Avaliação de desempenho de conectores de software

Alexandre Sztajnberg
DICC/IME/UERJ
alexszt@ime.uerj.br

Orlando Loques
IC/UFF
loques@ic.uff.br

Resumo

Aplicações modernas, que estão migrando do paradigma de sistemas distribuídos para sistemas móveis e ubíquos, precisam que a "cola" oferecida pelo middleware seja flexível, permitindo que os padrões de interação entre os componentes possam ser adaptados dinamicamente. Esta flexibilidade é obtida com uso de middlewares adaptáveis e reflexivos, que são baseados no uso de técnicas de indireção e mecanismos de acesso a informações e procedimentos de meta-nível. Existe um custo computacional introduzido pela intermediação do middleware na interação entre os componentes destas aplicações. Neste contexto, há considerações a se fazer com relação a aplicações que são elegantemente modeladas com o suporte de um middleware, mas têm restrições de desempenho. Neste trabalho apresentamos um estudo e avaliação de desempenho do middleware reflexivo de R-RIO composto por conectores genéricos de software e um serviço de gerência de configuração. Os resultados desta avaliação podem ser generalizados para outros middlewares dado o uso crescente de padrões em suas implementações.

Abstract

Modern applications, migrating form distributed systems to mobile and ubiquitous systems paradigm, need the middleware to provide a flexible "glue", allowing interaction patterns between components to be dynamically adapted. This flexibility is achieved with the use of adaptive and reflective middleware that are based on the use of indirection techniques and mechanisms to access meta-level information and procedures. There is a computational overhead introduced by the middleware mediation of component interactions of these applications. In this context there are some questions regarding applications that are elegantly deployed with the support of a middleware but have performance restrictions. In this paper we present a study and performance evaluation of the R-RIO reflective middleware composed of generic software connectors and a configuration management service. The results of this evaluation can be applied to other middleware due to the growing adoption of common patterns in their implementation.

1. Introdução

Middleware é uma camada de *software* que oferece ao projetista de aplicação uma visão uniforme e uma interface de programação abstrata para os recursos do sistema operacional e da rede. O *middleware* pode ser visto como uma "cola" que permite compor e interligar componentes de uma aplicação que aderem a determinados requisitos de interação. A interligação de componentes é realizada baseando-se apenas na assinatura de suas interfaces, e não em sua representação [Agha 2002].

Aplicações modernas, que estão migrando do paradigma de sistemas distribuídos para sistemas móveis e ubíquos, precisam que a "cola" oferecida pelo *middleware* seja flexível, permitindo que os padrões de interação entre os componentes possam ser adaptados dinamicamente. Neste contexto, a separação de interesses funcionais dos não-funcionais é fundamental tanto na arquitetura do *middleware*, quanto na arquitetura da aplicação [Sztajnberg 2002].

Para atender a esta nova demanda das aplicações propõe-se que os novos *middlewares* sejam adaptáveis ("*adaptive middleware*"), provendo facilidades para a configuração dinâmica de módulos da aplicação e a gerência de recursos dos subsistemas de suporte. Isto é obtido através de interfaces para serviços de meta-nível por meio de técnicas de reflexão [Kon 2002]. É desejável que tanto as aplicações quanto o próprios serviços e recursos utilizados pelo *middleware* possam ter suas informações de meta-nível consultadas e possam ser dinamicamente adaptadas.

De uma forma geral, *middlewares* para sistemas de objetos distribuídos utilizam como mecanismo básico a técnica de RPC (*Remote Procedure Call*) [Birrel 1984] ou suas variações mais “requintadas” (RMI, CORBA, Jini, DCOM) para prover a interação entre objetos remotos com alguma transparência [Laddga 1997]. Atualmente a implementação destes *middlewares* tende a ser realziada de maneira mais sistemática, adotando-se *design patterns*, tais como o *Proxy* e o *Serializer* [Gama 1995]. Para prover a flexibilidade de acesso a informações de meta-nível e a capacidade de adaptação, as implementações de *middleware* também tendem a adotar em sua implementação uma combinação de *design patterns* de acesso / configuração (*Wrapper*

Facede, *Component Configurator*, *Interceptor*, *QoS Connector*, por exemplo) e de concorrência (*Active Object*, *Monitor Object*, etc.) [Schmidt 2000, Cross 2002]. Esta prática facilita o entendimento dos serviços oferecidos pelo *middleware* e permitem sua própria evolução. Assim, observa-se que, mesmo baseados em conceitos diferentes, *middlewares* modernos compartilham características comuns [Crane 1995, Schmidt 1999].

Neste contexto, há considerações a se fazer com relação a determinadas aplicações que são elegantemente modeladas com o suporte de um *middleware*, mas têm restrições de desempenho. A flexibilidade provida pelo uso de *middlewares* adaptáveis e reflexivos está, como citado anteriormente, associada ao uso de padrões que utilizam técnicas de interceptação de mensagens, *marshaling* / *demarshaling* de parâmetros, serialização de objetos e invocação dinâmica de métodos [Coulson 2000]. Existe um custo computacional introduzido pelo uso destas técnicas que podem afetar diretamente o desempenho destas aplicações. Nós destacamos duas questões gerais:

- o custo computacional introduzido pela intermediação do *middleware* na interação dos componentes e
- o custo computacional de um procedimento típico de adaptação ou reconfiguração de uma aplicação ou do próprio *middleware*.

Em [Sztajnberg 2002] apresentamos o *Framework* R-RIO que integra Arquiteturas de Software [Shaw 1996] e Programação de Meta-Nível, facilitando a concepção e manutenção de aplicações distribuídas que precisam evoluir dinamicamente. Uma das atividades deste trabalho foi avaliar o desempenho dos conectores de *software* em conjunto com o serviço de gerência de configuração, elementos do *middleware* reflexivo de R-RIO. Gostaríamos, então, de compartilhar as experiências desta avaliação com a comunidade de pesquisa em sistemas distribuídos e *middleware*.

O restante do texto está dividido da seguinte forma. Inicialmente apresentamos o nosso modelo de conector de *software* genérico e uma aplicação simples, alvos de nossas medidas. Em seguida apresentamos uma primeira seqüência de medidas e uma análise preliminar das mesmas. Depois, o caminho crítico de uma interação entre objetos através do conector é examinado mais detalhadamente. Em seguida elencamos algumas otimizações realizadas na estrutura do conector a partir da primeira fase de medidas, e uma segunda seqüência é, também, apresentada. Por último relacionamos alguns trabalhos correlatos e nossas observações finais.

1.2. Conectores Reflexivos por Contexto em R-RIO

Componentes de uma aplicação em R-RIO são interconectados por conectores de *software*, que intermedeiam a interação destes componentes e podem encapsular procedimentos de acesso aos recursos necessários para atender aos aspectos operacionais da aplicação. As aplicações em R-RIO executam com o suporte de um serviço de gerência de configuração, que oferece APIs para (i) acesso à informações de meta-nível da arquitetura das aplicações sendo executadas (API de Reflexão Arquitetural - uma forma de reflexão estrutural) e (ii) para atividades de configuração e reconfiguração destas aplicações (API de Configuração).

Em [Carvalho 2001] os elementos de R-RIO são apresentados como um (*design*) *pattern language*, denominado *Architecture Configurator*. A figura 1 apresenta o diagrama de classes da arquitetura de uma aplicação cliente-servidor, utilizando uma configuração deste *pattern*. Podem ser identificados os elementos do conector de *software* (com um *Proxy* e uma composição de objetos da classe *Conector*) e os elementos do serviço de gerência de configuração (com a classe *Gerente de Configuração*, o repositório de meta-nível contendo as informações da arquitetura em execução e as interfaces das APIs mencionadas anteriormente). Verificamos que no *Architecture Configurator* são utilizados *design patterns* mais elementares como o *Proxy* e o *Chain-of-Responsibilities* (para encadear instâncias da classe *Connector*), que se encontram presentes na idéia original de RPC (por exemplo, na cadeia de stubs → conversor XDR → segurança).

Durante a elaboração do primeiro protótipo de R-RIO foi desenvolvido o conceito de conectores de *software* genéricos, reflexivos por contexto [Lobosco 1999]. A idéia de conectores reflexivos por contexto permite que um conector de *software* seja utilizado para interligar quaisquer pares de módulos, independentemente da interface dos mesmos, como esquematizado na figura 2. Durante a configuração de uma aplicação, quando dois módulos são interligados para interagir, o conector que irá intermediar a interação dos mesmos é adaptado às suas respectivas interfaces.

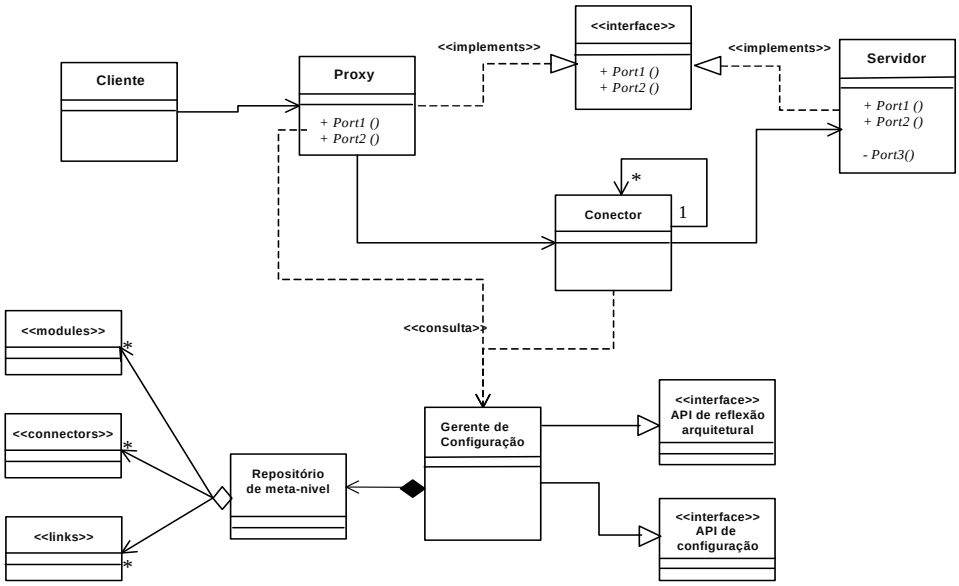

Figura 1. Architecture Configurator

A figura 3 apresenta a configuração de um arquitetura simples com dois módulos, instanciados em nós distribuídos, que irão interagir por intermédio de um conector encapsulando o mecanismo de *sockets* (que podemos considerar o "*assembly*" dos *middlewares*). Após a criação dos objetos *Cliente* e *Servidor*, são criados os objetos das classes que encapsulam os detalhes do *middleware* de comunicação (neste caso, *sockets*) - primeira parte da figura 3. O objeto do lado do cliente, pode encapsular elementos básicos deste *middleware*, tais como *sockets* ou *stubs* cliente, por exemplo, e o outro, do lado do servidor, da mesma forma pode encapsular *sockets*, *stubs* ou *skeletons* servidores. Em seguida (segunda parte da figura 3), a assinatura da interface do objeto servidor é obtida através da API de Reflexão Arquitetural. Esta informação é então utilizada para a adaptação de um *Proxy* (no momento de sua instanciação) que representa o objeto servidor perante o objeto cliente, e para a adaptação da parte servidora do conector que deverá montar invocações dinâmicas a serem realizadas pela implementação do objeto servidor.

O objeto *Cliente* realiza as chamadas ao objeto *Servidor* normalmente. Estas chamadas são, na realidade, direcionadas para o objeto *Proxy*, que tem a mesma interface do objeto *Servidor* original. O *Proxy* identifica para qual conector repassar a requisição (através da API de Reflexão Arquitetural) e cuida de empacotar as informações necessárias para que a chamada ao objeto *Servidor* seja realizada na estação remota. Em seguida o *Proxy* encaminha estas informações para a parte cliente do conector através de uma interface padronizada, que serializa e transmite as mesmas. Na estação remota, a parte servidora do conector recupera as informações necessárias e monta dinamicamente uma chamada para o objeto *Servidor*, como se fosse o objeto *Cliente*. O retorno de resultados e as exceções que possam surgir são transferidos de forma semelhante.

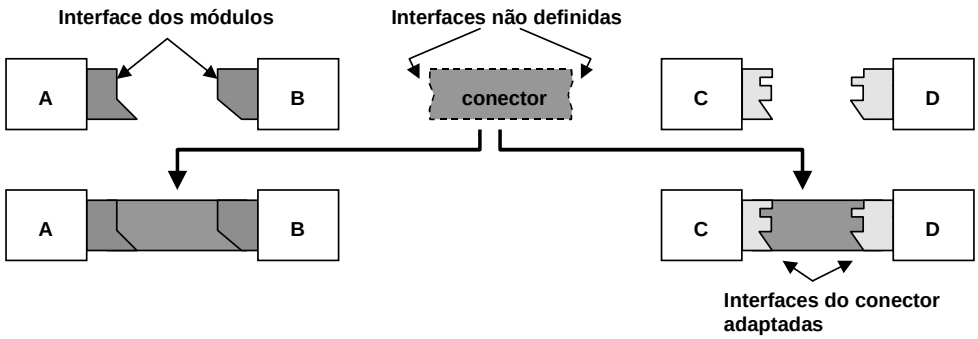

Figura 2. Conector genérico

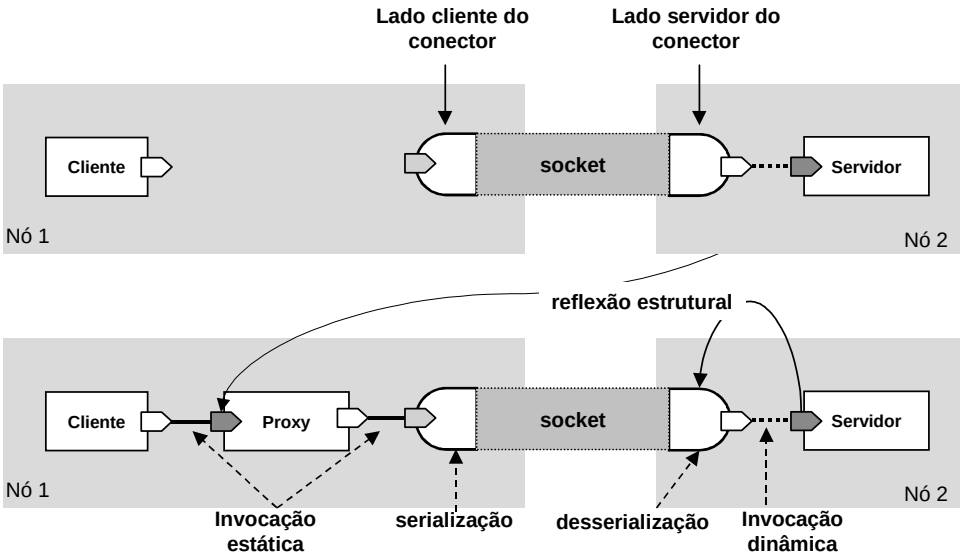

Figura 3. Funcionamento do conector reflexivo por contexto

A concepção de um conector segue passos bem definidos e assim sendo permite que novos conectores sejam implementados de forma sistemática. Além disso, por ter uma interface padronizada, os conectores são intercambiáveis, ficando ao encargo do *Proxy* e da utilização de reflexão estrutural, fazer as adaptações ao contexto em operação. Por exemplo, na situação da figura 3 seria possível substituir o conector *socket* por um conector CORBA ou RMI por questões de interoperabilidade. A técnica permite que esta substituição ocorra em tempo de execução, facilitando a evolução dinâmica da aplicação em face de novos requisitos operacionais. Uma outra característica que também surge em conseqüência da utilização deste mecanismo é a possibilidade de se substituir dinamicamente os objetos que estão interagindo. No exemplo anterior, o cliente poderia ter desfeita sua ligação com o servidor atual e esta ser refeita com um segundo servidor, desde que este mantivesse a mesma interface.

A ênfase de nosso trabalho foi o modelo conceitual de R-RIO e sua aplicabilidade. Entretanto, durante o desenvolvimento do protótipo do serviço de gerência de configuração e das primeiras aplicações de teste, foram realizadas algumas medidas para verificar o custo computacional da utilização dos conectores de *software*. Nestas medidas, dois aspectos foram verificados: (i) o *overhead* no uso dos conectores como intermediário da interação de objetos em relação ao custo para realizar invocações de métodos diretamente pela JVM e (ii) o custo dos conectores encapsulando mecanismos de comunicação em comparação com o uso direto destes mecanismos.

2. Cenário das medidas

As medidas foram realizadas em duas maquinas Ultra 1 Sparc com sistema operacional Solaris 5.1 da Sun Microsystems. A infra-estrutura de R-RIO e as aplicações de teste foram programadas em Java e compiladas para a versão 1.2 da JVM. As máquinas foram isoladas do restante da rede, mas o ambiente de operação (processos carregados, nível do sistema Unix, escalonamento de processos, etc.) não foi alterado.

Para realizar as medidas construímos uma aplicação simples com um módulo cliente e um módulo servidor (como a da figura 3). Duas versões desta aplicação foram utilizadas: uma com a infra-estrutura de R-RIO (conectores, serviço de gerência de configuração, etc.) e outra apenas com os recursos nativos da JVM (sem conectores).

As medidas registraram o intervalo de tempo de uma transação composta de uma invocação de método, o processamento e o retorno dos resultados. A interface do método do servidor consistiu de uma cadeia de caracteres, utilizando-se a classe *String*, como argumento de entrada, e o retorno de um inteiro, utilizando-se a classe *int*. Foram feitas medidas variando-se o tamanho da *String* de 1 até 32 k caracteres. Para cada tamanho de *String* foram realizadas 10 medidas com 100 invocações de método. Para o levantamento das curvas foi calculada a média dos valores. Nas medidas da segunda fase foi também calculado o desvio padrão.

Toda infra-estrutura de R-RIO foi instrumentada para a cronometragem de tempos, tomando-se o cuidado de não se medir o tempo de

processamento no objeto cliente nem no objeto servidor. Não havia processamento, nem criação de instâncias de objetos no servidor, que simplesmente recebia o argumento e retornava um inteiro como resultado. O desempenho foi medido no caminho crítico da interação dos módulos, mais especificamente dentro do código dos conectores em R-RIO.

Três tipos de mecanismos / conectores foram avaliados:

(a) invocação direta de método, que chamamos de MI, com os objetos cliente e servidor na mesma JVM;

(b) invocações remotas de método (*Remote Procedure Call* - RMI), com objetos cliente e servidor em maquinas distintas e,

(c) *sockets*, também com os objetos cliente e servidor em maquinas distintas

Os resultados obtidos são apresentados e discutidos a seguir.

3. Primeira fase

3.1. Invocação de método local (MI)

Nas medidas com os objetos cliente e servidor na mesma estação, foram comparados os seguintes casos:

1. invocações a método tradicional, ou seja, cujo código é ligado estaticamente pelo compilador Java, com a inclusão de um objeto intermediário simples;
2. mesmo do caso 1, acrescentando-se a compactação dos argumentos e invocação dinâmica;
3. mesmo do caso 1, acrescentando-se a serialização dos argumentos.

O objetivo do objeto intermediário foi verificar qual o impacto do emprego de um nível de indireção adicional nas invocações de método (comparar conectores apenas com invocações estáticas de método não nos traria nenhuma informação útil), como no conector em R-RIO (e como ocorre na maioria dos *middlewares* através de *stubs*). A compactação dos argumentos de entrada (no caso um objeto da classe *String*) é realizada por uma classe "container" chamada *ArgumentHolder*. O mecanismo de invocação dinâmica e a serialização de objetos foram programadas usando-se as classes disponíveis em Java [Gosling 1996].

Estas medidas foram comparadas com a aplicação em R-RIO utilizando-se um conector *nulo*, que contém todos os procedimentos de um conector genérico, mas simplesmente repassa as invocações de métodos e respectivos retornos sem realizar nenhuma outra operação. O gráfico da figura 4 apresenta as medidas.

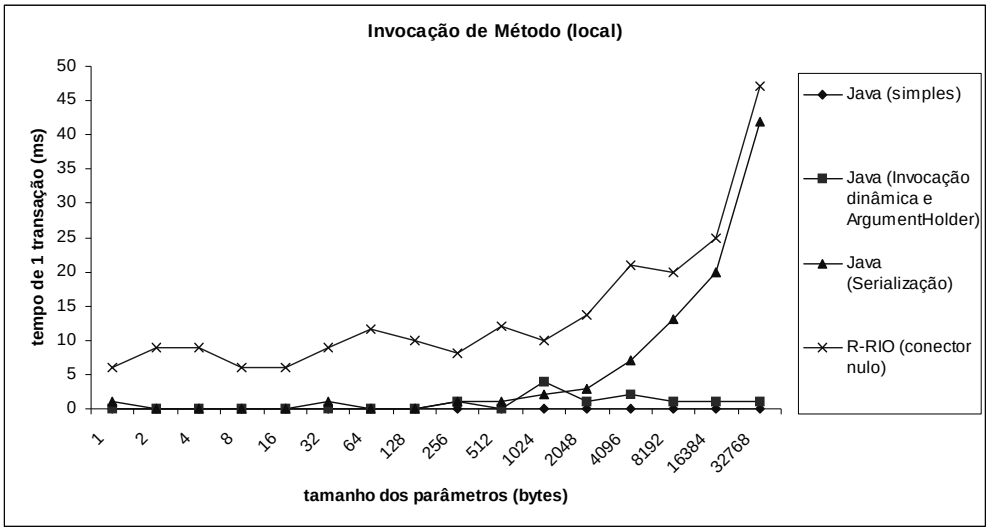

Figura 4. Medidas de desempenho para invocação de método (1a fase)

3.2. RMI

Comparamos o desempenho do mecanismo de RMI, utilizado isoladamente, e um conector R-RIO encapsulando este mecanismo. Observa-se que ao se utilizar RMI em sua forma estática, não existe a necessidade de se serializar explicitamente os argumentos dos métodos dos módulos que estão interagindo, pois isto é feito pelo mecanismo de *marshalling* executado internamente nos *stubs* [Sun 1998]. Este é o caso tanto da aplicação de teste usando RMI diretamente, como da aplicação utilizando o conector RMI. Entretanto, na aplicação em R-RIO ainda é necessária a utilização de invocações dinâmicas e consultas ao repositório de meta-nível, para que o módulo servidor seja acionado. A figura 5 apresenta as medidas deste experimento.

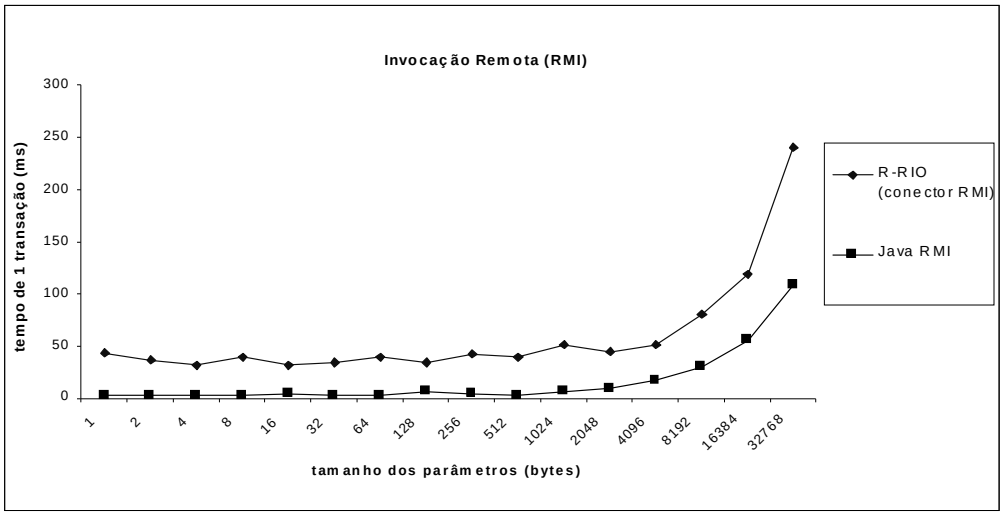

Figura 5. Medidas de desempenho para RMI (1a fase)

3.3. *Socket*

O mesmo conjunto de medidas utilizado para o mecanismo de RMI foi realizado para a aplicação que empregou diretamente o mecanismo de *sockets* do pacote *java.net* de Java. Neste caso, tanto a aplicação em Java, quanto a aplicação em R-RIO utilizaram a serialização de argumentos, mas somente o conector utilizou invocação dinâmica. O resultado das medidas se encontra na figura 6.

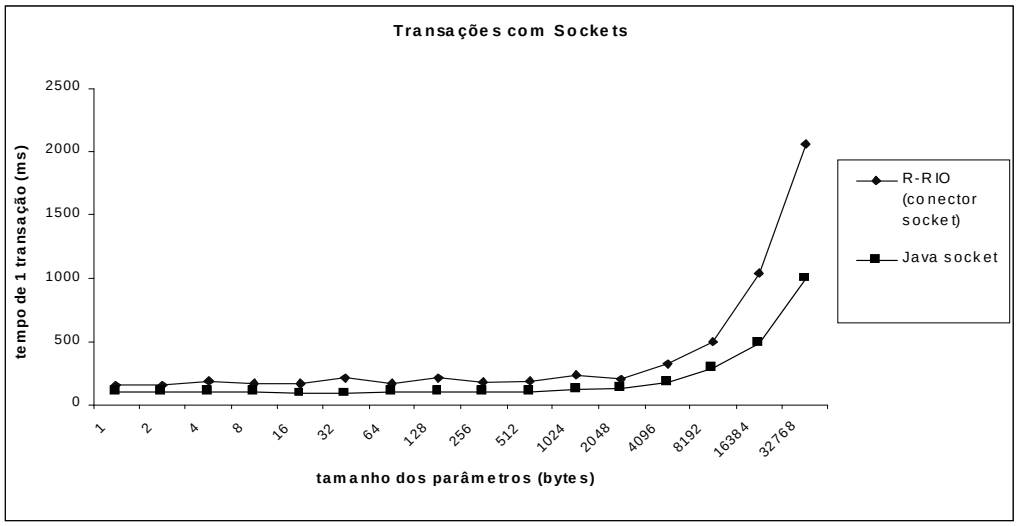

Figura 6. Medidas de desempenho para *sockets* (1a fase)

3.4. Análise

Do gráfico apresentado na figura 4 observa-se que o custo do chaveamento de contexto entre objetos, invocação de métodos e da invocação dinâmica é pequeno, independente do tamanho do argumento. O mecanismo de serialização parece ter um custo elevado e o tempo necessário para a operação em cadeias de caracteres (encapsuladas em objetos da classe *String*) maiores cresce muito com o tamanho das mesmas. O conector de *software* embute os custos dos mecanismos avaliados individualmente e ainda inclui outros pontos onde o tempo é consumido, por exemplo, as consultas por reflexão estrutural, que devem ser investigados.

A variação de algumas partes das curvas pode ser atribuída ao suporte da JVM que pode iniciar uma rotina de manutenção a qualquer instante. Estão incluídos aí o mecanismo de coleta de lixo (*garbage collection*) e o escalonamento. Além disso, como o sistema operacional não sofreu preparos especiais para as medidas, a JVM é gerenciada como um processo Unix comum, passível de preempção.

As medidas de RMI e *sockets*, figuras 5 e 6, apresentam-se dentro do esperado. Existe um *overhead*

no uso do mecanismo conector que se mantém proporcional para a variação no tamanho do argumento. Observa-se que a diferença entre os experimentos com RMI é maior que entre os experimentos com *sockets*. Surpreendentemente, entretanto, as medidas com o mecanismo de *socket*, independente do uso do suporte R-RIO, apresentaram-se mais custosas do que as com o mecanismo de RMI. Em princípio isto seria um contra-senso já que o mecanismo de RMI utiliza o mecanismo de *sockets* para realizar a comunicação. Este ponto será examinado adiante.

4. Caminho crítico de um conector

Com os resultados da primeira fase medidas, verificou-se a necessidade de avaliar-se onde exatamente estariam as fontes da diferença de *overhead* entre as aplicações com suporte R-RIO e as aplicações utilizando apenas Java e seus serviços, uma vez que R-RIO é inteiramente desenvolvido em Java.

Com esta finalidade o código do serviço de gerência de configuração de R-RIO (o Gerente de Configuração), a aplicação em teste e o conector *nulo* (incluindo o *Proxy*) foram reinstrumentados. O objetivo foi fazer uma análise qualitativa do código e identificar onde se perde mais tempo. As medidas foram feitas para um argumento de 100 caracteres.

Reexaminando a figura 3, em uma transação completa da aplicação de teste com o suporte de R-RIO estão envolvidos quatro elementos: o módulo cliente, o módulo servidor, o *Proxy* e o conector. O *Proxy* e o conector precisam criar objetos e consultar o repositório de informações de meta-nível. O *Proxy* ainda precisa empacotar os argumentos do método a ser invocado. O conector, por sua vez precisa serializar os dados para serem transmitidos e desserializar os mesmos em sua chegada para, então, realizar uma invocação dinâmica. No caso do conector *nulo*, a serialização não é necessária pois os dados não precisam ser transmitidos.

Cada uma das etapas de uma transação foi medida. A tabela 1 apresenta os resultados (em milisegundos, com a precisão limitada pelo mecanismo de medida de tempo da JVM) para cada elemento, preservando-se a causalidade dos eventos. Os valores entre parênteses são o desvio padrão de cada medida e entre colchetes a percentagem em relação ao tempo total.

Tabela 1. Medidas do caminho crítico para um conector *nulo*

cliente	proxy	conector nulo	servidor
invc. proxy 0,6 (0.49)[4]			
	cria objetos 2.9 (0,54)[18]		
	reflexão 3,6 (0,8) [22]		
	empac. args 4 (4,7) [25]		
	invc. conector 0,6 (0,49)[4]		
		desempac. args 1,4 (0,49)[9]	
		reflexão 0,9 (0,54)[6]	
		invc. dinâmica 1,3 (0,46)[8]	
			retorno 0 (0)
		retorno 0 (0)	
	retorno 0,8 (0,4)[5]		
Total 0,6 [4%]	11,9 [74 %]	3,6 [22 %]	0 [0%]

4.1. Análise

A partir das medidas da tabela 1, algumas observações puderam ser feitas:

- uma vez que a configuração da aplicação é estabelecida, o suporte de R-RIO não participa diretamente do fluxo de informações da aplicação, como era esperado. Isto só ocorre quando as APIs de Reflexão Arquitetural ou Configuração são utilizadas;
- acesso aos repositórios de meta-nível (via APIs de Reflexão Arquitetural) são responsáveis por ~30% do custo no *proxy* (22% do total) e ~25 % do custo no conector (6% do total);

- procedimentos de invocação dinâmica e chaveamento de contexto custam em torno de 1ms (16 % do total);
- cada criação de uma nova instância de objeto, necessária em vários pontos do caminho crítico (*threads*, objetos de retorno, etc.), custa em torno de 1 ms. No total isto representa 24 % do custo do *Proxy* (18% do total);
- procedimentos de empacotamento dos argumentos consomem em torno de 4 ms, com um desvio padrão grande, o que indica grande variação neste tempo (34 % do total);.
- tempo utilizado para invocações de métodos não é muito representativo em relação ao tempo total. Assim, verifica-se que a indireção do conector não impõe restrições (o que já havia sido observado na primeira série de medidas);
- tempo gasto no cliente e no servidor também pode ser desprezado.

Em consulta a outros trabalhos em desempenho envolvendo Java [Matjas 2000, por exemplo] e à lista de discussão sobre RMI (http://archives.java.sun.com/archives/ rmi-users.html) foi possível entender a razão do consumo de tempo em alguns casos. Por exemplo, em Java é possível utilizar-se duas classes para representar uma cadeia de caracteres: *StringBuffer* ou *java.lang.String*. A segunda alternativa, utilizada na aplicação (e normalmente recomendada nos tutoriais sobre Java), apresenta pior desempenho embora seja mais completa. A utilização da primeira alternativa poderia melhorar o desempenho dos conectores.

Observou-se na primeira série de medidas que os procedimentos de serialização / desserialização de dados são mais custosos em relação aos outros, e apresentam grande variação. O mecanismo de serialização de objetos em Java tem internamente o suporte de classes (por exemplo, *ObjectInputStream e ObjetcOutputStream*) que possuem rotinas para manipulação de *buffers* "com toda a pompa". Estes *buffers* podem ser ajustados e alinhados constantemente sem o controle da aplicação. Isto acrescenta mais um argumento na justificativa da variação dos tempos medidos na seção 3.1, na utilização da classe *ArgumentHolder* - Tabela 1 (que se utiliza destas classes para realizar o empacotamento dos argumentos no *Proxy*), e nos conectores. Estes fatos também explicam o desempenho do mecanismo de *sockets* em relação ao RMI no caso de nossa implementação: as rotinas de serialização para o procedimento de *marshalling*, e a multiplexação / demultiplexação das interações são otimizadas nos *stubs* de RMI, cujo código é gerado por um compilador especial, e não recebem tratamento especial no caso de nosso código com *sockets*.

Embora existam alguns procedimentos que estejam consumindo mais tempo que outros, aparentemente não existe um único gargalo no desempenho. Assim, além da serialização - que não foi medida nesta fase, a criação de objetos (18%), o acesso ao repositório de meta-nível (28%), e a compactação (25%) / descompactação (9%) de argumentos, em conjunto, representam os procedimentos onde a maior parte do tempo de uma transação é consumido (80%).

5. Otimização

Com as informações obtidas na primeira série de medidas foram feitas algumas otimizações concentradas na diminuição de criação de objetos. Na rotina de geração automática do *Proxy* observamos que uma pequena reestruturação permitiria reduzir o número de criação de instâncias de objetos, reaproveitando objetos criados na execução do construtor. O código das classes conectores também foi reorganizado para manipular melhor as *threads* criadas inicialmente, evitando-se a criação constante de *threads* (que são ainda mais custosas em relação a objetos comuns).

Não investimos, entretanto, nas rotinas de compactação e serialização, nem na restrição ao uso da classe *String*. Isto implicaria em restringir as próprias aplicações e em muitas alterações no código do Gerente de Configuração e na estrutura dos conectores. Assim sendo, algumas rotinas não puderam ser dispensadas, como por exemplo o uso do *ArgumentHolder*, a própria serialização de objetos e a invocação dinâmica.

Um outro ponto que poderia ser otimizado é a implementação da API de Reflexão Arquitetural, para tornar o acesso às informações de meta-nível mais eficiente. O Gerente de Configuração foi concebido prevendo-se problemas de escalabilidade. Existe um "repositório global" onde todas as informações sobre a arquitetura estão armazenadas e "repositórios locais" onde ficam armazenadas informações da parte da arquitetura localizada no mesmo nó. O repositório global é acessado pela rede, mas pode ser replicado para otimizar o acesso. Uma outra alternativa, como a utilização do *pattern Component Configurator* em [Kon 2000], seria descentralizar a localização de informações de meta-nível fazendo que cada componente seja responsável por armazenar e disponibilizar as mesmas. Em nossa proposta preferimos não impor este *overhead* aos componentes e procuramos escalabilidade "*cacheando*" e replicando as informações. Assim, optamos por tratar outras otimizações neste ponto na continuação deste trabalho.

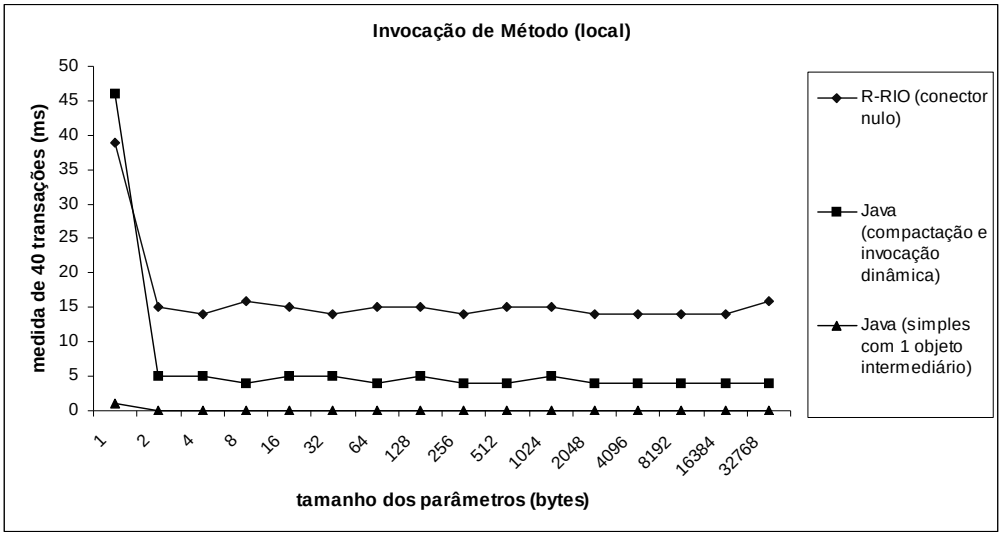

Figura 7. Medidas de desempenho para invocação de método (2a fase)

As medidas apresentadas nas próximas seções são do mesmo tipo da primeira série de medidas, agora com as otimizações. Foi possível diminuir a diferença do desempenho dos conectores em relação ao uso isolado dos vários mecanismos.

5.1. Invocação de método local (MI)

Na figura 7 é apresentado o gráfico em que se compara a aplicação com o conector nulo em R-RIO e a aplicação sem o suporte R-RIO. Observa-se que, após a otimização, o desempenho da aplicação em R-RIO é comparável a de Java puro, ficando em média abaixo de 1 ms (no gráfico são plotados os tempos para 40 transações).

.

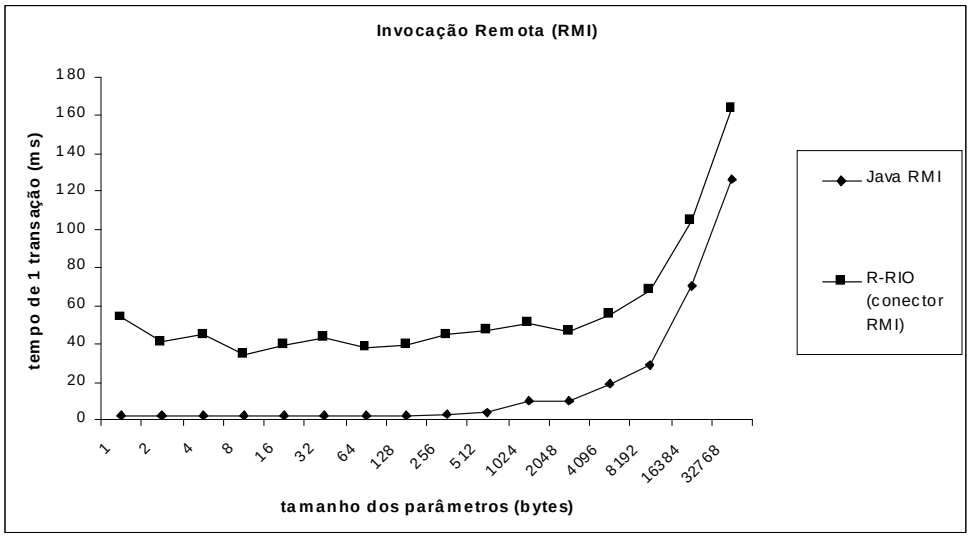

Figura 8. Medidas de desempenho para RMI (2a fase)

5.2. RMI

Para o caso de conectores RMI não houve diferença significativa pois, ao contrário dos outros conectores de comunicação (incluindo-se *sockets*, CORBA, *multicast*, etc.) a serialização dos argumentos e multiplexação de mensagens é feita nos *stubs*. Tanto a versão R-RIO, como a versão Java da aplicação utilizaram o mesmo conjunto classes e *stubs* para transmitir / receber mensagens. Assim sendo, apenas as otimizações aplicadas ao *Proxy* tiveram efeito para o caso do conector RMI.

5.3. *Socket*

Além dos pontos identificados na seção 2 para todos os conectores, o código do conector de *sockets* ainda tinha

alguns pontos para serem otimizados. No lado do cliente, uma nova instância do conector estava sendo criada a cada transação. Além disso, a cada transação era estabelecida uma nova conexão, uma vez que o protocolo TCP é utilizado. Estes problemas foram corrigidos. Não foram feitas alterações no lado do servidor.

O gráfico da figura 9 apresenta uma comparação entre aplicações *socket* com R-RIO e sem o suporte de R-RIO após as otimizações. Repetimos no mesmo gráfico as medidas da aplicação sem as otimizações (valores no eixo Y do lado direito do gráfico).

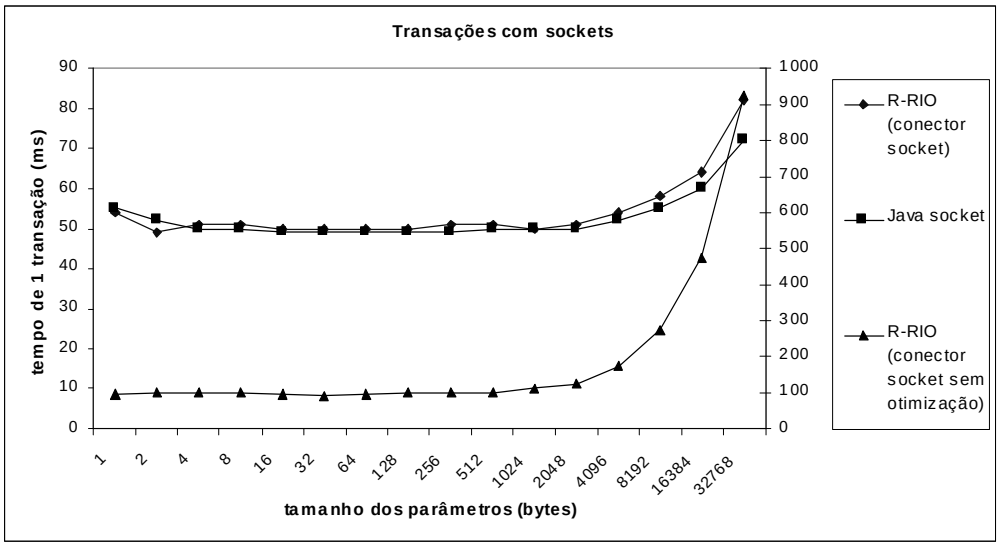

Figura 9. Medidas de desempenho para *sockets* (2a fase)

Observa-se também que após as otimizações no conector *socket*, os valores comparados com o conector RMI, são de ordem equivalente.

Os problemas remanescentes no desempenho dos conectores *socket* se encontram nas rotinas de serialização e desserialização. Estas rotinas, como já ressaltamos, freqüentemente precisam criar objetos, copiar *buffers* e ativar métodos custosos, sem possibilidade de controle pelo programador. No caso dos conectores de R-RIO isto é necessário para permitir que o mesmo código contendo um mecanismo de comunicação possa se adaptar dinamicamente ao contexto de qualquer interface. Em princípio, está se pagando um preço, em desempenho, pela flexibilidade oferecida.

6. Trabalhos Relacionados

Durante a realização da avaliação procuramos examinar trabalhos relacionados para averiguar se estávamos utilizando procedimentos adequados e comparar os resultados obtidos. Outra preocupação era a de termos uma implementação muito aquém da média em relação a desempenho. Ficamos menos intranqüilos ao comparar os resultados destes trabalhos com os nossos.

Em [Abdul-Fatah 2002] são examinados detalhadamente os aspectos de desempenho de aplicações cliente-servidor baseadas em CORBA. Além do efeito do tamanho dos dados sendo transportados, foram considerados aspectos como a carga do servidor.

Em um nível mais abrangente [Matjas 2000] fez vários estudos relacionados ao desempenho dos mecanismos de transporte da interação entre objetos usando Java RMI (stubs, skeletons, IIOP, etc.). Durante estes estudos foram propostas algumas otimizações na implementação destes mecanismos que posteriormente poderiam ser adotadas pela IBM em sua implementação de IIOP/RMI.

Em [Hirano 1998] vários *middlewares* para objetos distribuídos foram comparados quanto a (i) criação e conexão de objetos remotos, (ii) invocações a métodos em objetos remotos e (iii) transferência de vetores de objetos (dados). Um dos resultados destas avaliações foi a verificação da necessidade de se otimizar a criação e serialização de objetos através de um sistema de *cache*. Em nosso trabalho chegamos a conclusão semelhante (seção 4).

Um outro tipo de trabalho, mais abstrato, [Carando 1996] (desenvolvido na AT&T) propõe um conjunto de critérios para comparação de ORBs com o objetivo de ajudar a seleção de produtos para sistemas corporativos. Entre os critérios estão a independência de linguagem de programação, escalabilidade, tolerância a falhas e, claro, desempenho.

Por último, uma boa fonte de informações

sobre desempenho da diversos *middlewares* comerciais seria [Cutter 2002]. Os resumos dos documentos disponíveis no sítio apontam para avaliações de desempenho quantitativas detalhadas e abrangentes, e análises qualitativas realizadas por *experts* da área. Entretanto este é um serviço comercial e não foi possível obter acesso a estas informações.

7. Conclusão

Apresentamos neste trabalho uma avaliação do custo computacional dos conectores de *software* em R-RIO. Nesta avaliação, medimos o desempenho da interação entre dois módulos, intermediada por um conector. No caminho crítico de um conector existem dois níveis de indireção, onde são necessárias (i) consultas ao repositório de informações de meta-nível, (ii) serialização-empacotamento e desserialização-desempacotamento de informações e (iii) invocações dinâmicas de métodos. Observou-se que estas rotinas são responsáveis por mais de 90% do custo computacional da uma interação. Tais resultados podem ser aplicados a outros exemplos de *middlewares* adaptáveis.

O custo direto dos procedimentos de reconfiguração em R-RIO não foi avaliado. Em medidas realizadas em sistemas similares como, por exemplo, o sistema 2K [Kon 2000] constatou-se que os procedimentos de reconfiguração não comprometeram o desempenho esperado da aplicação. 2K utiliza um esquema de *wrapping*, para adicionar a capacidade de reconfiguração de componentes, e utiliza o serviço de nomes de CORBA, para manter as referências dos componentes. Embora R-RIO difira conceitualmente de 2K em vários aspectos, o esforço computacional envolvido em uma reconfiguração típica pode ser considerado equivalente nos dois sistemas. Por exemplo, as consultas ao serviço de nomes de CORBA (um repositório de meta-nível) e a execução de métodos de reconfiguração pelo *wrapper* do componente têm o seu equivalente em R-RIO: as APIs de Reflexão Arquitetural e Configuração.

Observamos que os mecanismos básicos para a construção de conectores (a interceptação / manipulação do fluxo de informações das interações e a invocação dinâmica) são também utilizados individualmente em outros *frameworks*. Por exemplo, CORBA 2.0 [OMG 1996] oferece um mecanismo, chamado *interceptor*, que permite a interceptação do fluxo em pontos pré-determinados de uma invocação a método. Este mecanismo possui uma API através da qual pode-se inspecionar e alterar as informações de uma invocação a método. CORBA também possui uma API para o uso de invocações dinâmicas (*Dynamic Invocation Interface* – DII) para métodos desconhecidos em tempo de compilação. Estes dois mecanismos são comparáveis aos utilizados internamente nos conectores em R-RIO.

Para avaliarem-se outros aspectos dos conectores de software seriam necessárias outras medidas. Por exemplo, não foram exploradas as características de concorrência dos conectores intermediando a interação entre vários clientes e vários servidores. Estas medidas trariam informações de escalabilidade dos conectores, e do uso da API de Reflexão Arquitetural, provavelmente apontando para novos caminhos de otimização.

Os resultados obtidos nas medidas e as otimizações implementadas foram interessantes. Concluímos que como norma, o uso de *cache* (ou replicação) como forma de reutilização de objetos e a otimização da serialização são recomendados.

No contexto de nossa proposta é possível utilizarmos o *framework* R-RIO para prototipar uma aplicação, realizar testes, e otimizar a implementação final. Entretanto, na maioria dos casos, isso implicaria em perda de flexibilidade (por exemplo, substituindo as interceptações, consultas a informações de meta-nível e invocações dinâmicas por ligações estáticas). Em resumo, para qualquer mecanismo utilizado na construção de conectores sempre haverá uma penalidade em desempenho pelo uso de mecanismos de meta-nível. É o preço que se paga pela flexibilidade.

Referências

Abdul-Fatah, I e Majumdar, S, "Performance of CORBA-Based Client-Server Architectures", IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 2, pp.111-127, Fevereiro, 2002.

Agha, G. A., "Adaptive Middleware", Communications of the ACM, Vol. 45, No. 6, pp. 31-32, Junho, 2002.

Birrell, A., Nelson, B. J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, Fevereiro, 1984.

Carando, P, "Toward the Development of an Object Request Broker Evaluation Instrument", OOPSALA-96 Workshop on Objects in Large, Distributed and Persistent Systems, 1996.

Carvalho, S. T., *Um design pattern para a configuração de arquiteturas de software*, Dissertação de M.Sc., Instituto de Computação/UFF, Março, 2001.

Coulson, G., "What is Reflective *Middleware*?", *IEEE Distributed Systems On-Line*, IEEE, 2000. http://computer.org/dsonline/middleware/RM.htm

Crane, S., Maggee, J., Pryce, N., "Design Patterns for Binding in Distributed Systems", OOPSLA'95, *Workshop on Design Patterns for Concurrent and Distributed Object-Oriented Systems*, Outubro, 1995.

Cross, J. K. e Schmidt, D. C., "Quality Connector", OOPSLA Workshop on Patterns in Distributed Real-Time and Embedded Systems, Seattle, Washington, USA, Novembro, 2002.

Cutter Consortium, "8 Reports on Enterprise Middleware Technologies", Dezembro, 2002. http://www.cutter.com

Gama, E., Helm, R. Johnson, R. e Vilissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, EUA, 1995.

Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Sun Microsystems, Addison Wesley Publishing, EUA, 1996.

Hirano. S, Yasu, Y. e Igarashi, H., "Performance Evaluation of Popular Distributed Object Technologies for Java", ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford University, Palo Alto, California, 1998.

Kon, F., "Automatic Configuration of Component-Based Distributed Systems", Tese de D.Sc., Department of Computer Science, UIUC, EUA, Maio, 2000.

Kon, F., Costa, F., Blair, G. e Campbell, R. H., "The Case for Reflective Middleware", Communications of the ACM, Vol. 45, No. 6, pp. 33-42, Junho, 2002.

Laddaga, R., Veitch, J., "Dynamic Object Technology", Communications of the ACM, Vol. 40, No. 5, pp. 37-38, Maio, 1997.

Lobosco, M., *R-RIO: Um Ambiente para Suporte à Construção e Evolução de Sistemas*, Dissertação de M.Sc., Instituto de Computação/UFF, Março, 1999.

Matjaz, B. Juric., Rozman, Ivan. e Nash, Simon., "Java 2 Distributed Object Middleware Performance Analysis and Optimization", ACM SIGPLAN Notices, V. 35, No. 8, pp. 31-40, Agosto, 2000.

Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0, Julho, 1996.

Schmidt, D. C., Levine, D., Mungee, S., "The Design of the TAO Real-Time Object Request Broker", *Computer Communications*, Elsevier Science, Vol. 21, No. 4, Abril, 1998.

Schmidt, D. C., Cleeland, C., "Applying Patterns to Develop Extensible ORB *Middleware*", *IEEE Communications Mag. Special Issue on Design Patterns*, 1999.

Schmidt, D. C., Stal, M., Ronhert, H., Buschmann, F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, John Wiley & Sons, New York, 2000.

SHAW, M., GARLAN, D., *Software architecture: perspectives on an emerging discipline*, Prentice-Hall Inc., EUA, 1996.

Sun Microsystems, *Java™ Remote Method Invocation Specification*, Revision 1.50, JDK 1.2, Outubro, 1998.