# Avaliação do Desempenho de Alternativas Otimizadas para Comunicação em Java

Sidney Barbosa Ansaloni
*Universidade Federal Fluminense*
*ansaloni@ic.uff.br*

Orlando Loques
*Universidade Federal Fluminense*
*loques@ic.uff.br*

## Resumo

*Tem sido observado um crescente interesse na otimização de vários mecanismos suportados pela plataforma Java visando amenizar a penalização introduzida pelo fato da linguagem Java ser interpretada.*

*O presente trabalho realiza a avaliação do desempenho de três propostas para diminuir os custos associados ao mecanismo de comunicação padrão do Java. São avaliadas as propostas KaRMI, Manta e JPVM tomando como referência a implementação Java RMI padrão. Os testes realizados são baseados no tempo de resposta de mensagens e no tempo de execução de um algoritmo distribuído baseado em troca de mensagens.*

## 1. Introdução

Devido à crescente demanda por aplicações distribuídas, a linguagem de programação Java [1, 2] oferece como suporte à distribuição um modelo de objetos distribuídos que permite a um objeto, em uma máquina virtual Java (JVM) específica, invocar métodos de objetos em outras máquinas virtuais. Os objetos que podem ter os métodos invocados a partir de objetos que residam em JVMs distintas são denominados objetos remotos e são descritos através de suas interfaces. A invocação a métodos remotos (RMI) [3] é o mecanismo que permite a um método ser invocado remotamente, porém, algumas questões sobre o seu desempenho têm sido discutidas.

O presente trabalho tem como objetivo avaliar algumas propostas relevantes que tentam melhorar a eficiência do mecanismo de invocação remota a métodos (RMI) oferecido pelo Java. O texto está organizado da seguinte forma: A seção 2 descreve resumidamente o mecanismo Java RMI. A seção 3 apresenta os ambientes KaRMI [4], Manta [5, 6] e JPVM [7]. A seção 4 apresenta a avaliação destes ambientes através dos experimentos realizados. E, finalizando, a seção 5 conclui o trabalho.

## 2. Java Remote Method Invocation (RMI)

A tecnologia *Remote Method Invocation* (RMI) define um modelo de objetos distribuídos que tenta manter a semântica do modelo de objetos padrão da plataforma Java. A arquitetura RMI é baseada no princípio que a interface e a implementação de objetos são conceitos diferentes. A interface de um objeto define os serviços (métodos e atributos) que o objeto pode oferecer e a implementação contém o código que define como os serviços definidos pela interface devem ser executados. O suporte RMI permite que o código que define a interface e o código que corresponde à implementação fiquem separados e sejam executados em JVM's diferentes, usando para isso duas classes distintas. A primeira classe define a implementação dos serviços oferecidos pelo objeto remoto instanciado no servidor. A segunda classe age como um *proxy* para o serviço remoto e é instanciada no lado cliente. Este objeto *proxy* intercepta as chamadas a métodos feitas pelo cliente e redireciona estas chamadas para o serviço remoto. Deste modo, um programa cliente faz invocações a métodos do objeto *proxy*, o RMI envia a chamada para a JVM remota, passando-a para a implementação. Os valores de retorno fornecidos pela implementação são enviados de volta para o *proxy* e então para o programa cliente.

Como discutido em [8], diversos autores identificam os aspectos de serialização, transporte sobre TCP/IP e mecanismos de cópia e criação de objetos do protocolo RMI como os mais relevantes quanto ao desempenho da tecnologia RMI, conforme apresentado a seguir.

### 2.1. Serialização de Objetos

Em chamadas a métodos locais, a semântica normal de passagem de parâmetros de tipos primitivos na plataforma Java é por valor e a passagem de objetos é por referência. Em chamadas a métodos remotos, a passagem de parâmetros é um pouco mais complexa. Quando um tipo de dados primitivo é passado como parâmetro para um método remoto, o mecanismo RMI faz passagem por

valor, como em chamadas locais. Porém, quando um objeto composto é passado para um método, o RMI também faz passagem por valor, ou seja, é feita uma cópia do objeto e esta cópia é enviada para o método remoto. Ao contrário de tipos primitivos, o envio de um objeto para uma JVM remota não é uma tarefa trivial devido à serialização, pela qual o RMI transforma objetos em um formato linear, que pode ser enviado pela rede, e que permite a reconstrução dos objetos na JVM remota. O protocolo de serialização da plataforma Java é baseado em introspecção dinâmica de tipos, ou seja, uma inspeção recursiva de tipos é feita até alcançar tipos que possuam serializadores definidos. Esta inspeção para determinar o tipo de cada objeto é feita através da capacidade de reflexão estrutural (embutida em Java) para produzir uma representação em bytes apropriada para cada tipo. Isto resulta em perda de desempenho, porque esta inspeção é feita em tempo de execução e é interpretada, como qualquer aplicação Java.

### 2.2. Transporte sobre TCP/IP

O Java RMI é implementado sobre *sockets* IP e a camada de transporte do RMI faz conexões utilizando o protocolo TCP, que provê serviço orientado a conexão que inclui garantia de entrega e controle de fluxo. Porém, aplicações de alto desempenho nem sempre estão preocupadas com a confiabilidade ao nível de transporte, permitindo a utilização de protocolos mais eficientes que o TCP em ambientes específicos de rede. Conjugado a isso, por ser baseado em *sockets* IP, o RMI não oferece suporte para sua utilização em outras arquiteturas de rede que não sejam baseadas em IP. Portanto, uma estratégia que também pode ser adotada para aumentar o desempenho do transporte RMI é a construção de suportes para tecnologias de redes de maior desempenho.

### 2.3. Mecanismos de Cópia e Criação de Objetos

O RMI é construído sobre uma hierarquia de subclasses das classes java.io.ObjectOutputStream e java.io.ObjectInputStream para organizar parâmetros e valores de retorno. A partir dos métodos annotateClass() e resolveClass(), o RMI recupera informações sobre as classes dos parâmetros. Esta característica é um tanto ineficiente porque a passagem de objetos é feita por valor. Neste caso é necessário fazer cópias sucessivas dos parâmetros para que eles passem por toda a hierarquia até atingirem o nível de transporte. Há também o *overhead* imposto pela criação de objetos de manipulação de fluxo e pelo gerenciamento de *buffers*.

## 3. Propostas Avaliadas

Existem diversas propostas que têm como objetivo melhorar o desempenho da comunicação do Java RMI ou inserir outros paradigmas de comunicação não oferecidos pela linguagem Java [9]. Para este estudo foram escolhidos os ambientes KaRMI, Manta e JPVM. Os ambientes Manta e KaRMI foram escolhidos por estarem presentes nas últimas conferências do Java Grande Forum e também por serem projetos de pesquisa que estão em plena atividade atualmente. O ambiente JPVM foi escolhido por fornecer uma alternativa de suporte à computação de alto desempenho, baseada em PVM (*Parallel Virtual Machine*), totalmente construído em Java, apresentando primitivas de comunicação funcionalmente compatíveis com as das outras propostas estudadas.

### 3.1. Manta

O Manta é um sistema Java para computação de alto desempenho baseado em um compilador nativo que traduz do Java diretamente para código executável. O Manta substitui muito do processamento em tempo de execução do protocolo RMI padrão por análises em tempo de compilação. Quanto à serialização de objetos, o compilador nativo gera código serializado e rotinas para serializações especializadas, baseadas no fato de que as informações dos tipos dos parâmetros em tempo de compilação são suficientes para identificar o serializador correto a ser utilizado. Os objetos a serem enviados são serializados diretamente em um *buffer* do sistema de comunicação, eliminando o *overhead* imposto pelas cópias sucessivas entre vários objetos da hierarquia RMI padrão. O protocolo de serialização do Manta também provê algumas otimizações adicionais; por exemplo, no caso de um *array* de tipos primitivos é feita uma cópia direta da memória para o *buffer* de mensagens, evitando assim a travessia de todo o *array* para a leitura de dados. Além de compilar o código Java diretamente em código binário executável, toda a estrutura do protocolo RMI oferecida pelo Manta é compilada a partir de código C, ou seja, o Manta não utiliza interpretação de código em nenhuma etapa da execução das aplicações.

Enquanto o RMI padrão se restringe ao uso do TCP como protocolo de transporte, o Manta oferece uma variedade de implementações eficientes para diversas tecnologias de rede através da biblioteca de comunicação Panda. Por exemplo, em uma rede Myrinet, a biblioteca Panda usa o sistema de comunicação de alto desempenho LFC (*Link-level Flow Control*); em redes Ethernet, a biblioteca Panda é implementada sobre o protocolo UDP.

### 3.2. KaRMI

A abordagem do KaRMI é baseada na substituição dos mecanismos de serialização e de invocação remota da plataforma Java por bibliotecas que implementam estes serviços de forma mais eficiente. Em relação ao Manta, o KaRMI se apresenta como uma proposta mais compatível com a plataforma Java, uma vez que não faz uso de código nativo. Assim como o Java RMI padrão, o KaRMI é implementado em três camadas: camada *Stub*, camada de referência e camada de transporte. Na implementação padrão do Java RMI, estas camadas não têm uma interface claramente definida, causando falhas como a exposição da implementação de *sockets* no nível de aplicação e fazendo com que objetos sejam acessados através de portas de número específico, o que obriga a utilização de *sockets* no nível de transporte. No KaRMI estas camadas foram reestruturadas de forma que as interfaces entre elas ficaram mais definidas (em relação à original), facilitando a integração de implementações alternativas das camadas de referência e transporte. Por exemplo, no KaRMI existe a possibilidade de utilização de múltiplas interfaces de rede, simultaneamente; para cada hardware de rede em um determinado computador, um objeto *technology* associado é criado na inicialização, permitindo que a camada de referência utilize o melhor objeto *technology* e, por conseqüência, o hardware de rede mais adequado para alcançar um dado objeto. Outro benefício obtido com a reestruturação das camadas KaRMI é que uma invocação a método remoto requer somente duas invocações adicionais de métodos nas interfaces entre as camadas e não requer a criação de objetos temporários como exigido no protocolo RMI padrão.

O KaRMI pode utilizar uma biblioteca de serialização desenvolvida em Java que implementa uma descrição de tipos mais eficiente, além de mecanismos de persistência de informações sobre tipos e serviços de bufferização especializados. Esta biblioteca, conhecida como *uka.transport* [10], é desenvolvida pela equipe de desenvolvimento da KaRMI e também foi utilizada em nossos experimentos.

### 3.3. JPVM

A biblioteca JPVM fornece um sistema para trocas de mensagens explícitas baseado em programação paralela em Java. A JPVM suporta uma interface similar às interfaces oferecidas pelas implementações de PVM (*Parallel Virtual Machine*) [11] em C e Fortran, mas com algumas modificações na sintaxe e semântica para se adequar ao estilo de programação em Java. Embora não se assemelhe ao modelo de objetos distribuídos apresentado pelo Java RMI, o JPVM implementa um ambiente baseado no modelo de troca de mensagens que pode ser considerado como uma alternativa ao Java RMI.

Apesar das limitações de desempenho, o uso do Java como linguagem de implementação do JPVM provê algumas vantagens, principalmente no que diz respeito à habilidade de integrar conceitos de orientação a objetos e serviços como *threads* ao conhecido modelo de programação PVM, através de uma interface portável sobre um grande número de plataformas de hardware e software. A implementação de troca de mensagens do JPVM é baseada na comunicação sobre *sockets* TCP, onde são estabelecidas conexões diretas entre as tarefas, deixando de utilizar *daemons* de roteamento de mensagens como nas implementações padrões de PVM.

## 4. Avaliação do Desempenho das Propostas Estudadas

Em [12], é proposto um conjunto de aplicações para a avaliação de desempenho (*benchmarks*) de implementações de máquinas virtuais Java. Estas aplicações compõem a iniciativa do Java Grande Forum de desenvolver uma base de referência para a avaliação de diferentes ambientes de execução Java de alto desempenho. São definidos *benchmarks* para a avaliação de uma grande variedade de aplicações Java, como aplicações seqüenciais, *multithreaded* e distribuídas. Estes *benchmarks* são classificados em três tipos e para o escopo de aplicações distribuídas são assim organizados:

- *Microbenchmarks*: medem o desempenho de invocações a métodos remotos (operações básicas).
- *Kernels*: testam o comportamento dos ambientes avaliados sobre certas situações. São compostos por algoritmos clássicos que possuem um grau de complexidade relevante. As aplicações distribuídas do tipo *kernel* usam mais invocações a métodos do que é adequado para resolver o problema, ou seja, uma implementação seqüencial poderia ser mais rápida. Mas por outro lado, permitem a avaliação de padrões de comunicação comuns e do desempenho levando em consideração atividades de escalonamento e sincronização.
- Aplicações de Grande Escala: são representativas no que diz respeito à solução de problemas do mundo real, porém não permitem avaliar os efeitos que fazem o desempenho de um ambiente específico ser maior ou menor do que o de outro.

Dadas estas informações, a análise aqui proposta foi baseada na medição do tempo médio para a realização de invocações a métodos remotos, que representam os *microbenchmarks* e na medição do tempo de execução do algoritmo SOR (*Successive Over-Relaxation*), representando os testes do tipo *kernel*.

Para medir o tempo médio na realização de invocações

a métodos remotos foram implementados métodos que têm como parâmetros tipos primitivos, tipos complexos e *arrays* destes tipos, além de um método sem parâmetros.

O SOR é um algoritmo iterativo para resolver equações de Laplace em uma matriz de duas dimensões. Em cada iteração, o valor de cada ponto da matriz é atualizado com base nos valores dos quatro pontos vizinhos na matriz. A implementação utilizada é baseada no algoritmo *red/black* [4]. As cores vermelha e preta são associadas aos pontos da matriz de tal forma que nenhum ponto da matriz tenha vizinhos da mesma cor, tornando a matriz semelhante a um tabuleiro de xadrez. Durante a fase vermelha, somente os pontos vermelhos são atualizados e durante a fase preta, somente os pontos pretos são atualizados. Entre as fases, os valores nas bordas da matriz são trocados. Em sua versão distribuída, as linhas da matriz são distribuídas entre os processadores disponíveis.

O código fonte Java foi compilado e executado na plataforma Java 2 SDK 1.4.0 e todos os computadores utilizados têm processador AMD Athlon 700 MHz e são equipados com 128MB, utilizando o Linux como sistema operacional (distribuição *Red Hat Linux* 7.1 com *kernel* 2.4.2-2). Estes computadores são conectados através de uma rede *Fast-Ethernet* (100Mbps) livre de outros tráfegos. Utilizamos as seguintes versões dos mecanismos de invocação remota:

- Java RMI (Java 2 SDK 1.4.0)
- Manta v0.1
- KaRMI v1.06-pre
- JPVM v0.2.1

Para cada experimento realizado, foram obtidas 100 amostras e calculado o valor médio. Utilizamos a dispersão relativa, que expressa o desvio padrão como porcentagem do valor médio, para medir a variabilidade dos resultados. Deste modo, quanto menor o valor da dispersão relativa, maior é o grau de confiança dos valores médios apresentados.

### 4.1. Transferência de Tipos Primitivos

O tempo de resposta é o principal fator de desempenho em sistemas de comunicação baseados em troca de mensagens. Os dados da Tabela 01 ilustram os resultados dos experimentos realizados para medir o tempo médio para a realização de invocações a métodos remotos com tipos primitivos como argumento. A maior dispersão relativa deste experimento foi de 0,52% apresentada pelo RMI padrão e pelo JPVM para a transferência do tipo *double*. Como esperado, o Manta obteve o melhor resultado devido a dois principais fatores. Primeiro, por gerar código nativo compilado ao invés de interpretar pseudo-códigos como os demais ambientes. O outro fator foi a utilização da biblioteca de comunicação Panda, implementada sobre o protocolo UDP. Os resultados do KaRMI são animadores. Neste experimento, o tempo de resposta apresentado pelo KaRMI corresponde a um terço do tempo de resposta apresentado pela implementação padrão do Java RMI. Deve-se lembrar que este ambiente não possui nenhuma otimização através de códigos nativos.

Tabela 1. Tempo de resposta para métodos com tipos primitivos como argumento (µs)

| | void | byte | char | int | float | double |
|---|---|---|---|---|---|---|
| RMI | 528 | 576 | 579 | 579 | 583 | 592 |
| KaRMI | 177 | 184 | 184 | 183 | 186 | 189 |
| Manta | 57 | 60 | 61 | 61 | 60 | 61 |
| JPVM | 588 | 1691 | 1774 | 1765 | 1789 | 1809 |

A biblioteca JPVM, que não apresenta nenhuma otimização em relação à implementação padrão do RMI, apresenta desempenho similar ao Java RMI na chamada nula e um desempenho muito pior no envio de tipos primitivos. A justificativa para esse comportamento deve-se ao fato de que na biblioteca JPVM, todo valor a ser passado como argumento para um método remoto deve ser inserido em um *buffer*. Este *buffer* é um objeto Java da classe jpvmBuffer e é este objeto contendo os argumentos que é efetivamente enviado. Portanto, o envio de um simples byte em JPVM corresponde ao envio de um objeto, que insere os custos da serialização.

### 4.2. Transferência de Objetos

A Tabela 02 apresenta os resultados para os métodos que recebem objetos como argumento. O primeiro método testado recebeu um objeto do tipo String com o conteúdo vazio e o outro método recebeu um objeto constituído de 32 atributos do tipo int (inteiro de 32 bits do Java), contendo assim 1kB de informação de usuário. Assim como no experimento anterior, a variabilidade dos resultados se manteve baixa, sendo a maior dispersão relativa observada de 0,45%.

Tabela 2. Tempo de Resposta para métodos com String e Objeto (µs)

| | String | Objeto |
|---|---|---|
| RMI | 648 | 1181 |
| KaRMI | 313 | 745/291* |
| Manta | 75 | 86 |
| JPVM | 1765 | - |

* com uso de *uka.transport*

Nesta avaliação, os ambientes mantiveram a ordem apresentada nos testes com tipos primitivos, porém um

ponto importante que deve ser destacado neste experimento diz respeito a serialização. Seguindo o modelo PVM original, a biblioteca JPVM não possui suporte para a transferência de objetos de usuários, o objeto jpvmBuffer que é enviado pelo JPVM só tem suporte a tipos primitivos. É possível criar o suporte para objetos para este *buffer* utilizando recursos da linguagem Java, mas não era objetivo deste estudo alterar os ambientes avaliados.

O Manta novamente se destaca como o mais eficiente devido à sua geração de código nativo compilado e a utilização da biblioteca de comunicação Panda. Porém vale ressaltar que o seu suporte à serialização também é gerado em tempo de compilação, portanto sua posição como o mais eficiente não é nenhuma surpresa. O KaRMI utilizando o serialização padrão do Java obtém uma performance melhor do que o Java RMI padrão, o que confirma a eficiência do seu protocolo. Porém, ao utilizar o mecanismo de serialização *uka.transport*, o tempo de resposta do KaRMI cai para 39,06% do tempo de resposta dele mesmo utilizando a serialização padrão, confirmando que o protocolo de serialização da plataforma Java é uma grande fonte de *overhead* para a invocação de métodos remotos.

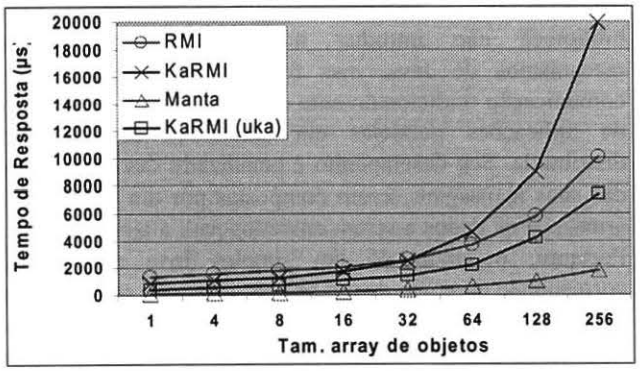


Figura 01. Tempo de resposta para *arrays* de objetos (µs)

Outro experimento feito foi a transferência de *arrays* de objetos, onde foram invocados métodos remotos cujos argumentos eram *arrays* de objetos constituídos de 32 inteiros (como o utilizado no experimento anterior) com tamanho definido em tempo de execução. Como a biblioteca JPVM não possui suporte à transferência de objetos, ela não participou deste experimento. Os resultados (Figura 01) apresentam um comportamento inesperado do KaRMI. Em sua utilização sem a tecnologia de serialização *uka.transport*, o tempo de resposta do KaRMI rapidamente ultrapassa o tempo de resposta da implementação padrão do Java RMI, retratando um gerenciamento de *buffers* ineficiente. A utilização da *uka.transport*, que além de outras vantagens, provê um gerenciamento eficiente de *buffers*, ameniza este sintoma, porém, nota-se uma tendência de aproximação ao tempo de resposta do RMI convencional à medida que o tamanho do *array* aumenta. O KaRMI apresentou a maior dispersão relativa neste experimento, 1,87%, porém este valor ainda é considerado baixo em termos de variabilidade.

## 4.3. Transferência de Arrays de Tipos Numéricos

Geralmente, aplicações distribuídas de alto desempenho requerem uma alta taxa de transferência de dados numéricos. Nestas aplicações, uma grande quantidade de dados é manipulada pela rede através da transferência de *arrays* de tipos numéricos. Geralmente, algoritmos numéricos de alta complexidade são distribuídos ou paralelizados para se obter um tempo de execução menor do que o mesmo algoritmo de forma seqüencial. Portanto, o mecanismo de mensagens nestas aplicações deve introduzir o menor *overhead* possível. Para a avaliação da taxa de transferência de massa de dados numérica, foram utilizados quatro métodos, cada um com um *array* de um dos seguintes tipos: byte, int, float e double. Os tamanhos dos *arrays* foram definidos em tempo de execução.

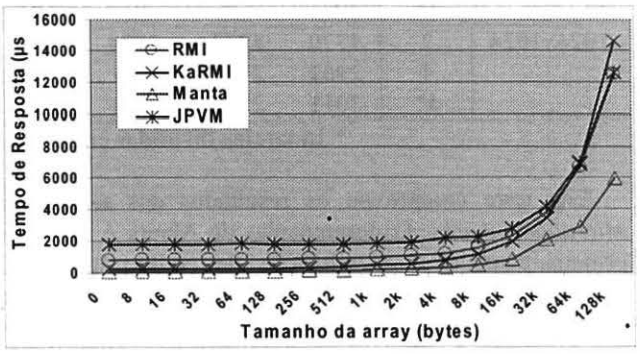


Figura 02. Tempo de Resposta: *Arrays* de bytes

A Figura 02 apresenta o tempo médio obtido para transferência de *arrays* de bytes. O comportamento para *arrays* de outros tipos é análogo, sendo a maior dispersão relativa apresentada neste experimento de 2,13%. Nesta figura, pode-se notar a saturação da rede na região onde os *arrays* são compostos por 32768 bytes (32kB). A partir de então, o comportamento dos ambientes torna-se um pouco instável. Em particular, o KaRMI apresenta um aumento no tempo de resposta superior aos demais ambientes. Como identificado no experimento anterior, o KaRMI não reage bem à bufferização sem a presença da serialização *uka.transport*, o que é comprovado por este comportamento associado com a saturação da rede.

### 4.4. Tempo de Execução do Algoritmo SOR

Em adição aos testes básicos, foi realizada uma avaliação do desempenho total, baseada na aplicação distribuída *Successive Over-Relaxation* (SOR). O SOR é um algoritmo iterativo para resolver equações de Laplace em uma matriz de duas dimensões. Para este experimento, foram utilizadas matrizes 64x64, 256x256 e 1024x1024 e o tempo médio por iteração é apresentado na Tabela 03. A dispersão relativa apresentada para as medições deste experimento foi consideravelmente maior do que a observada nos experimentos anteriores, chegando a 10,6 %. Este comportamento é justificado porque, neste experimento, as atividades de escalonamento e sincronização têm uma maior influência nos resultados.

Tabela 03. Tempo médio de execução do algoritmo SOR (ms)

| Matriz | Nº de *hosts* | RMI | KaRMI | Manta | JPVM |
|---|---|---|---|---|---|
| 64x64 | 2 | 15 | 15 | 5 | 12 |
| | 4 | 187 | 54 | 9 | 116 |
| | 4* | 775 | 212 | 24 | 348 |
| 256x256 | 2 | 283 | 291 | 100 | 116 |
| | 4 | 341 | 237 | 61 | 307 |
| | 4* | 1022 | 425 | 85 | 404 |
| 1024x1024 | 2 | 4770 | 4922 | 1704 | 2517 |
| | 4 | 2961 | 2725 | 726 | 732 |
| | 4* | 3011 | 2585 | 575 | 614 |

* 16 tarefas divididas em 4 *hosts*

Este teste comprovou os resultados dos testes tipo *Microbenchmarks*. O desempenho do Manta é atribuído principalmente à sua utilização de código nativo. O KaRMI apresentou um desempenho menor do que a implementação padrão do RMI para a execução em dois *hosts*, devido ao fato que nesta situação o volume de dados transmitidos por tarefa é maior, indicando o problema de gerenciamento de *buffers* identificado nos *Microbenchmarks*. O *uka.transport* não foi utilizado porque os dados não eram compostos de objetos serializáveis. O JPVM se apresenta como uma boa alternativa à medida que as dimensões das matrizes aumentam, este comportamento pode ser justificado pela implementação otimizada do algoritmo SOR em JPVM, uma vez que este modelo é mais adequado para aplicações paralelas.

## 5. Conclusões

Quanto às tecnologias apresentadas, o Manta se destaca por seu desempenho. Os experimentos apontam que o Manta é substancialmente mais eficiente do que os demais ambientes que participaram do estudo. Por exemplo, enquanto uma invocação nula do Sun Java RMI leva 528 µs para ser completada, o tempo de resposta da mesma invocação no Manta é de apenas 57 µs. Porém dois aspectos que devem ser considerados sobre o Manta são: (i) não é totalmente compatível com outras JVMs; e (ii) a compilação e execução de aplicações não é tão trivial como para os outros ambientes estudados.

O KaRMI destaca-se por ser compatível com a plataforma Java, uma vez que não faz uso de código nativo e por ter como idéia central a substituição de mecanismos da plataforma Java por bibliotecas que implementam otimizações de desempenho. Sua utilização mostrou sua eficiência para a transferência de baixas quantidades de dados, mas apresenta problemas para gerenciamento de *buffers* para grandes quantidades. Entretanto, ao substituir o mecanismo padrão de serialização pelo mecanismo *uka.transport*, percebe-se uma melhora significante no desempenho. Uma vantagem do KaRMI, que não foi explorada neste estudo, é a possibilidade da utilização de tecnologias de redes de alto desempenho.

A biblioteca JPVM modela o conceito de máquinas paralelas virtuais baseado em troca de mensagens. Esta biblioteca não introduz nenhuma otimização nos mecanismos do Java, mas fornece um paradigma de comunicação tradicionalmente utilizado para a construção de aplicações paralelas em ambientes de memória distribuída. Seu desempenho é penalizado devido ao fato das suas mensagens serem compostas por um objeto que armazena os dados a serem enviados para a tarefa remota. Portanto, o envio de um simples byte em JPVM corresponde ao envio de um objeto, inserindo os custos da serialização padrão Java. Contudo, nota-se que seu desempenho se aproxima do desempenho da implementação padrão do Java RMI e chega a ultrapassar o desempenho do KaRMI (sem *uka.transport*) para grandes quantidades de dados. No experimento com a aplicação SOR, o JPVM se destacou por ter seu tempo de resposta reduzido em relação às demais propostas à medida que as dimensões das matrizes utilizadas foram sendo aumentadas. Isto pode ser explicado pelo fato do modelo PVM ser mais adequado do que o modelo de objetos distribuídos para a construção de aplicações paralelas, fornecendo assim uma implementação mais eficiente do algoritmo SOR.

A implementação padrão do Java RMI foi originalmente projetada para oferecer um suporte flexível para a computação de objetos distribuídos e não para fornecer um ambiente de alto desempenho. Entre os ambientes avaliados, esta implementação é a que oferece mais facilidades para os usuários, além de ser a mais simples de ser utilizada, uma vez que é o mecanismo

padrão da linguagem Java. Não se pode dizer que algum destes ambientes é consistentemente o melhor para a implementação de sistemas baseados em troca de mensagens. Cada um tem suas características e se adapta melhor a certas condições. Implementações eficientes, como o Manta ou KaRMI, fazem que o Java se apresente como uma alternativa para a construção de aplicações de propósito geral que exigem altos requisitos de desempenho. Contudo, diversas aplicações podem requerer múltiplas abstrações de comunicação, tornando interessante a pesquisa de outros paradigmas de comunicação em Java, como o PVM ou o MPI. A situação ideal seria a possibilidade de utilizar uma combinação destes ambientes de forma a atender as necessidades de aplicações específicas. Uma ferramenta que permita a integração destes ambientes de forma que o projetista possa selecionar os mecanismos adequados para atender os requisitos de desempenho de sua aplicação, tirando o máximo proveito do suporte operacional disponível, seria de grande utilidade para este fim. Por exemplo, através desta ferramenta poderia ser combinada a utilização do modelo de programação PVM, tal como o fornecido pela JPVM, em conjunto com as otimizações introduzidas pelo compilador Manta. Isto permitiria usuários acostumados com a sintaxe e a semântica do ambiente PVM utilizarem a plataforma Java, tirando vantagem de seus mecanismos de programação concorrente, sem serem penalizados por grandes restrições de desempenho. Ainda, neste mesmo exemplo poderia ser utilizado o suporte do KaRMI para o acesso a múltiplas interfaces de rede, facilitando a adaptação do ambiente de suporte de software à infra-estruturas de redes de alto desempenho específicas.

## 6. Referências

[1] Arnold, K., J. Gosling, and D. Holmes, The Java™ Programming Language – Third Edition, The Java Series, Addison-Wesley Publisher Co., 2000.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification – Second Edition", Sun Microsystems Inc., 2000, disponível em http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[3] Sun Microsystems, "Java™ Remote Method Invocation", Sun Microsystems Inc., 1999, disponível em http://java.sun.com/j2se/1.3.0/docs/guide/rmi/spec.

[4] C. Nester, M. Philippsen, and B. Haumacher, "A More Efficient RMI for Java", In Proceedings of the ACM Java Grande Conference, San Francisco, CA., Jun 1999, pp. 152-157.

[5] J. Maassen, R. van Nieuwpoort, R. Valdema, A. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programing", ACM Transactions on Programming Languages and Systems (TOPLAS), Vrije Universiteit Amsterdam, Faculty of Sciences, 2001.

[6] T. Kielmann, P. Hatcher, L. Bougé, and H. E. Bal, "Enabling Java for High-Performance Computing", Communications of the ACM, ACM, New York, Oct 2001.

[7] A. J. Ferrari, "JPVM: Network Parallel Computing in Java", Technical Report CS-97-29, Department of Computer Science, University of Virginia, Charlottesville, 1998.

[8] Java Grande Conference, "Interim Java Grande Fórum Report", Java Grande Fórum Technical Report JGF-TR-4, ACM Java Grande Conference/JavaOne Conference, San Francisco, CA, Jun, 1999.

[9] M. Lobosco, C. Amorim e O. Loques, "Java for High-Performance Network-Based Computing: A Survey", Concurrency and Computation: Practice and Experience, Wiley, (36 pp), V. 114, Issue 1, 2002.

[10] B. Haumacher, J. Reuter, and M. Philippsen, "Fast Object Serialization uka.transport", JavaParty release 1.05b, Oct, 2001, disponível em http://wwwidp.ira.uka.de/JavaParty/ukatransport.html.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing Scientific and Engineering Series" , MIT Press, 1994.

[12] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A Methodology for Benchmarking Java Grande Applications", The University of Edinburgh, Scotland, U.K. – ACM 1999 Java Grande Conference Program, 1999.