trabalho bruto de codificação. Podemos também representar os próprios testes como um componente. Esse componente, que chamaremos de "EventServiceTester", pode ser desenvolvido de diversas formas diferentes. Exemplificaremos com a seguinte interface, que prioriza a simplicidade para o usuário do componente:

Listing 5.3: Interface do Componente EventServiceTester

```
1 module testing {
2   exception ParameterNotDefined{ string name };
3
4   interface SimpleTest {
5     boolean run() raises ( ParameterNotDefined );
6     sequence<string> neededParameters();
7   };
8 };
```

Para um teste diferente da mesma aplicação, basta que sejam trocados certos parâmetros de funcionamento do componente. Esses parâmetros podem ser a linguagem, o Nó de Execução e o nome da máquina virtual a serem utilizados, parâmetros de inicialização que a implementação do Serviço de Eventos espere, entre outros. Para modificá-los, pode-se utilizar uma implementação de faceta fornecida pela infra-estrutura para isso, chamada "ComponentProperties". Pode-se executar diversos testes em sequência, apenas mudando os parâmetros, ou em paralelo, possivelmente utilizando-se várias instâncias de "EventServiceTester". É fácil perceber também que, caso admita-se um conjunto de parâmetros único entre diferentes aplicações, pode-se generalizar o componente "EventServiceTester"para um componente chamado "TestRunner", que seja utilizado entre várias aplicações, e que pode também servir de base para o desenvolvimento de um arcabouço de suporte à execução de testes distribuídos.

Em nosso exemplo, o método *run* da faceta SimpleTest se encarregará de instalar os componentes necessários ao teste do Serviço de Eventos, utilizar um instanciador para instanciá-los, realizar as conexões e colocá-los para funcionar. Para isso, precisará de três parâmetros (ou propriedades): um *proxy* para um instanciador e três *proxies* para nós de execução, que serão divididos entre consumidores, produtores e o canal. O método retornará apenas um *boolean*, indicando se o teste foi bem sucedido ou não. Ao invés de utilizarmos estruturas complexas para obter todos os resultados desejados, optaremos por outros métodos que também se aproveitem do modelo de componentes e da infra-estrutura de execução. Esses serão descritos mais à frente, ainda nesta seção.

Vejamos um exemplo da utilização do EventServiceTester, assumindo que temos acesso a um Componente Instanciador cujo *proxy* chama-se "instanciador", e que temos acesso a um vetor de *proxies* para facetas "ExecutionNode", representando diferentes nós da rede (que poderia ser obtido através de um Serviço de Registro, por exemplo), chamado nodes:

Listing 5.4: Exemplo de Uso do Componente EventServiceTester

```
1 -- Definição das propriedades do contêiner
2 local containerName = "ComponentContainer"
3 local containerProperties = { { name = "language" , value = "
    lua", read_only = true },
4   { name = "machine", value = "lua", read_only = true } }
5
```

```
6  -- Definição das propriedades dos componentes a serem
       carregados
7  local testerId = { name = "EventServiceTester", major_version =
       1, minor_version = 0,
8    patch_version = 0, platform_spec = "" }
9  local args = { {} }
10
11 -- Chamada a launchApplication
12 local container, testerHandles, errors = instanciador:
       launchApplication(enFacet, containerName,
13   containerProperties, {testerId}, args)
14 local tester = testerHandles[1].cmp
15 local props = tester:getFacetByName("ComponentProperties"):
       _narrow()
16 props:setProperty( { name = "instanciador" , value = orb:
       tostring(instanciador), read_only = true } )
17 props:setProperty( { name = "node1" , value = orb:tostring(
       nodes[1]), read_only = true } )
18 props:setProperty( { name = "node2" , value = orb:tostring(
       nodes[2]), read_only = true } )
19 props:setProperty( { name = "node3" , value = orb:tostring(
       nodes[3]), read_only = true } )
20 local simpleTest = tester:getFacetByName("SimpleTest"):_narrow
       ()
21 simpleTest:run()
```

Outros benefícios podem ser vistos ao se modificar os scripts para a obtenção e agrupamento de resultados. O código para a obtenção de memória, por exemplo, pode ser transformado em um pequeno componente a ser executado no mesmo nó do canal. Possivelmente no mesmo contêiner também, dependendo da forma utilizada para se medir a memória (o coletor de lixo Lua poderia ser utilizado no lugar de chamadas do sistema operacional, apesar disto proporcionar resultados potencialmente diferentes). Este componente pode também realizar o trabalho de manipulação e formatação dos dados obtidos. Além da óbvia melhoria em termos de organização e separação do código, todos os resultados podem ser disponibilizados remotamente através de uma faceta, facilitando a inspeção dos resultados parciais ou finais. O Componente Instanciador pode fornecer uma interface de inspeção, e receber os dados já prontos para serem exibidos, com apenas uma chamada. Só o fato de se abstrair os detalhes de chamadas específicas do sistema operacional retira a preocupação com várias especificidades. Alguns poucos exemplos são:

- Diferentes interpretadores de comando (ex: *statements* como "*for*"com diferentes sintaxes);
- Diferentes utilitários com mesmo propósito em diferentes sistemas operacionais (ex: obtenção de dados de *hardware* no Linux e Windows);
- Versões diferentes de utilitários em um mesmo sistema operacional, possivelmente levando a parâmetros esperados diferentes (ex: comandos gerais no Linux, como *rename* e *ps*);
- Diferenças comuns entre sistemas operacionais como a representação de caminhos e *pipes*.
- Necessidade de diversos *scripts* diferentes para tratar todas as possibilidades que se aplicarem.

Para as medições de quantidade de eventos por segundo, ao invés de inserir o código no componente, pode ser utilizado um interceptador, que tenha um receptáculo para componentes interessados nos resultados. Ao término da medição, o próprio interceptador pode tratar e enviar os resultados automaticamente. Como componentes interessados provavelmente estarão localizados em outras máquinas, uma boa quantidade de código será poupada somente pelo uso de receptáculos.

O Apêndice C.2 mostra o código utilizado para a obtenção e formatação dos resultados da Seção 5.3. Além de complexo e difícil de entender, principalmente para quem não tem conhecimento profundo de scripts UNIX, é totalmente dependente de plataforma e de versões de ferramentas externas.

Assumindo que clientes interessados nos resultados de eventos por segundo implementem uma faceta chamada DataCollector, que contenha um método *sendData*, vejamos agora o código necessário para a criação de um interceptador que calcule a quantidade de eventos por segundo e faça o envio destes resultados aos clientes conectados:

Listing 5.5: Exemplo de Interceptador Para Obtenção e Envio de Resultados

```
1 local oil  = require "oil"
2 local oo   = require "loop.base"
3 local scs  = require "scs.core.base"
4 local orb  = oil.init()
5 orb:loadidlfile("eps.idl")
6
```

```
7  -- Definição de funções internas
8
9  local function sendData(self)
10   for _, dc in self.DataCollector:__all() do
11     dc:sendData(self.eps)
12   end
13 end
14
15 -- Definição de funções de interceptação
16
17 local function receiverequest(self, request)
18   if self.numEvents == 0 then
19     self.startTime = os.time()
20   end
21 end
22
23 local function sendreply(self, reply)
24   self.numEvents = self.numEvents + 1
25   if self.numEvents == self.totalEvents then
26     self.endTime = os.time()
27     self.eps = self.totalEvents / (self.endTime - self.
         startTime)
28     oil.newthread(sendData, self)
29   end
30 end
31
32 -- Descrições de Facetas e Receptáculos
33 local facetDescs = {}
34 facetDescs.IComponent = {name = "IComponent", interface_name =
     "IDL:scs/core/IComponent:1.0", class = scs.Component}
35 facetDescs.IReceptacles = {name = "IReceptacles",
     interface_name = "IDL:scs/core/IReceptacles:1.0", class =
     scs.Receptacles}
36 local receptDescs = {}
37 receptDescs.DataCollector = {name = "DataCollector",
     interface_name = "IDL:event/DataCollector:1.0",
     is_multiplex = true, type = "ListReceptacle"}
38 -- ComponentId
39 local componentId = {name = "OpenbusClientInterceptor",
     major_version = 1, minor_version = 0, patch_version = 0,
     platform_spec = ""}
40
41 -- Fábrica para que possa ser carregado em um Contêiner de
     Componentes
42 local Factory = oo.class{ facetDescs = facetDescs, receptDescs
```

```
    = receptDescs, componentId = componentId }
function Factory:create(args)
  local instance = scs.newComponent(self.facetDescs, self.
      receptDescs, self.componentId)
  instance.numEvents = 0
  instance.totalEvents = tonumber(args[1])
  instance.receiverequest = receiverequest
  instance.sendreply = sendreply
  return instance.IComponent
end
return Factory
```

O interceptador demonstrado seria instalado como um interceptador servidor no contêiner que estivesse hospedando o canal. Opcionalmente, pode-se identificar a chamada nas funções receiverequest e sendreply, caso espere-se que o contêiner receba chamadas de outros tipos que não eventos. Essa funcionalidade é provida pelo ORB OiL.

5.5 Considerações Finais

Através das diferenças citadas na seção anterior, a quantidade de linhas de código escritas relativas à aplicação de testes torna-se menor e o processo todo torna-se menos trabalhoso. Além disso, torna-se mais fácil automatizar tarefas tediosas como a obtenção e formatação de resultados de testes. Acreditamos também que as abstrações criadas pela orientação a componentes torne o código mais fácil de entender. Notamos também que deve ser levada em conta ainda a possibilidade de uso de diferentes linguagens de programação para cada componente, o que facilita a distribuição de tarefas entre times diferentes de desenvolvimento e permite o uso da linguagem mais adequada para cada tarefa. Sem o apoio da infra-estrutura, seriam necessárias ainda mais variações de *scripts*, ou versões mais complexas dos mesmos.

No entanto, ainda que o uso do modelo de componentes SCS e da infra-estrutura de execução apresentada simplifique tarefas como a execução de testes distribuídos, em certos momentos no desenvolvimento dos testes sentimos a falta de outros tipos de serviços. Um exemplo seria o acompanhamento do uso dos recursos do sistema. Apesar dos exemplos incluirem dados desse tipo, con-

sideramos que muitas aplicações, principalmente arcabouços de teste, têm interesse nessas informações. Dessa forma, seria útil que a própria infra-estrutura as fornecesse. O Nó de Execução seria um ótimo candidato para tal funcionalidade.