5
Avaliação

Decidimos avaliar a arquitetura de componentes para o OiL proposta neste trabalho em duas dimensões diferentes. Na primeira, demonstramos a capacidade de configuração do *middleware* com alguns exemplos utilizando trocas de componentes. A segunda dimensão é a análise de desempenho do OiL, comparando-o à versão antiga, que não era baseada em componentes, com clientes iguais.

5.1
Exemplos de uso

Mediante a instanciação de componentes específicos, o OiL pode ser configurado de acordo com a necessidade da aplicação final. A seguir apresentamos alguns exemplos de configurações possíveis. Os experimentos incluem: a seleção do protocolo de comunicação utilizado pelo *middleware*; a seleção do tipo de transporte usado na comunicação entre o cliente e o servidor; a escolha do uso ou não de um escalonador no *middleware*; um servidor que responde dois protocolos diferentes simultaneamente; e finalmente, uma configuração do *middleware* para que ele possa servir como uma ponte entre dois protocolos diferentes. Os arquivos de configuração dos componentes para cada um dos exemplos estão no apêndice C.

PUC-Rio - Certificação Digital Nº 0321107/CA

5.1.1
Seleção de protocolo de comunicação

O modelo de componentes do OiL nos permite selecionar o protocolo utilizado pelo *middleware*. Implementamos para provar esse conceito um protocolo simples chamado de LuaDummy. A conexão é feita através de *sockets* TCP e a codificação é feita na forma de serialização de objetos Lua, ou seja, são enviados pedaços de código Lua que são interpretados no objeto remoto antes de serem repassados às camadas superiores do *middleware*.

Por ser simples, um dos requisitos desse protocolo foi não possuir todas as funcionalidades utilizadas em um protocolo mais complexo, como o GIOP do CORBA. Por isso, ele não utiliza um repositório de interfaces, nem as usa

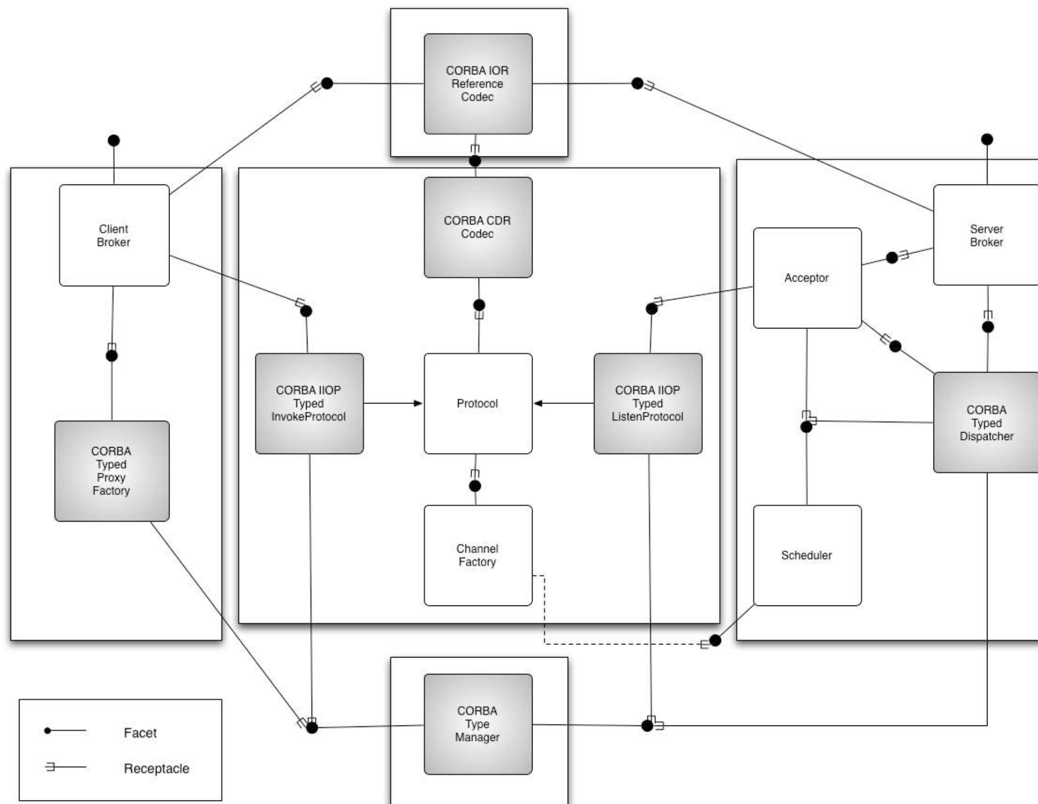

Figura 5.1: OiL com o protocolo CORBA instanciado.

para definição dos tipos dos objetos. Dessa forma, o protocolo LuaDummy não utiliza os tipos de componente `Typed`, eliminando a necessidade do bloco `Interface`. Consideramos esse, portanto, um bom exemplo de configuração do protocolo de comunicação.

A implementação de componentes específicos para CORBA foi feita a partir da implementação anterior do OiL, que já dava suporte a esse protocolo, através de uma refatoração do código. O diagrama na figura 5.1 mostra a configuração usando CORBA como o protocolo escolhido. Já para o LuaDummy, tivemos que criar os seus componentes específicos, como mostrado na figura 5.2.

Como ambos os protocolos são baseados em chamadas remotas de procedimento, pudemos reutilizar componentes que independem da informação do tipo de protocolo. Essa reutilização também pode ser vista na comparação entre as duas figuras. Por exemplo, o `Acceptor`, `ChannelFactory` e os dois componentes que falam diretamente com o desenvolvedor da aplicação, `ClientBroker` e `ServerBroker` não sofreram modificação.

Alguns componentes, no entanto, precisam de uma especialização maior para um protocolo como CORBA, já que eles necessitam do bloco `Interface`, como é o caso do `ProxyFactory` e `Dispatcher`. Mesmo assim, em um protocolo

PUC-Rio - Certificação Digital Nº 0321107/CA

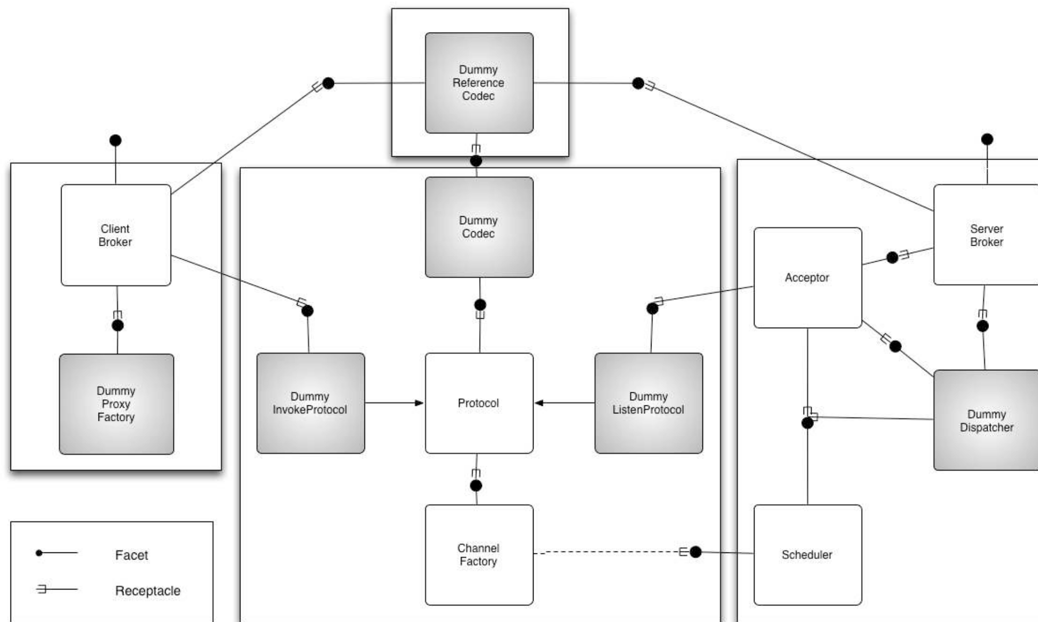

Figura 5.2: OiL com o protocolo LuaDummy instanciado.

PUC-Rio - Certificação Digital Nº 0321107/CA

mais simples, como o LuaDummy, podemos reutilizar esses componentes mais especializados, pois eles, além de possuir a mesma interface de programação, ainda foram construídos de forma a perceber se há a conexão com o bloco `Interface` e, se não houver, ignorar a tipagem do objeto em questão.

5.1.2 Seleção do tipo de transporte

Em algumas situações, pode ser necessário trocar o tipo de canal de transporte utilizado pelo ORB para fazer as chamadas remotas. No caso de aplicações que precisem de segurança, pode não ser desejado usar a transmissão através de *sockets* TCP simples. Há então a necessidade da criação de um canal de transporte SSL para ser utilizado no lugar do canal de transporte tradicional.

Para a implementação de SSL no OiL, foi criada uma implementação do componente `ChannelFactory` que, neste caso, fornece canais de transporte SSL, desde que o desenvolvedor, na configuração do componente, forneça as informações de certificados de segurança requeridos pelo SSL.

Dessa forma, a complexidade do uso de SSL como transporte fica encapsulada no novo componente. Todos os outros componentes não precisam ser modificados, como mostrado na figura 5.3.

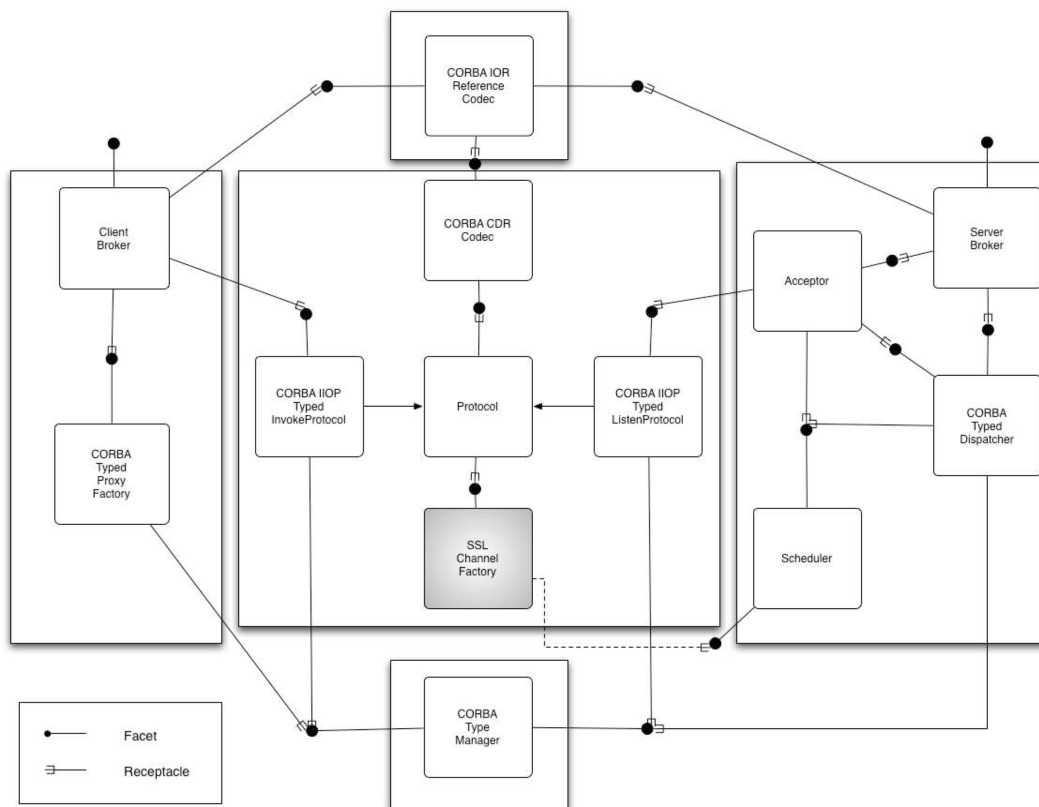

Figura 5.3: Seleção do tipo de transporte.

5.1.3 Seleção do tipo de escalonador

O objetivo deste exemplo é demonstrar a capacidade de incluir um escalonador no OiL com um mínimo de modificações nos outros componentes. O componente `Scheduler`, que no nosso modelo faz parte do bloco `Servant`, possui duas implementações: uma, mais simples, que executa em série qualquer tarefa que é passada para ele; outra, mais complexa, usa o modelo de corrotinas mencionado no capítulo 4, criando um novo *thread* para cada tarefa que é registrada através de sua faceta `threads`.

Para o uso desse escalonador, algumas modificações devem ser efetuadas nos dois componentes no bloco `Servant` que o utilizam diretamente: tanto o `Acceptor` quanto o `Dispatcher` utilizam as diretivas para suspender um *thread* e para continuá-lo quando a execução do *thread* atual estiver bloqueada. As modificações dos componentes necessários para a escolha do tipo do escalonador são representadas na figura 5.4.

PUC-Rio - Certificação Digital Nº 0321107/CA

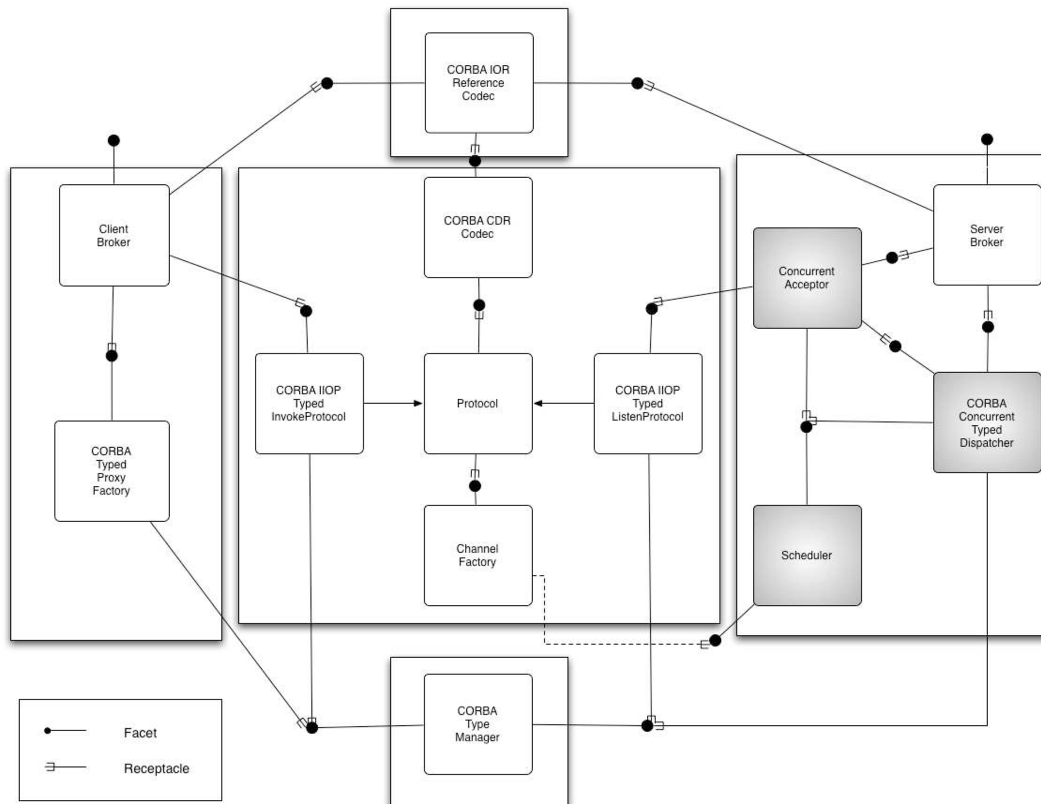

Figura 5.4: Seleção do tipo de escalonador.

PUC-Rio - Certificação Digital Nº 0321107/CA

5.1.4
Servidor respondendo múltiplos protocolos

Um outro exemplo prático é a implementação de um servidor usando o OiL que responda mais de um protocolo simultaneamente, pois pode ser interessante exportar um mesmo serviço para clientes de mais de um *middleware* diferente. Assumimos que haja, para cada *middleware* diferente, a especificação da interface dos objetos exportados por esse servidor. A implementação do servidor, em Lua, também deve se comportar de maneira similar para qualquer tipo de conexão que ele aceite, seja de qual for o *middleware* que a inicie. Através da criação desse tipo de servidor, pode-se mostrar que o modelo trata bem dois ou mais tipos de middleware, apenas através da modificação da pilha de protocolos relacionada com cada conexão.

Criamos, então, um servidor que respondia ao mesmo tempo requisições CORBA e LuaDummy. O servidor, para cada um dos protocolos, criou um *Acceptor*, que possui uma porta de comunicação específica e, com isso, uma referência específica. As duas referências ficam disponíveis para clientes de cada um dos dois protocolos que queiram se comunicar com os objetos registrados nesse servidor. Demonstramos que podemos reutilizar o mesmo `Dispatcher`

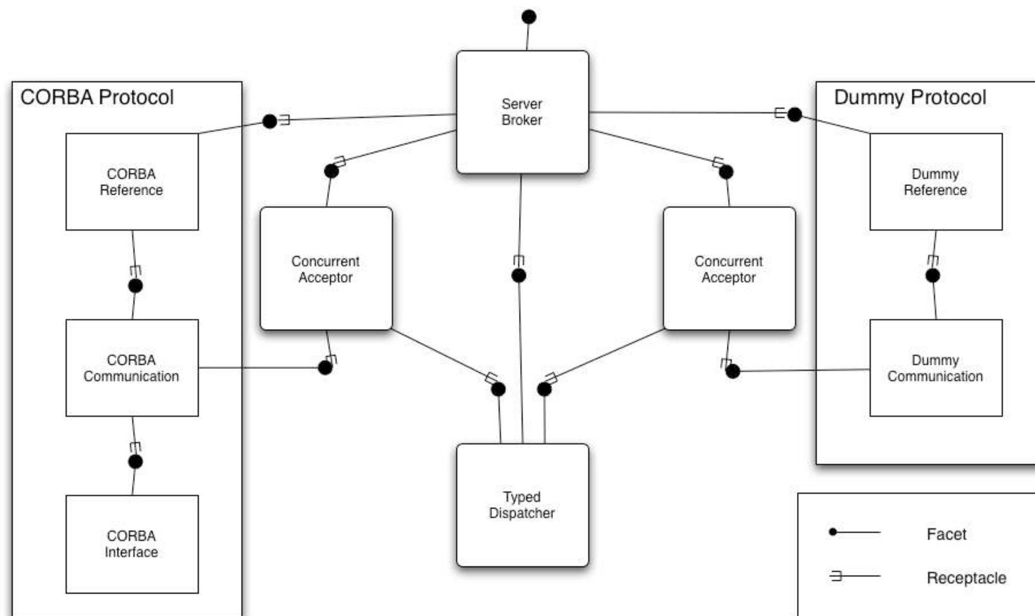

Figura 5.5: Servidor respondendo dois protocolos simultaneamente.

PUC-Rio - Certificação Digital Nº 0321107/CA

para ambos os protocolos e, com isso, o mesmo objeto registrado neste `Dispatcher` é acessível por requisições que chegam nas portas relacionadas aos respectivos `Acceptors`.

5.1.5 Ponte entre protocolos

Construímos também um exemplo de uso do OiL através de uma aplicação que funciona como uma ponte entre dois *middlewares* diferentes. Com isso, demonstramos a capacidade de instanciar componentes de protocolos diferentes para blocos diferentes do *middleware*.

Criamos então um aplicativo que é ao mesmo tempo cliente CORBA e servidor LuaDummy. O bloco `Communication` na verdade é misto, pois o lado cliente é representado por um `InvokeProtocol` que implementa o protocolo de CORBA e um `Codec` que implementa o CDR, também de CORBA. A eles são conectados os componentes dos blocos `Reference` e `Interface`, assim como o `ProxyFactory`, do bloco `Invocation`.

Já o lado servidor do bloco `Communication` é representado pelo componente `ListenProtocol` implementando o protocolo LuaDummy, assim como o `Codec` correspondente. Há também um bloco `Reference` específico para este protocolo. Os dois protocolos utilizam o mesmo `ChannelFactory`. A figura 5.6 apresenta essa configuração.

O aplicativo que utiliza essa configuração de *middleware* apenas registra um objeto que possui uma referência para um objeto externo CORBA. Assim,

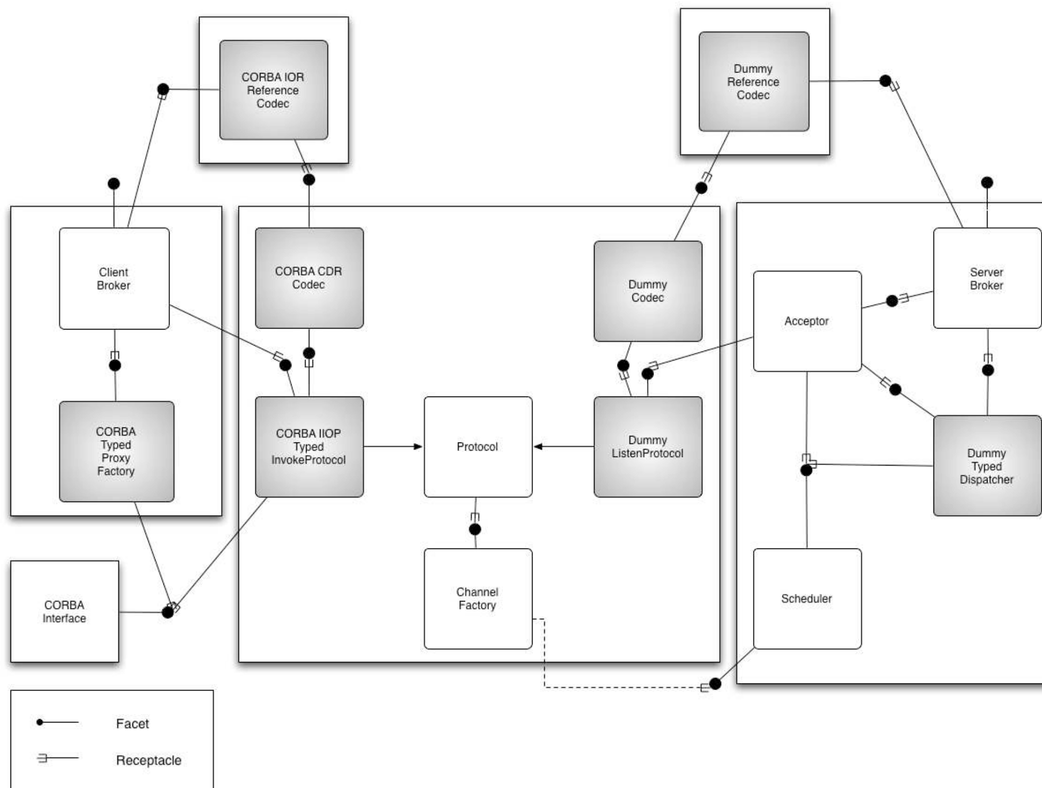

Figura 5.6: *Middleware* configurado para criação de um *proxy* entre protocolos.

as chamadas efetuadas naquele objeto são repassadas para este. A chamada recebida usando o protocolo LuaDummy é decodificada usando o objeto decodificador correspondente, recodificada usando o CDR e enviada ao objeto externo.

5.2 Desempenho

A análise de desempenho do OiL baseado no modelo de componentes pode ser efetuada através da comparação com a versão anterior, não componentizada. Criamos três tipos de operações, cujas interfaces IDL estão apresentadas na listagem 5.1:

1. Requisição tipo 1: requisições simples, passando como parâmetro uma *string* retornando uma *string*.

2. Requisição tipo 2: requisições um pouco mais complexas, com dois parâmetros, sendo cada um uma estrutura com dois campos, um deles sendo uma *string* e outro um inteiro.

PUC-Rio - Certificação Digital Nº 0321107/CA

3. Requisição tipo 3: requisições com uma estrutura com um dos campos uma *string* e o outro uma seqüência com tamanho variável de estruturas com dois campos *string*.

```
1  struct StructOne {
2    short number;
3    string txt;
4  };
5  struct StructTwo {
6    string name;
7    string email;
8  };
9  typedef sequence<StructTwo> StructSeq;
10 struct ComplexStruct {
11   string title;
12   StructSeq structs;
13 };
14 interface Hello {
15   string request1(in string name);
16   void request2(in StructOne struct1, in StructOne struct2);
17   void request3(in ComplexStruct struct1);
18 };
```

Listagem 5.1: Definição IDL das operações usadas na análise de desempenho.

Além disso, comparamos os dois *middlewares* usando configurações com e sem escalonador. Fizemos os testes na mesma máquina [1], para diminuir o impacto da rede durante as requisições. Os aplicativos cliente e servidor, que utilizaram os três tipos de requisição, executaram exatamente as mesmas operações. Fizemos uma média entre 10 execuções dos testes, cada teste executando 10000 vezes cada operação.

Os resultados da média do tempo necessário para processar as 10000 requisições nas duas configurações de cada *middleware* para cada um dos tipos de requisição é demonstrado na tabela a seguir:

Tipo de requisição	**OiL tradicional**	**OiL com componentes**
Sem escalonador		
Requisições tipo 1	8.78 s	8.55 s
Requisições tipo 2	10.13 s	11.12 s
Requisições tipo 3	43.6 s	40.60 s
Com escalonador		
Requisições tipo 1	10.08 s	9.54 s
Requisições tipo 2	11.48 s	11.53 s
Requisições tipo 3	44.81 s	41.52 s

[1] A máquina utilizada para os testes tem um processador Intel Core Duo 1.66 GHz, 1 GB de memória RAM e executa o sistema Mac OS X versão 10.4.7

PUC-Rio - Certificação Digital Nº 0321107/CA

Notamos pelos resultados que, mesmo fazendo a separação de interesses em componentes, o processamento por requisição não ficou prejudicado. Em alguns dos casos tivemos até uma melhora no tempo de processamento por requisição. A diferença entre os casos com e sem escalonador pode ser explicada pela própria utilização das corrotinas. O aumento da eficiência com o uso do escalonador aparece apenas quando os atrasos na leitura e escrita de dados na rede são significativos o suficiente, o que não acontece quando cliente e servidor se encontram na mesma máquina. Consideramos o tempo de carregamento do *middleware*, por volta de 0.1 segundos, irrisório perto do tempo total de processamento das requisições.

Fizemos também alguns testes simples para ver se a inclusão do modelo de componentes impactou de forma significativa o uso de memória. A versão antiga do OiL, ao ser carregada, na sua versão sem escalonador, ocupava 814Kb de memória, enquanto a versão nova, com a mesma configuração utilizada nos testes de desempenho acima, também sem o escalonador, ocupava em torno de 882Kb de memória. Depois do registro de objetos, também de forma idêntica nas duas versões do OiL e da execução dos testes, ambas as versões aumentaram o consumo de memória, sendo que a versão antiga passou a ocupar 824Kb enquanto a versão com o modelo de componentes passou a consumir 915Kb.

PUC-Rio - Certificação Digital Nº 0321107/CA

5.3 Considerações finais

O modelo proposto de componentes utilizava muitos conceitos que estavam enraizados na implementação original do OiL. Por isso, a divisão dos blocos inicialmente sugerida era feita de acordo com a divisão do protocolo CORBA, que tinha como verdadeiras premissas que muito provavelmente não seriam para todos os protocolos que conhecemos. Por exemplo, podemos citar o repositório de interfaces, utilizado no CORBA para a comunicação entre processos, para que os objetos que se comuniquem saibam qual é a interface exata de cada um deles. Esse repositório pode não ser utilizado por algum protocolo e, no entanto, estava presente na implementação original.

Durante a confecção dos exemplos de uso do OiL baseado em componentes, descobrimos alguns problemas com esse modelo proposto inicialmente. Para o primeiro exemplo, a troca do protocolo de comunicação, decidimos que seria interessante criar um protocolo mais simples, para mostrar como seria fácil a integração com esse modelo de componentes. Entretanto, esse protocolo simples não necessita do repositório de interfaces e, como o modelo dependia da existência dele, o encaixe do novo protocolo no modelo não ocorreu de forma direta. Assim, o modelo teve que ser modificado, de forma a separar partes

que dependiam do repositório de interfaces, criando assim as versões tipadas de vários dos componentes.

Outras funcionalidades, como a mudança do tipo de canal de transporte, também precisaram de mudanças no suporte a esses canais por parte da máquina virtual Lua. No exemplo em que utilizamos canais SSL no lugar de *sockets* comuns, precisamos incluir o suporte a esse tipo de transporte no LuaSocket. Já os exemplos que necessitam do escalonador, como por exemplo a ponte entre protocolos e o servidor que responde a mais de um protocolo ao mesmo tempo precisaram apenas de ajustes na forma como o modelo incluía ou não o escalonador. Além disso, no exemplo do servidor que responde a mais de um protocolo o modelo precisou ser modificado de forma a permitir a existência de mais de um bloco *Acceptor*, referenciados pelo bloco *ServerBroker*, que fica então responsável de descobrir qual das pilhas de protocolos deve ser acionada de acordo com a requisição que chega no servidor.

De maneira geral, a implementação dos exemplos permitiu observarmos a robustez do modelo que escolhemos. Pudemos perceber que o modelo teve que evoluir de acordo com alguns dos usos que resolvemos utilizar mas que, mesmo com essas evoluções, ele se adaptou corretamente. Pudemos observar também que o esforço de implementação dos exemplos não foi muito grande, uma vez que o modelo estava preparado. A prototipação de um novo protocolo, simples como o do primeiro exemplo, demorou um pouco mais de um dia, enquanto que a implementação do último exemplo, que já foi feita em cima do modelo definitivo, demorou apenas algumas horas.

PUC-Rio - Certificação Digital Nº 0321107/CA