Proposta para melhorar a eficiência da comunicação entre aplicações Java via redução de cópias intermediárias, redução de trocas de contexto, e serialização otimizada de objetos

Ricardo L. Kulzer[1], Leonardo de S. Mendes[1], Marcelo Pasin[2]

[1] Departamento de Comunicações - Faculdade de Engenharia Elétrica e Computação
Universidade Estadual de Campinas (UNICAMP) – 13081-970 – Campinas – SP – Brazil

[2] Laboratório de Sistemas de Computação - Universidade Federal de Santa Maria
(UFSM) - Cidade Universitária, prédio 7 – 97105-900 – Santa Maria – RS – Brazil
{ricardol,lmendes}@decom.fee.unicamp.br, pasin@inf.ufsm.br

Abstract. *This paper proposes an approach that aims for performance improvements in the communication of distributed Java applications for clustering. The proposal include an implementation of custom device drivers for network cards (aiming for reduction of the intermediate copies and context switches), a communication library for the Linux Operating System, and optimize the Java object serialization, in order to reduce the overhead involved in the end-to-end communication process.*

Resumo. *Este artigo propõe uma abordagem que visa melhorias de desempenho na comunicação de aplicações Java distribuídas para aglomerados. A proposta inclui a implementação de drivers customizados para placas de rede (buscando redução do número de cópias intermediárias e do número de trocas de contexto), uma biblioteca de comunicação para o Sistema Operacional Linux, e otimização da serialização dos objetos Java, no sentido de reduzir o overhead envolvido na comunicação fim-a-fim.*

1. Introdução

Aplicações distribuídas que requerem alto desempenho na comunicação, necessitam que o atraso na comunicação seja o menor possível. O atraso na comunicação entre dois processos remotos contém duas componentes: tempo gasto no processamento da mensagem e atraso da rede. Em redes de baixa velocidade, tais como *Ethernet* a 10 Mbps, o gargalo da comunicação é o atraso da rede. Isto levou ao desenvolvimento de mecanismos de comunicação baseados em *sockets*, onde o processamento das mensagens no núcleo do sistema operacional causa muitas cópias e muitas trocas de contexto, o que aumenta muito o atraso fim-a-fim.

Porém, em redes mais rápidas, como por exemplo, nas redes *Gigabit Ethernet*, o gargalo da comunicação passa a ser o tempo gasto no processamento das mensagens nos pontos terminais de comunicação [Shivam, Wyckoff and Panda 2001]. E esta mudança do gargalo tende a se evidenciar ainda mais com a evolução do *Ethernet*: hoje já é comum no mercado *hardware Ethernet* 10 Gbps, e o grupo de trabalho IEEE 802.3 já discute a tecnologia *Ethernet* a 100 Gbps [Recio 2003].

O mecanismo padrão empregado para comunicação de dados via rede, na maior parte dos sistemas operacionais, é baseado em *sockets*. Este mecanismo apresenta um alto grau de atrasos, decorrentes da excessiva quantidade de cópias e de trocas de contexto. A eliminação de algumas cópias e/ou chamadas de sistema permite reduzir o *overhead* e aumentar a performance da comunicação.

Já ao nível de aplicação, observa-se que as facilidades oferecidas pela linguagem Java aliadas à expansão das redes de computadores têm proporcionado um crescimento da quantidade de aplicações distribuídas. Porém, o mecanismo adotado pelo Java para a troca de mensagens é baseado em *sockets* tradicionais, que aliado a pouca otimização dos mecanismos de serialização e transferência da máquina Java, resulta em uma comunicação de baixo desempenho [Haumacher and. Philippsen 1999]. Esta característica é indesejável para aplicações que requisitam baixo nível de atraso na comunicação.

Neste contexto, o objetivo desta proposta é implementar mecanismos para melhorar o desempenho da comunicação em aplicações desenvolvidas em linguagem Java para aglomerados. Este trabalho está sendo desenvolvido abordando dois pontos passíveis de aprimoramento: redução do número de cópias e de trocas de contexto na comunicação de baixo nível do sistema operacional, e otimização da serialização de objetos Java.

Como este trabalho é voltado para a comunicação de aplicativos Java executados no Linux, a proposta inicial tratará da otimização da serialização de objetos em tempo de compilação, com a utilização do suíte *GNU Compiler Collection* (GCC) e sua interface nativa CNI - *Cygnus Native Interface* [Bothner and Tromey 2001], uma interface provida pelo GCC que permite a interação de códigos nativos com a linguagem Java de forma mais eficiente que a interface JNI fornecida pela Sun.

Cabe salientar que outros pesquisadores têm buscado melhorias no desempenho da comunicação com a redução dos atrasos decorrentes da complexidade presente na comunicação, reduzindo o número de cópias dos dados transferidos e o número de trocas de contexto [Eicken, Basu, Buch and Vogels 1995], [Skevik, Plagemann, Goebel and Halvorsen 2001], [Balaji, Shivam, Wyckoff and Panda 2002], [Kim, Kim, Jung and Ha 2003], [O'Carroll, Tezuka, Hori and Ishikawa 1998], [Lentini, Pham, Sears and Smith 2003] e [Rangarajan and Iftode 2004]. Pesquisas também têm buscado aprimorar o desempenho com melhorias na otimização da serialização dos objetos Java trocados entre aplicativos [Opyrchal and Prakash 1999], [Breg and Polychronopoulos 2003], [Haumacher and. Philippsen 1999] e [Maassen, Nieuwpoort, Veldema, Bal and Plaat 1999]. O que diferencia a nossa abordagem é que: 1) ela abrange a **otimização na comunicação entre processos Java remotos cobrindo todos os níveis da comunicação**, desde o *driver* até a aplicação. 2) os trabalhos acima utilizam dispositivos de rede de alto desempenho e que oferecem mais recursos para programação (Alteon TIGON-II, Myricom Myrinet, InfiniBand, etc.), porém tais dispositivos têm preço elevado, o que restringe muito a sua aplicabilidade. Já a nossa abordagem, focaliza dispositivos de rede mais comuns, e de **baixo custo**.

1.1. Organização deste artigo

No item 2, apresenta-se uma descrição do problema do atraso imposto pelo excessivo número de cópias e trocas de contexto ao nível de sistema operacional. No item 3,

analisa-se a lógica de transmissão e recepção de dados comumente implementada nas placas de rede modernas. No item 4, apresenta-se uma proposta para melhorar o desempenho da comunicação no nível de sistema operacional. No item 5, discute-se a otimização da serialização de objetos Java. No item 6, apresentamos alguns resultados preliminares. Finalmente, no item 7 são apresentadas algumas conclusões.

2. Atraso na comunicação via rede imposto pelo sistema operacional

Como vimos, o mecanismo empregado para comunicação de dados na maioria dos sistemas operacionais apresenta grande *overhead* devido às cópias intermediárias e trocas de contexto, além do processamento dos protocolos de rede nos pontos terminais de comunicação. Para este trabalho em especial, pretende-se implementar mecanismos que buscam reduzir este *overhead* no sistema operacional Linux. Porém, as idéias obtidas deste trabalho aplicam-se também a outros sistemas operacionais.

Em [Rubini and Corbet 2001] temos uma excelente descrição do processamento envolvido na troca de dados entre processos remotos no Linux. O problema existente na forma tradicional de comunicação é o excessivo número de cópias dos dados envolvidos na comunicação entre processos remotos. Conforme ilustra a Figura 1a (Abordagem Tradicional), para um processo A enviar dados a um processo B em outro computador, são necessárias, no mínimo, as seguintes cópias [Beck 1998]: 1) do espaço de endereçamento do processo A para o núcleo do sistema operacional do transmissor; 2) do núcleo do sistema operacional do transmissor para a placa de rede; 3) da placa de rede do transmissor para a placa de rede do receptor; 4) da placa de rede do receptor para o núcleo do sistema operacional receptor; e 5) do núcleo receptor para o espaço de endereçamento do processo B.

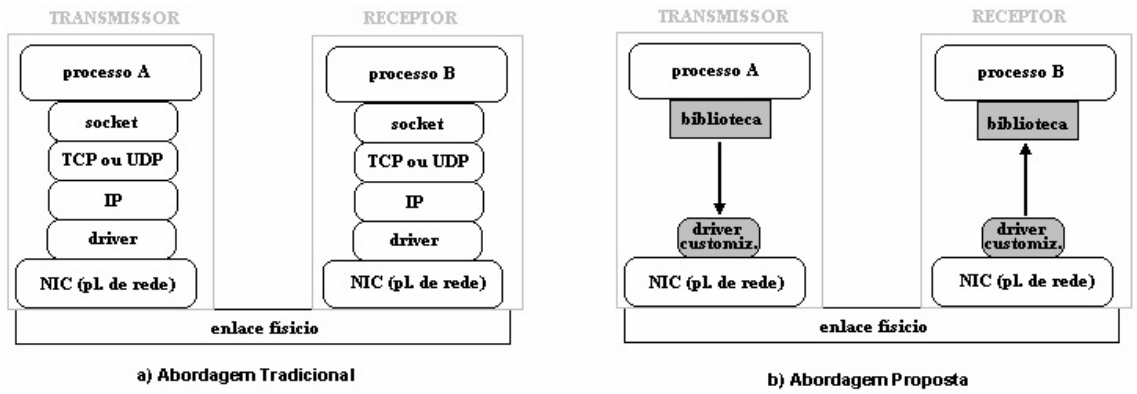

Figura 1. O caminho dos dados na comunicação entre processos remotos

Além das cópias dos dados, existe ainda o *overhead* imposto pelo processamento realizados nas diversas camadas (*socket* / camada de transporte / camada IP / *driver*) realizado nos pontos terminais de comunicação.

A partir dessa análise, propomos a criação de uma biblioteca e de um *driver* de comunicação de rede que permita a um processo enviar e receber dados controlando diretamente a placa de rede, poupando cópias dos dados na memória de cada computador, bem como poupando o atraso extra causado pelo uso de várias camadas na implementação do suporte à rede no sistema operacional. Os processos utilizam uma biblioteca de comunicação que interage com um *driver* controlador de dispositivo

customizado, o que evita as cópias feitas pelo sistema operacional. Desta forma, os dados trafegam pelo caminho ilustrado na Figura 1b (Abordagem Proposta).

3. Lógica de transmissão e recepção de dados nos dispositivos de rede

Com o intuito de buscar uma solução ao problema, inicialmente estudou-se a documentação técnica de algumas placas de rede *Fast Ethernet* e *Gigabit Ethernet* bastante difundidas e, principalmente, de baixo custo. Deste estudo, verificou-se que as placas modernas realizam a comunicação através de transferências DBDMA (*descriptor based DMA*). Este modo de operação permite que a própria placa realize a transferência entre a memória principal e a rede, através da varredura de uma lista pré-computada de "descritores de DMA" armazenados na memória principal. Assim, as transferências de baixo nível entre a memória principal e a rede são controladas pela própria placa de forma autônoma, liberando a CPU para outras atividades.

Os descritores de DMA permitem que a placa utilize regiões fixas da memória principal para armazenar quadros que chegam da rede (na recepção), bem como quadros que serão enviados pela rede (na transmissão). Cada descritor referencia um *buffer* da memória principal onde é armazenado o conteúdo do quadro.

Cabe salientar que as diferentes placas de rede existentes no mercado apresentam peculiaridades e funcionalidades distintas. Entretanto, em todos os dispositivos que utilizam DBDMA, o controle da transmissão e da recepção é feito através de uma estrutura de dados bastante simples: uma lista ligada.

3.1. A transmissão de dados em placas que operam com DBMA

O algoritmo de transmissão basicamente é: o *driver* cria uma lista de descritores de DMA, e escreve o endereço da lista em um registrador específico da placa de rede. A placa imediatamente descarrega os dados da memória principal (via DMA) logo que detectar um valor diferente de "0" (zero) no registrador. Após descarregar os dados referenciados pela lista de descritores, o circuito de transmissão zera o registrador, e segue testando-o, aguardando o *driver* escrever o endereço de novos descritores para novas transmissões.

3.2. A recepção de dados em placas que operam com DBMA

No lado do receptor, o processo é bastante similar. O *driver* constrói uma lista de descritores, sendo que, neste caso, cada descritor referencia uma região de memória para a qual a placa deverá carregar os dados do próximo quadro recebido. Quando um quadro chega, a placa de rede varre a lista de descritores, e copia (via DMA) os dados para os *buffers* referenciados pela lista de descritores.

O algoritmo de recepção essencialmente é: o *driver* cria uma lista de descritores de recepção, e escrever o endereço físico de memória da lista de descritores em um registrador específico da placa. Sempre que a placa receber um novo quadro, ela varre a lista de descritores e copia os dados recebidos para os *buffers* da memória principal, referenciados pelos descritores de DMA.

3.3. Dispositivos de rede analisados

A lógica utilizada nos dispositivos de rede para transferência de dados, descrita no item anterior, está presente na maioria das placas de rede *Fast Ethernet* e *Gigabit Ethernet* modernas. Para este trabalho, foi feito um estudo da documentação técnica dos seguintes dispositivos de rede:

FAST ETHERNET ⇒

- 3C905 da 3Com® [3C90x and 3C90xB NICs Technical Reference, 1998]; e
- RTL8100 da RealTek® [Realtek RTL8100 Programming Guide 2001]

GIGABIT ETHERNET ⇒

- RTL8169S e RTL8110S da RealTek® [RTL8169S/8110S Programming Guide 2004];
- 3C2000 da 3Com®, EG1032 ver. 2 da Linksys®, e SK-9841 da SysKonnect® [SK-NET GEnesis Technical Manual 1998];
- 3C985B da 3Com®, GA620 da Netgear®, e AceNIC da Alteon® [Tigon/PCI Ethernet Controller 1997]; e
- DGE-500SX da D-Link® [LXT1001 Network Controller Software Manual 1999].

O estudo da documentação destes dispositivos mostrou que a lógica de transmissão e recepção de dados, apesar das peculiaridades e características de cada placa, é baseada em listas de descritores de *buffers* para transmissão e recepção. Desta forma, uma solução implementada inicialmente em um dispositivo poderá ser posteriormente replicada para outros dispositivos.

4. Proposta para melhorar o desempenho da comunicação ao nível de sistema operacional

Para aumentar a eficiência da transmissão, o ideal é que os dados do processo sejam enviados diretamente para a placa, sem cópias intermediárias no núcleo. Na recepção, vale a mesma idéia.

Uma questão importante é o acesso aos dados da memória principal pelo mecanismo de DMA da placa de rede: a placa precisa receber endereços de memória reais em seus registradores para fazer DMA, pois os controladores de DMA não sabem nada sobre memória virtual, mas apenas sabem acessar blocos contínuos de memória física. Assim, não é possível efetuar DMA diretamente em endereços virtuais da área de memória dos processos, cuja tabela de tradução é mantida pela MMU (*Memory Management Unit*). Para resolver esta questão, nesta implementação, os *buffers*, para envio e recepção de mensagens, são alocados na área de memória correspondente ao segmento de núcleo, sendo posteriormente mapeados pelos processos em variáveis que podem ser usadas pela aplicação. Como os dados, a serem enviados ou recebidos, são armazenados no segmento de núcleo, a conversão entre o endereço de memória virtual e o endereço físico dos *buffers*, para que a placa faça DMA, é direta. Além disso, garante-se que a memória não sofra troca (*swapping*).

O *driver* implementado cria listas de descritores de DMA na fase de inicialização. Os *buffers* referenciados pelos descritores também são alocados na região do núcleo. Os programas podem acessar os *buffers* alocados pelo *driver* usando

mapeamento de memória. Para tal, a maioria dos sistemas operacionais possui uma chamada de sistema *MMAP*, que permite mapear endereços de memória do processo para outros endereços de memória, ou até mesmo para dispositivos.

Para um processo **enviar** uma variável, primeiramente mapeia o endereço de memória da variável para um dos *buffers* mantidos pelo *driver*, compõe a mensagem usando a variável e escreve, no registrador da placa, o endereço do descritor de DMA vinculado ao *buffer* da variável. A placa então transfere os dados via DMA, enviando os dados pela rede.

A **recepção** de mensagens é semelhante. Um processo mapeia uma variável para um *buffer* mantido pelo *driver*. Ao executar a chamada *RECEIVE*, o processo é bloqueado até a chegada de alguma mensagem. Neste caso, também não é necessário copiar os dados para a memória do processo, pois a variável é mapeada para a área alocada pelo *driver*, para onde serão transferidos os dados recebidos pela placa via DMA.

O *driver* possibilita a comunicação usando protocolo IP ao mesmo tempo. Desta forma, outros aplicativos que utilizam o protocolo TCP/IP ou UDP/IP podem funcionar normalmente. Observe que o *driver* customizado simplesmente identifica o protocolo através do campo "Tipo de Quadro", um campo de 2 octetos do cabeçalho dos quadros do padrão *Ethernet*. Esta identificação possibilita ao *driver* dar o tratamento adequado aos quadros.

A estrutura dos quadros é customizada. Ao invés de utilizar a estrutura do padrão MAC IEEE 802.3, os quadros trocados pelo *driver* apresentam, além da carga útil de informações, um cabeçalho composto de 19 *bytes*, com os seguintes campos: processo de origem (1 *byte*), processo de destino (1 *byte*), *timestamp* (1 *byte*), tamanho (4 *bytes*), endereço MAC de origem（6 *bytes*), e endereço MAC de destino（6 *bytes*). Já o controle de concorrência entre processos concorrentes ao acesso da placa de rede é baseado no emprego de semáforos.

O *driver* interage com o aplicativo através de funções de uma biblioteca de comunicação. O propósito desta biblioteca é ocultar do programador a complexidade da programação envolvida na comunicação usando o sistema proposto, e oferecer aos processos pontos terminais de comunicação.

5. Otimização do processo de serialização e transferência de objetos Java

O protocolo de serialização da máquina Java da Sun é escrito na linguagem Java e usa reflexão para determinar o tipo de cada objeto em tempo de execução [Maassen, Nieuwpoort, Veldema, Bal and Plaat 1999]. Esta reflexão em tempo de execução representa um grande gargalo na comunicação do Java.

Além disso, a serialização de objetos do JDK inclui a descrição completa do tipo da variável na seqüência de *bytes* que representa o estado de um objeto sendo serializado [Philippsen, Haumacher and Nester 2000]. Este esquema de serialização é necessário para o armazenamento de objetos persistentes, em que os objetos são armazenados em disco de forma que eles guardem seu estado, mesmo quando o programa for encerrado.

Como este trabalho é voltado para a comunicação de aplicativos Java executados no Linux, pretende-se otimizar este processamento através da geração de código de serialização em tempo de compilação, utilizando o suíte *GNU Compiler Collection* (GCC). O GCC, *software* livre, é uma coleção de compiladores para as linguagens C, Fortran, C++, Objective C e Java. Entre as virtudes do GCC temos que todos os compiladores compartilham a mesma estrutura básica, usam um mesmo conjunto de *back ends* (parte do programa que executa tarefas secundárias, não diretamente controladas pelo usuário), é portado para várias plataformas [Baker and Ong 2003].

A biblioteca de comunicação, responsável pelo controle da transmissão e recepção de quadros através do *driver* customizado, é implementada em linguagem C++. Para que o código gerado pela compilação das aplicações Java possa interagir com o código da biblioteca, utiliza-se a interface nativa CNI, uma interface provida pelo GCC que permite a interação de códigos nativos com a linguagem Java de forma mais eficiente que a interface JNI fornecida pela Sun. Cabe aqui ressaltar que a Sun definiu para o seu compilador JDK uma interface de programação conhecida como *Java Native Interface* (JNI) para possibilitar a chamada de métodos nativos escritos em C ou C++. Porém, o *overhead* do JNI é muito alto. Por exemplo, o código nativo precisa fazer duas chamadas de sistema para acessar um campo em um objeto: uma para encontrar a classe e outra para encontrar o campo dentro da classe. O GCC provê a interface CNI como alternativa, que é bem mais eficiente que a JNI.

6. Resultados preliminares

Apesar de a implementação deste sistema não estar completa, um protótipo para teste de desempenho foi desenvolvido, visando validar os conceitos expostos neste artigo, bem como prover uma análise qualitativa desta abordagem comparada à abordagem tradicional. A placa escolhida para o protótipo é a 3c905b, por ser barata, possuir bom desempenho, ser bastante difundida e possuir uma excelente documentação técnica. Posteriormente, pretende-ser replicar esta solução para outros dispositivos *Fast Ethernet* e *Gigabit Ethernet*.

Para os experimentos, foram utilizados dois computadores (um Athlon XP 1.7, e outro K6-II 500), conectados à LAN com placas de rede *fast-ethernet* 3C905B, ambos executando Linux 2.4.17. Para avaliar os atrasos na troca de mensagens, foram feitos repetidos experimentos com "ping-pong" para troca de *strings* com tamanhos variáveis. A aplicação "ping-pong" envolve executar um processo "ping" em um computador cliente, e um processo "pong" no computador servidor. "Ping" inicialmente guarda o tempo corrente do sistema, envia uma mensagem e bloqueia, aguardando a resposta do "pong". Assim que chegar a resposta, o aplicativo novamente toma o tempo corrente do sistema, para computar o tempo total entre o envio da mensagem e a chegada da resposta. Este procedimento é repetido inúmeras vezes.

Para avaliar o desempenho com o "ping-pong", foram realizados utilizando três tipos de experimentos:

- ABORDAGEM PROPOSTA. Comunicação utilizando a abordagem proposta;

- JAVA UDP: Comunicação utilizando UDP (objetos *DatagramPacket*) com aplicação "ping-pong" escrita em Java, compilada com Sun-JDK e executadas usando a máquina virtual Java da Sun, versão 1.5; e
- GCC UDP: Comunicação utilizando UDP (*sockets SOCK_DGRAM*) entre aplicações implementadas em C++, compiladas com GCC.

A Figura 2 mostra o resultado da regressão linear aplicada aos valores de atraso obtidos na comunicação.

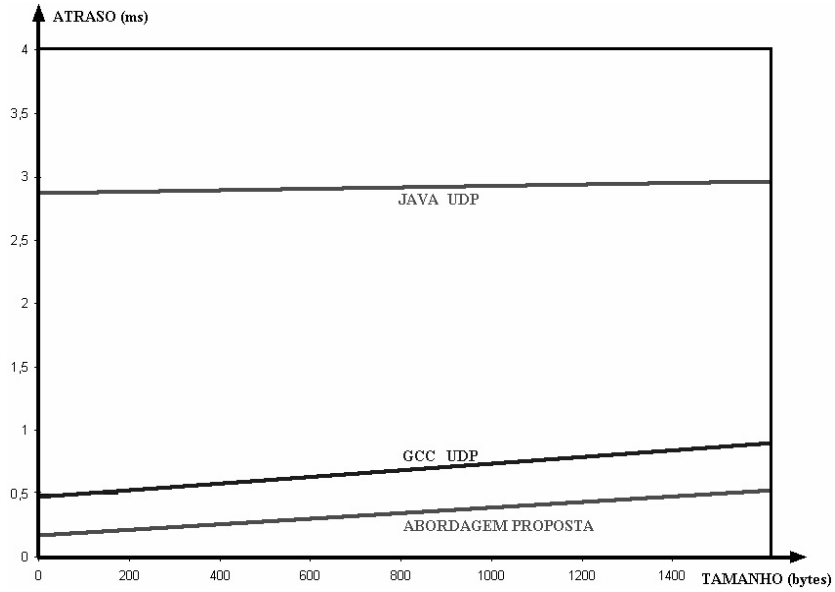

Figura 2. Atrasos em função do tamanho da mensagem

O resultado demostra que a comunicação "JAVA UDP" apresentou um atraso elevado se comparado à comunicação "GCC UDP". O atraso obtido na "ABORDAGEM PROPOSTA" está dentro do que se esperava, pois a compilação utilizando o GCC e sua interface nativa CNI gera código nativo. Deste modo, a aplicação utiliza-se dos mecanismos de redução de *overhead* implementados neste protótipo, obtendo assim um desempenho melhor quando comparado ao obtido com comunicação "GCC UDP", também executada com código nativo.

7. Conclusões

Este artigo propõe uma nova abordagem para a implementação da comunicação entre processos remotos desenvolvidos em linguagem Java para aglomerados. Nesta abordagem, a melhora da eficiência é obtida com a redução de cópias intermediárias e de trocas de contexto, bem como com a otimização da serialização dos objetos Java. Entretanto, a implementação atual requer que ambos os lados utilizem o *driver* e a biblioteca customizada para a comunicação. Neste caso, a compatibilidade é trocada por desempenho.

Com o argumento de que a probabilidade de perda de pacotes devido a erros de *hardware* ou congestionamentos na rede é muito baixa, a proposta inicial não trata da detecção e correção de erros na comunicação. Porém, em trabalhos futuros poderão ser implementados também mecanismos para correção de erros, controle de fluxo e de congestionamento na biblioteca.

Quanto à compatibilidade na otimização da serialização de objetos Java, pretende-se explorar este aspecto, não utilizando mais o GCC para gerar código nativo, mas sim com a geração de *byte-code* (.class) em tempo de compilação com código de serialização otimizado e também com acesso a comunicação usando o sistema desenvolvido neste trabalho. Uma proposta é, por exemplo, adaptar o compilador Manta [Maassen, van Nieuwpoort, Veldema, Bal, and Plaat 1999], que possibilita gerar código do tipo *byte-code* com otimização da serialização de objetos. Assim, pretende-se gerar um conjunto de classes com API similar às oferecidas pela *DatagramPacket* provida pela máquina Java. Desta forma, para utilizar a solução proposta, o programador poderá simplesmente substitui a classe *DatagramPacket* pela nossa classe, mantendo desta forma intacto o restante do seu código.

8. Referências

3C90x and 3C90xB NICs Technical Reference; 3COM Corporation, Aug. 1998.

B. Haumacher, M. Philippsen: More efficient object serialization. In Parallel and Distributed Processing, number 1586 in Lecture Notes in Computer Science, pages 718-732, Puerto Rico, April 12, 1999.

Baker, M. and Ong, H.: Design and Implementation of Java Embedded Microkernel Software Architecture. Submitted to IPDPS (International Parallel and Distributed Processing Symposium) - 2003. http://dsg.port.ac.uk/projects/JEM/docs/baker.pdf

Balaji P., Shivam P., Wyckoff P. and Panda, D. K.: High Performance User-Level Sockets over Gigabit Ethernet, IEEE International Conference on Cluster Computing, Sept 23-26, 2002, Chicago, IL.

Beck, M.; Linux Kernel Internals - 2nd Edition. Addison-Wesley, 1998.

Bothner, P. and Tromey T.: Java/C++ integration: Writing native Java methods in natural C++. Submitted to Usenix JVM 01 (Java Virtual Machine Research and Technology Symposium) – 2001. http://dsg.port.ac.uk/projects/JEM/docs/baker.pdf

Breg, F. and Polychronopoulos C. D.: Java Virtual Machine Support for Object Serialization. Proceedings of JavaGrande 2001/International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE) 2001, Concurrency and Computation: Practice and Experience (vol 15, issue 3-5). John Wiley & Sons. 2003. 263-275.

Kim J., Kim, K., Jung, S., and Ha, S: Design and Implementation of a User-level Sockets Layer over Virtual Interface Architecture. Concurrency and Computation: Practice and Experience, Volume 15, Issue 7-8, pp.727-749, June-July, 2003.

Lentini, J., Pham, V., Sears, S., and Smith R.: Implementation and Analysis of the User Direct Access Programming Library. In 2nd Workshop on Novel Uses of System Area Networks, SAN-2, February 2003.

LXT1001 Network Controller Software Manual; Level One Communications, Inc.; Rev. 1.0, July 1999.

M. Rangarajan and L. Iftode.: Building a User-Level Direct Access File System over Infiniband. In the Proceedings of the 4th Annual Workshop on System Area Networks (SAN-4), Madrid, February 2004.

Maassen, J., van Nieuwpoort R., Veldema, R., Bal, H. E. and Plaat, A: An Efficient Implementation of Java's Remote Method Invocation, Proc. Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pp. 173-182, Atlanta, GA, May 4-6, 1999.

O'Carroll, F., Tezuka, H., Hori, A., Ishikawa, Y.: The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. International Conference on Supercomputing 1998: 243-250

Opyrchal, L. and Prakash A.: Efficient Object Serialization in Java, in Proc. of ICDCS 99 Workshop on Middleware, Austin, June 1999.

Piyush Shivam, Pete Wyckoff, D. Panda EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing Proceedings of SC2001, Denver, Colorado, November '01.

Realtek single chip Fast Ethernet Controller with Power Management RTL8100 Programming Guide. Realtek Semiconductor Corp., Rev.1.0, 10 Dec. 2001.

Recio RJ. Server I/O networks past, present, and future. In: Proc. of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications. New York: ACM Press, 2003. 163-178.

RTL8169S/RTL8110S Gigabit Ethernet Controller Programming Guide. Realtek Semiconductor Corp., Rev. 1.0, 14 Feb. 2004.

Rubini, A. and Corbet, J.: Linux Device Drivers, 2nd edition, O'Reilly, Sebastopol, 2001.

Skevik, K.-A., Plagemann, T., Goebel, V., Halvorsen,. Evaluation of a Zero-Copy Protocol Implementation. P., Proceedings of the 27th Euromicro Conference - Multimedia and Telecommunications Track (MTT'2001),Warsaw, Poland,September,2001

SK-NET GEnesis Technical Manual; SysKonnect, Inc., Version 1.0, November 1998.

Stancevic, Dragan. "Zero Copy I: User-Mode Perspective". Linux Journal, Issue 105, January 2003. http://www.linuxjournal.com/article/6345

Tanenbaum, S. A.; Redes de Computadores; Editora Campus, 1997.

Tigon/PCI Ethernet Controller; Alteon Networks, Inc.; Revision 1.04, August 1997.

von Eicken, T., Basu, A., Buch, V. and Vogels, W.: U-Net: A user-level network interface for parallel and distributed computing. In Proc. Fifteenth ACM Symp. on Operating System Principles (SOSP), pages 40--53, 1995.