UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

TÍTULO: Uma Família de Abstrações Myrinet para EPOS

AUTOR: Fernando Roberto Secco

ORIENTADOR: Prof. Antônio Augusto Medeiros Fröhlich

BANCA EXAMINADORA: Prof. José Mazzucco Junior, Prof. Mário A. R. Dantas, Mestre André Luís Gobbi Sanchez

PALAVRAS-CHAVE: Protocolos de Alto-Desempenho, EPOS, Redes de baixa latência, Myrinet

Florianópolis, Junho de 2004

A minha Família

Agradecimentos

Bem, nem sei por onde começar, mas acho que meus pais são o que realmente me vem a cabeça nessa hora por terem acreditado em mim e terem me dado uma chance. Também gostaria de agradecer a minha amada Aline por toda força e paciência que ela teve durante esse período e minha gata Jubilee por sempre estar no meu colo, aos meus amigos aqui de Florianópolis Boi, SpaceTug, Guille, Gobbi, Hederson, Frango, Jujuba, Tiga, Oldman, Pink e aos meus distantes amigos de muitos combates, Midwinter, Crowzer, C.E.R.T., PlasticHandBag, ZEEP:(, Iom, Prowler, a HK69 e AKMSU por tudo que elas fizeram por mim. As bandas de metal Steel Warrior, Iron Maiden, Grave Digger, Black Sabbath, Riot, Judas Priest e outras pelas trilhas sonora maravilhosas das noites no Cluster. Gostaria também de agradecer ao Professor Guto por toda a paciência e pela oportunidade, ao Robert "sem-luz-sem-lei" pela grande força e pelos *brainstorms*, ao professor Aldo que me deu a primeira chance de realmente me envolver em pesquisa e ao professor Júlio Zeremeta por ter ajudado muito com a formalização do protocolo e do sistema de comunicação, aos professores Mazzucco e Dantas por aceitarem ser parte da minha banca. Ao pessoal da Bioinfo, Charles, Haeser, Edmundo. Por fim, gostaria de agradecer as pessoas e projetos que me ajudaram bastante, mesmo sem saber, Linus Torvalds, Leslie Lamport, GNU, Ispell, XMMS, TrollTech, SentryStudios, Mozzila, Emacs e Google Inc.

Sumário

Lista de Figuras

Resumo

Este trabalho apresenta um sistema de comunicação dedicado para redes de baixa-latência myrinet que utiliza o sistema operacional EPOS. Como metodologia foi utilizada a AOSD cujas maiores caracteristicas são a flexibilidade e simplicidade de sua componentização. A proposta do sistema de comunicação é manter o sistema o mais simples possíveis para garantir o menor atraso para mensagens curtas e a melhor utilização da largura de banda para mensagens longas.

É possível ainda, observar o impacto do software de comunicação no desempenho da aplicação. Como um sistema dedicado pode vir a contribuir com melhores soluções utilizando componentes mais simples. Os resultados alcançados com um sistema dedicado em comparação com um genérico encoraja os desenvolvedores a repensarem as metodologias empregadas no desenvolvimento de software básico.

PALAVRAS CHAVE: Protocolos de Alto-Desempenho, EPOS, Redes de baixa latência, Myrinet

Abstract

This work introduces a dedicated communication system over myrinet , a low-latency network, for EPOS operating system. As a methodology AOSD was chosen for its flexibility and the ability to create new components easily. The main objective of this communication system is provide performance using a well designed code, acquired with methodology, benchmarks, formalized methods and simplicity.

Besides, its shown the impact caused by software in the application performance and how can a dedicated system contribute with better solutions using simple and combinable components. The obtained results ,using a dedicated system compared with a generic one, encourages developers to rethink about the use of ordinary methodology in basic software.

KEYWORDS: High perfomance Protocols, Low-Latency Networks, EPOS, Myrinet

Capítulo 1

Introdução

Uma das mais eficientes maneiras de se realizar grandes tarefas computacionais é com a utilização de Super Computadores também conhecido como Massively Parallel Processors (MPP). Estas máquinas são dotadas de uma coleção de processadores e barramentos internos extremamente rápidos e utilizam software básico desenvolvido especialmente para a arquitetura existente naquela máquina. Assim, obtém-se desempenho próximo ao oferecido pelo hardware.

O hardware presente nesses supercomputadores faz com que o preço deste tipo de equipamento seja muito elevado. Estas peças são desenvolvidas com tecnologia de ponta. Seu processo de construção é bastante demorado e um projeto pode levar mais de dois anos para um projeto ficar pronto. Assim, se for levada em conta a velocidade com que a tecnologia de hardware muda ao chegar ao mercado, a maioria dos componentes de hardware de um supercomputador estará disponível para computadores pessoais (PCs), ou já estará defasada. Como exemplo pode-se citar a melhora nos processadores de 32 bits que a cada 1,5 anos tem seu poder de processamento duplicado [Anderson et al. 1995].

Assim, quanto mais rápido se der o término do projeto e o lançamento no mercado melhor será sua recepção. Mas em contrapartida, as despesas para acelerar o término prematuro do projeto acabam elevando o preço final do produto

[Anderson et al. 1995]. Além do processo de fabricação, outros fatores implicam no custo de um MPP a longo prazo. O hardware é bastante inovador e específico, logo o desenvolvimento e a manutenção do software básico também eleva o custo de comercialização em larga escala. Assim, com o lançamento de uma nova verão do supercomputador o seu software básico precisa ser praticamente todo refeito.

Em 1991 uma alternativa aos MMPs surgiu: os agregados de computadores pessoais ou *clusters* de computadores pessoais. Um *cluster* é um conjunto de computadores interconectados por uma rede local ou de sistema . Agregados podem ter desempenho semelhante ao dos MPPs, se o software adequado for utilizado.

Através da melhoria nos componentes de hardware e software tais como placas de rede, cabos, *switches*, memória, barramento, sistema operacional e com a melhora nas técnicas de confecção de software, foi possível obter hardware para CCWs que possuísse um desempenho próximo ao do hardware de um MPP. A LAN passou de ser apenas uma interconecção entre computadores e se tornou algo muito mais poderoso tornando-se o coração do CCW.

Visando especificamente a rede como um ítem fundamental para CCW, a indústria de hardware investiu em tecnologias de rede para proporcionar maior desempenho e confiabilidade. Surgiram então placas de rede extremamente poderosas dotadas de processadores e memória próprias, voltadas inteiramente a processamento de rede evoluindo a *Local Area Network* (LAN) para *System Area Network* (SAN).

Mesmo com todo o avanço de hardware os CCW não conseguiam ser melhores que os MPP, mesmo nos casos onde a tecnologia utilizada, tanto em processadores quanto em memória, era muito melhor.

A Divisão de Supercomputação Avançada da Nasa (NASA Advanced Supercomputing Division, também conhecida como, Numerical Aerospace Simulation Systems Division) estudara por anos o desempenho de CCW e MPP. Em um de seus estudos sobre a performance de computação paralela, conhecido como NAS [Wong 2003],

foi constatado que as aplicações utilizadas tanto por CCW quanto por MPPs seriam um dos pontos principais no ganho de desempenho além da tecnologia de hardware presente nesses computadores.

Visando o desenvolvimento de aplicações paralelas de alto desempenho o projeto *SNOW* [Fröhlich 2001] escolheu como objeto de estudo estações de trabalho comuns (PCs) e como barramento de comunicação interfaces de rede e *switches* myrinet [Boden 1995]. Um dos tópicos pesquisados pelo *SNOW* é o impacto do artefato de software desenvolvido e sua relação com o ganho de desempenho.

Nesses sistemas paralelos como *clusters*, o sistema de comunicação passa a ser uma parte vital para as aplicações pois através dele ocorre tanto o compartilhamento de recursos quanto a distribuição de tarefas entre os membros envolvidos no processamento. Sendo que qualquer atraso causado pelo sistema de comunicações, ou por uma dos seus componentes, pode afetar todo o desempenho do sistema computacional.

Sabendo da importância da comunicação para sistemas paralelos este estudo tem como objetivo o aperfeiçoamento do sistema de comunicação do EPOS e o desenvolvimento de um novo membro desse sistema. Este estudo visa tanto encontrar os pontos críticos para no ganho de desempenho quanto o desenvolvimento de novos componentes de software.

Capítulo 2

Estado da Arte

2.1 Cluster Computing

Antes de mais nada é preciso saber para que alguém utilizaria um *cluster* quais os benefícios e os problemas que este tipo de máquina computacional oferece. *Cluster* de Computadores Pessoais possuem, em primeira instância, um custo pequeno para sua concepção, tanto em tempo de planejamento (escolha do hardware ideal, software, refrigeração etc) e sua concepção é rápida pois normalmente é construído com equipamento disponível no mercado. Outro ponto importante é a facilidade de manutenção e expansão de *clusters*. A facilidade de manutenção vem da existência de vários software livres ou pagos que ajudam na automatização do processo além de ser possível desenvolver ferramentas para este fim com grande facilidade. Na sua concepção é possível utilizar diversos tipos de topologia de rede interligados por diferentes tipos de redes com a vantagem da configuração das máquinas poder ser diferente. Por esta razão é possível expandir um *cluster* para qualquer nova tecnologia emergente de forma bastante simplificada e rápida portanto, *clusters* pode sempre utilizar tecnologia de ponta. Na maior parte das vezes o custo-benefício de um *cluster* pode superar o de um supercomputador, tanto em capacidade de processamento quanto custo [Baker 2000]. Em contrapartida os problemas mais comuns estão relacionados a utilização de espaço, a qualidade do software e desem-

penho. Dependendo da quantidade de nodos de processamento utilizados o espaço físico utilizado pode superar o espaço reservado para sua implantação. Outro problema grave é a qualidade do software utilizado, tanto o software de gerenciamento quanto o de processamento. Esses, em sua maioria, são projetos de universidades sendo, normalmente, muito instáveis e que muitas vezes são descontinuados. Em outros casos são software livre e possuem um grau de complexidade elevado que dificulta a sua utilização. Mas um dos maiores problemas é a baixa performance. Dependendo do tipo de hardware adquirido e do tipo de software utilizado pode-se criar vários gargalos e degradar o desempenho.

Portanto, somente a agregação de hardware não é suficiente para que um cluster apresente o resultado esperado. Ao utilizar-se uma tecnologia como essa é necessário saber exatamente o que se espera do supercomputador criado, para que se possa escolher de forma mais eficiente o software e o hardware que será utilizado nesse *cluster*. Mas essa é uma tarefa difícil, pois é necessário tempo para pesquisar tecnologias ou mesmo desenvolver novas. Por esta razão softwares e hardware padronizados são muito utilizados em aglomerados, pois a facilidade de aquisição combinado com uma vasta documentação facilitam o seu emprego e manutenção. Mas a utilização deste tipo de software e hardware apresenta custos quando a maior meta é a performance.

Utilizando-se de hardware comum e com o intuito de utilizar software padronizado diversas pesquisas foram realizadas para encontrar pontos-chave dentro do sistema computacional, que vai desda aplicação até o controlador de hardware, que pudessem afetar de alguma maneira o desempenho do sistema sobre esse hardware, melhorando ou degradando.

2.2 User level networking

No sistema de comunicação, por exemplo, tem-se como meta a máxima utilização da largura de banda da rede (*bandwidth*) com o mínimo atraso possível (*la-*

tency). Entende por atraso, o tempo que leva para a aplicação montar a mensagem, esta passar pelo cerne do sistema, pelo sistema de comunicação e em fim pela rede física e assim chegando ao seu destino em um ouro computador e entende-se por largura de banda a máxima taxa de transferência suportada pela rede, ou seja, quanta informação pode ser enviada ao mesmo tempo pela mesma banda física(Esta é normalmente medida em bits por segundo, *bits per second* ou bits/s). Neste caso tem-se o compromisso de garantir uma melhor performance mantendo os padrões existentes.

Com esse objetivo desenvolveu-se artifícios de software que garantissem os padrões.*User Level Comunication* (ULC) [Geoffray, Prylli e Tourancheau 1999] é uma dessas técnicas que procura melhorar o desempenho simplificando os caminhos entre as operações de emissão e recepção e afeta diretamente a maneira com a qual a aplicação interage com o sistema operacional. Normalmente, para a aplicação poder enviar alguma informação pela rede esta faz uma requisição de emissão ao Núcleo do sistema (Kernel), em seguida o kernel gera uma interrupção e faz a cópia da região do usuário para a região do kernel que em seguida e faz a requisição de emissão para o controlador de hardware gerando outra interrupção.

Através da utilização da ULC procura-se diminuir a interação entre a aplicação e o SO, afetando assim a quantidade de cópias feitas diminuindo-as ao máximo para que tempo necessário para essas operações seja reduzido. Com a ULC reduze-se a intervenção do kernel (chamadas ao sistema (*system calls*), troca de contexto, etc) no processamento de mensagens, diminuindo drasticamente o tempo desperdiçado nessas intervenções. Para deixar mais claro como software melhor projetado pode apresentar maior desempenho, suponha que um acesso a memória em um PC é de 2 ciclos, que a leitura leve 3 ciclos e que a escrita leva mais 4 ciclos. Usando *UPC* para a cópia da região do usuário para região do kernel e *CPB* a cópia da região do kernel para os *buffers* da interface de rede e chamaremos T o tempo gasto no acesso do barramento e na copia dos dados de uma barramento a outro. Para o emissão de uma mensagem teríamos um acesso

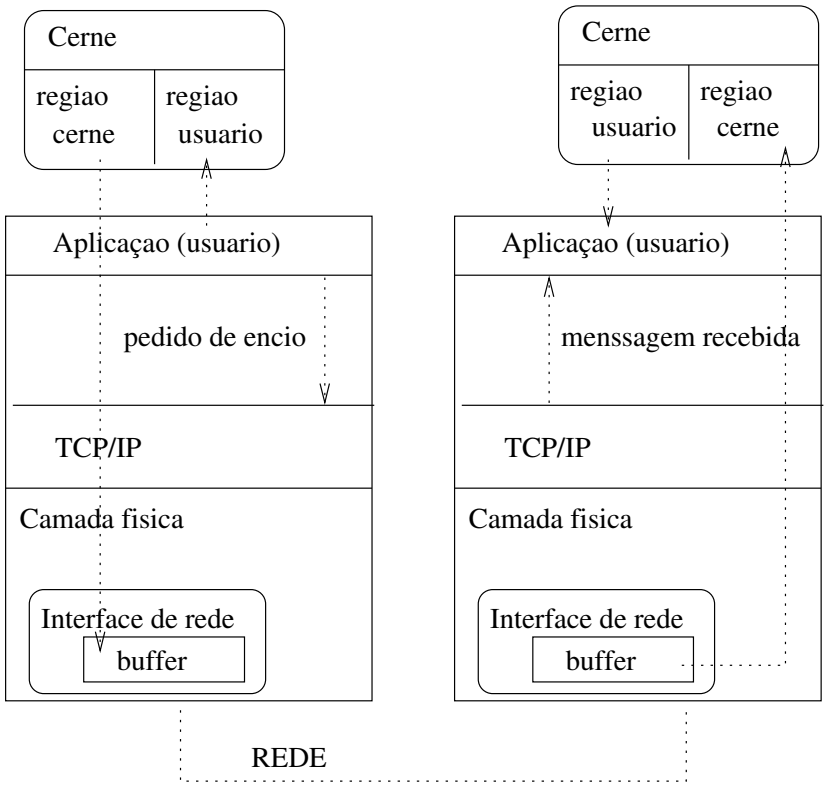

Figura 2.1: Modelo abstrato de TCP/IP

a memória para leitura seria de 2+3 ciclos, uma escrita seria de 2+4 ciclos de clock assim:

$$((UPC+T)+CPB+T) = (((2+3+2+4+T)+2+3+2+4+T) = 19+2T)ciclos.$$

Agora com uma medida de melhora de desepenho adota-se o ULC. O ULC que pula o estágio *UPC* através a utilização de um controlador de hardware (device driver) para este tipo de operação. Este controlador de hardware possui partes mapeadas na memória de endereçamento do usuário interagindo diretamente com a aplicação pulando o estágio de kernel.

Utilizando o artifício de ”pular” o estágio de *UPB* seria possível que a aplicação interagisse diretamente com a interface de rede e assim poderia-se obter um

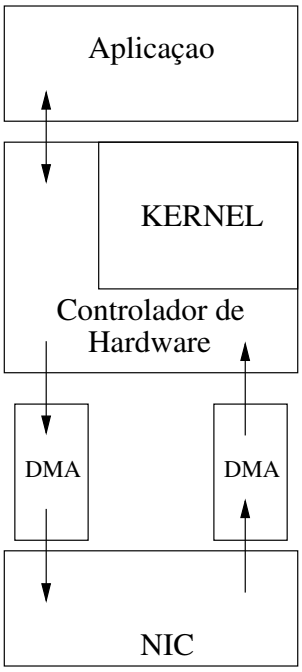

Figura 2.2: Utilização de ULC para diminuir as camadas entre a aplicação e a rede.

ganho de tempo próximo à 57%, pois reduziríamos a quantidade de cópias como segue: $(CPB + T) = ((2 + 3 + 2 + 4 + T) = (11 + T)ciclos$

Outras técnicas desenvolvidas em hardware podem ser utilizadas, como o *write combine*. Em cada cópia, os dados são armazenados numa cache interna e os endereços da cache são marcadas como *write combine*. Quando chega ao limite de 256 bits apenas uma cópia é feita . Esta técnica é mais um exemplo de uma técnica para ganhar tempo.

2.3 Protocolos de Comunicação

Neste capítulo, analisa-se o impacto do protocolo de comunicação no desempenho. Para efeito de ilustração, é feita uma análise da aplicação do protocolo TCP/IP em *clusters*.

A pilha TCP/IP não foi criada para uso em *clusters*. O protocolo IP permite o endereçamento e roteamento em escala global e TCP é um protocolo de transporte complexo com avançado tratamento de perdas de pacotes. Foi desenvolvido numa época em que as redes apresentavam problemas graves de perdas de dados aonde um protocolo extremamente robusto era necessário. O TCP/IP logo se tornou o padrão para a comunicação em rede, tanto locais quanto remotos (World Wide Web). Por ser o mais popular e acessível foi um dos primeiros protocolos de comunicação a ser utilizados em aglomerados. Por simplicidade faremos a comparação entre seus dois protocolos de transporte TCP e UDP.

Em linhas gerais, o TCP não prioriza o desempenho do sistema de comunicação procurando utilizar todas as ferramentas disponíveis para garantir a confiabilidade da transmissão. Na visão do SO, para enviar uma mensagem,por exemplo, a aplicação copia essa mensagem para um espaço de armazenamento temporário (*buffers*)e em seguida, uma requisição é feita para copiar essa mensagem para os *buffers* do sistema, necessitando uma interrupção do sistema. Quando a mensagem chega do outro lado o processo reverso ocorre, o sistema operacional gera uma interrupção e copia a mensagem para os *buffers* do usuário e uma outra interrupção é gerada para informar a aplicação que uma mensagem chegou. Como visto na seção anterior, este tipo de interação com o sistema operacional pode ser por demais custoso ao desempenho. Sem contar

Outro ponto é o tamanho dos cabeçalhos TCP e IP que são de 20 bytes e o UDP é de 8 bytes [Computer Networks 2004]. Para a transmissão de dados com mensagens curtas, até 256 bytes, teríamos uma perda enorme de taxa de transferência apenas por causa desses 40 bytes (TCP+IP). Com *payloads* de 1 a 256 bytes teríamos um perda que vai de 2000% à 7% para o UDP e de 800% à 2%. Existe também um pequeno *overhead* gerado no processamento desses cabeçalhos mas quando algumas centenas de mensagens forem transmitidas pode haver um significativo aumento do tempo de processamento.

Uma maneira de contornar o *overhead* para o processamento dos cabeçalhos e aumentar a taxa de transferência é a utilização de protocolos leves ou *lightweight procotols*.

Um protocolo leve sofre modificações sobre o *standard* no qual ele se baseia (TCP/IP por exemplo) modificando o formato dos dados ou a maneira com que esses são tratados. Se uma determinada rede mostra-se segura o suficiente o protocolo poderia ignora o *checksum* e a confirmação. Caso o atraso fosse a maior prioridade poderia-se ignorar a fragmentação.

Muitas vezes um novo protocolo é criado de forma que possua apenas o necessário para transmitir os dados de um ponto ao outro assumindo características da rede (como baixa taxa de perdas ou ordenação) de forma segura e eficiente, normalmente visando alguma aplicação ou sistema específico. Protocolos leves normalmente estão associados com técnicas de ULC visando melhorar ainda mais o desempenho (mais dados transferidos em menos tempo).

Como exemplos de aplicações desse tipo citar :

Active Messages, "Active Messages [Eicken et al. 1992] is the enabling low-latency communications library for the Berkeley Network of Workstations (NOW) project [Anderson et al. 1995]. Short messages in Active Messages are synchronous, and are based on the concept of a request-reply protocol. The sending user-level application constructs a message in user memory. To transfer the data, the receiving process allocates a receive buffer, also in user memory on the receiving side, and sends a request to the sender. The sender replies by copying the message from the user buffer on the sending side directly to the network. No buffering in system memory is performed. Network hardware transfers the message to the receiver, and then the message is transferred from the network to the receive buffer in user memory. This process requires that user virtual memory on both the sending and receiving side be pinned to an address in physical memory so that it will not be paged out during the network operation. However, once the pinned

user memory buffers are established, no operating system intervention is required for a message to be sent. This protocol is also called a zero-copy protocol, since no copies from user memory to system memory are used. Active Messages was later extended to Generic Active Messages (GAM) to support multiple concurrent parallel applications in a cluster. In GAM, a copy sometimes occurs to a buffer in system memory on the receiving side so that user buffers can be reused more efficiently. In this case, the protocol is referred to as a one-copy protocol."

U-Net, "The U-net network interface architecture [Basu et al. 1995] was developed at Cornell University, and also provides zero-copy messaging where possible. U-net adds the concept of a virtual network interface for each connection in a user application. Just as an application has a virtual memory address space that is mapped to real physical memory on demand, each communication endpoint of the application is viewed as a virtual network interface mapped to a real set of network buffers and queues on demand. The advantage of this architecture is that once the mapping is defined, each active interface has direct access to the network without operating system intervention. The result is that communication can occur with very low latency."

Fast Messages, "Fast Messages extends Active Messages by imposing stronger guarantees on the underlying communication. In particular, Fast Messages guarantees that all messages arrive reliably and in-order, even if the underlying network hardware does not. It does this in part by using flow control to ensure that a fast sender cannot overrun a slow receiver, thus causing messages to be lost. Flow control is implemented in Fast Messages with a credit system that manages pinned memory in the host computers. In general, flow control to prevent lost messages is a complication in all protocols that require the system to pin main memory in the host computers, since either a new buffer must be pinned or an alreadypinned buffer must be emptied before each new message arrives."

BIP, " BIP is a low-latency protocol that was developed at the University

of Lyon. BIP is designed as a low-level message layer over which a higher-level layer such as Message Passing Interface (MPI) can be built. Programmers can use MPI over BIP for parallel application programming. The initial BIP interface consisted of both blocking and non-blocking calls. Later versions (BIP-SMP) provide multiplexing between the network and shared memory under a single API for use on clusters of symmetric multiprocessors. Currently BIP-SMP is only supported under the Linux operating system. BIP achieves low latency and high bandwidth by using different protocols for various message sizes and by providing a zero or single memory copy of user data. To simply the design and keep the overheads low, BIP guarantees in-order delivery of messages, although some flow control issues for small messages are passed to higher software levels." [Baker 2000].

2.4 Processadores de Rede

Como pode ser visto em [Baker 2000] a ethernet foi a primeira tecnologia de rede utilizada em *clusters* mas com o aumento na demanda de largura de banda por partes das aplicações esta mostrou-se pouco adequada. Atualizações para *fast ethernet* e *giga ethernet* aumentaram a capacidade de enviar dados mas mesmo assim apresentam uma pequena degradação de performance em partes devido ao tipo de protocolo de acesso ao meio por estas utilizado, o CSMACD (*Carrier Sense Multiple Access/Collision Detect*), em partes ao meio físico utilizado para a interconecção. Outras propostas para melhorar o desempenho em redes é o*Asynchronous Transfer Mode - ATM* que consiste na utilização de pequenas células fixas de 53 bytes que utiliza canais virtuais que proporcionam serviços orientados a conecção. É possível citar ainda inúmeras tentativas e propostas de tecnologias de rede, mas o objetivo é demonstrar que todas as idéias seguem uma mesma linha que é a de se ter um hardware eficiente extremamente vinculado ao SO sendo que a iteração com este hardware é feito do *Host*, tanto informações de recebimento de dados quanto o envio necessariamente são intermediados pelo *Host*.

Buscando novas tecnologias em hardware em 1993 a ATOMIC [Cohen et al. 1993] propôs uma nova idéia de como modelar hardware de alto desempenho para MPPs. A Myricom foi fundada por membros da equipe de desenvolvimento do ATOMIC e assim toda tecnologia existente no ATOMIC foi levada para a Myricom e em 1995 foi lançada a proposta da Myricom para intercomunicação entre aglomerados.

A proposta da Myricom é uma interface de rede que possua uma maior independência. Em sua proposta a Myricom propõe a utilização de um processador existente no NIC, o LANai, como uma forma de otimizar a troca de mensagens. Utilizando um software que roda no próprio NIC chamado de Myrinet Control Program, que é praticamente um firmware, é possível que a própria placa trate as interrupções decorridas da recepção de mensagens dando a liberdade ao NIC para que este guarde os datagramas e esses sejam acessados num momento mais apropriado, sendo possível também ao NIC tratar o datagrama e transferir os dados diretamente para a aplicação requisitante. Dessa forma é possível utilizar o processador do NIC como uma coprocessador do *Host* sendo que aquele fica inteiramente dedicado à rede diminuindo a sobrecarga sobre este. Uma outra vantagem é a existência de funções implementadas em hardware, como é o caso de *checksum* e filas , por exemplo, que facilitam o desenvolvimento, diminuem o tempo que seria gasto na implementação e geram um significativo aumento no desempenho.

Assim, com o surgimento dos *Network Processors - NPs* ou Processadores de Rede uma nova concepção de desenvolvimento de sistemas de comunicação em geral. Como os NPs apresentam memória própria, um processador rápido (66MHz - 266MHz), autonomia de acesso a memória através da utilização de DMA, funcionalidades importantes à comunicação disponíveis no hardware é possível aos desenvolvedores criarem pilhas inteiras de protocolos e implantá-los diretamente no hardware, possibilitando ao NP executar tarefas que antes eram possíveis somente ao *Host*. A grande vantagem deste tipo de abordagem é a grande abstração que o NIC se torna para o *Host*. Antes o *Host* precisava tanto de um controlador quanto uma ou mais camadas que administrassem

as tarefas relacionadas à rede e sempre que algum evento relacionado ao recebimento de mensagens acontecesse era necessário uma intervenção junto ao NIC. Agora é possível diminuir a interação já que o NP possui uma grande independência para o tratamento de mensagens recebidas (a utilização de DMA possibilita que um dado recebido seja encaminhado diretamente a aplicação requisitante) sendo que o *Host* precisa do NP apenas para enviar dados.

Os NPs apresentam ainda outras vantagens sobre outros tipos de interfaces de rede, como alta largura de banda (1.2GB *full-duplex*) , baixo atraso (menos de 1 nicro segundo), mais de um canal independente de DMA (que possibilita interação com mais de uma aplicação ao mesmo tempo), unidade de transmissão máxima limitada à memória física da NP, custo relativamente barato quando comparado à NICs comuns, *switches* que garantem a disponibilidade do sistema podendo detectar e isolar áreas que venham a apresentar problemas com a comunicação.

2.5 Sistemas operacionais dedicados

Como em todas as áreas da computação aonde o sistema operacional se encontra no centro da execução de tarefas em *clusters* o SO é responsável por fazer o interfaceamento do usuário com o hardware e disponibilizar os recursos existentes de forma eficiente oferecendo um alto grau de confiabilidade. A grande diferença neste caso é a incerteza do que realmente poderá trazer degradação de desempenho, o que realmente está sendo utilizado pelas aplicações e o quão eficiente são os recursos disponíveis. Dessa forma se torna difícil dizer o que deve ser disponibilizado em um sistema operacional para *clusters*.

Em [Baker 2000] há uma visão pontual de como um SO deveria se comportar para este caso: “The ideal operating system would always help, and never hinder, the user. That is, it would help the user (which in this case is an application or

middleware-designer) to configure the system for optimal program execution by supplying a consistent and well-targeted set of functions offering as many system resources as possible. After setting up the environment, it is desired to stay out of the user's way avoiding any time-consuming context switches or excessive set up of data structures for performance-sensitive applications. The most common example of this is in high-speed message passing, in which the operating system pins message buffers in DMA-able memory, and then allows the network interface card, possibly guided by user-level message calls, to transfer messages directly to RAM without operating system intervention. On the other hand, it might be desired that the operating system offer a rich functionality for security, fault-tolerance and communication facilities which of course contrasts with the need for performance that is omnipresent."

O grande problema aqui é a garantia de disponibilidade de recursos garantindo desempenho e confiabilidade.

É possível encontrar outras formas de aproximação do velho problema de desempenho com software genérico e funcionalidades. Em 2001 Fröhlich propôs um sistema operacional adaptável que procura através das técnicas de *Application Oriented System Design*(AOSD)[Fröhlich, Tientcheu e Schröder-Preikschat] aproximar ao máximo o SO das necessidades da aplicação. O problema da utilização de software genérico como solução é a dificuldade em realizar alguns tipos de tarefas, como é o caso do ULC que necessita de uma camada intermediária (*middle-ware*) que possibilite essa tarefa, que exigem abordagens diferentes para serem solucionadas.

O SO resultante, o EPOS, demonstra ser um sistema bastante adequado para ser utilizado em aplicações e sistemas dedicados, como é o caso de *clusters*, pois não seria necessário pagar o preço pela sobrecarga de um sistema genérico, como citado anteriormente, garantindo o acesso aos recursos disponíveis ou somente os necessários além de garantir uma maior confiabilidade do sistema.

Outros problemas podem ser encontrados, como nos tipos de ferramen-

tas disponíveis. Muitas vezes um sistema dedicado ao envio de mensagens nem sempre pode escolher o tipo de gerenciamento de memória , desativar o escalonador de processos ou mesmo garantir o acesso à memória pelo NP. No caso do EPOS é possível desabilitar ou mesmo configurar o sistema às necessidades apresentadas por cada aplicação. Com essa abordagem é possível que haja uma diminuição na sobrecarga (*overhead*) que pode vir a decorrer. Como mostrado em [Fröhlich, Tientcheu e Schröder-Preikschat], é possível diminuir em muito a sobrecarga causada por funções indesejáveis existentes nas várias camadas do SO.

Por essas características o EPOS foi escolhido como a plataforma ideal para *clusters* do projeto SNOW e o sistema de comunicação proposto e implementado utilizará o EPOS como suporte a execução e a AOSD como ferramenta para a construção do sistema de comunicação.

Capítulo 3

Implementação

3.1 Prototipação

Como base para a implementação utilizou-se o sistema operacional Linux por este ser gratuito e apresentar facilidades para o desenvolvimento deste tipo de aplicação. Contudo, para ter certeza que o protótipo desenvolvido no Linux fosse algo muito próximo do que seria o modelo final foi necessário desenvolver desdo controlador de hardware ao sistema de comunicação. Como linguagens de programação utilizou-se C e C++ e como compilador o GCC da GNU.

Para que fosse possível desenvolver o sistema de comunicação era necessário poder utilizar o NIC. Como ferramenta de manipulação de hardware foi desenvolvido um controlador de hardware (*device driver*) utilizando-se a interface do Linux com o kernel e seus dispositivos como mostrado em [Rubini e Corbet 2001]. A necessidade da criação de um controlador de hardware vem do fato deste NIC possuir todos os seus registradores e memória mapeados em memória (*memmory mapped*). Sendo assim, o controlador de hardware é responsável por manipular esses elementos mapeados em memória servindo como uma interface entre o dispositivo, o kernel e o usuário.

Nesta etapa foi possível verificar as interdependências existentes entre o kernel, o usuário e o dispositivo. O kernel mostrou-se ser uma peça fundamental no

sistema de comunicação pois se o sistema de comunicação ficasse dependente do kernel a performance poderia ser comprometida. Tendo em vista todo o atraso decorrente da utilização do kernel em um sistema de comunicação, procurou-se desenvolver o controlador de hardware de forma com que ULC fosse utilizado. Como resultado, o sistema de comunicação "pula"os estágios de cópia que envolvem o kernel e trabalhar diretamente com o NIC e o usuário, diminuindo o atraso e aumentando o desempenho como mostrado no Capítulo 1.

Em seguida, com o controlador de harware pronto, o próximo passo foi desenvolver ferramentas que usando o controlador de hardware executassem a seqüência de inicialização do NIC, como mostrado em [PCI64 Programming]. Basicamente, a inicialização consiste em utilizar estruturas de dados que representam a região mapeada da placa (tanto os registradores quanto a memória) alterando os valores dos registradores de controle do NIC e também manipular o estado do PCI.

```
struct Myrinet{
    SRAM sram;
    Config_Regs config_regs;
    Control_Regs control_regs
    Bank_Pointers bp;
    Special_Regs special_regs
    EEPROM eeprom;
}
```

Figura 3.1: Estrutura de dados utilizada como interface com o NIC

O próximo passo é, utilizando as ferramentas criadas, manipular o núcleo do sistema de comunicações que consiste em algoritmos que utilizam os recursos de hardware disponíveis no NIC, como registradores e DMA, para poder enviar e receber dados.

Tendo a myrinet um processador de rede ela nos proporciona dois modos distintos de operação, podendo tratar os dados,tanto para enviar quanto para receber,

com ou sem a utilização do processador disponível no NIC. Caso ele não seja usado, a myrinet comporta-se como uma interface de rede comum.

Numa primeira abordagem pode-se deixar o processador desligado (*halt*) e a ferramenta para comunicação criada (o sistema de comunicação é nada menos que a combinação de todas as ferramentas criadas, controlador de hardware, inicialização e comunicação), manipula todas as etapas desdo recebimento da mensagem até a sinalização para envio e na segunda utiliza-se um outro software desenvolvido para o processador do NIC que ira dividir etapas com a ferramenta de comunicação. Uma outra opção disponível é a utilização de *DMA* (Direct Memory Access) como substituto ao I/O programado na cópia dos dados de uma região à outra. Com a utilização do *DMA*, o *NIC* adquire autonomia e é capaz de buscar ou colocar dados em uma determinada região de memória (*Busmaster*) que lhe é acessível [Patterson 2000].

Primeiro utilizou-se o *NIC* com o processador desligado e para a cópia dos dados utilizou-se I/O programado. Para que uma operação de envio ocorra, deve-se escolher uma região de memória do *NIC* para ser compartilhada pelo *NIC* e pelo usuário. Esta servirá para armazenar os dados, tanto para envio quanto recepção. Um ponto importante nesta etapa é observar que existe um quadro (*frame*) que irá transitar pela rede e esta não irá permitir a circulação um quadro que não esteja dentro do formato especificado. Assim, torna-se necessária a criação de um quadro para envio. Agora, os dados a serem enviados deverão ser inseridos dentro desse quadro e o quadro é copiado para a região de memória compartilha, pela ferramenta de comunicação, com intermédio do controlador de hardware. Finalmente, com o quadro já na memória da placa, a próxima etapa é o ajuste dos registradores de controle do *NIC*. Estes são atualizados para as posições de memória aonde começa e aonde termina (sendo esse o tamanho total dos dados a serem enviados) a região dos dados e então ajusta-se o registrador de envio de forma a confirmar que os dados daquela região de memória podem ser enviados.

A figura 6 demonstra como esse esquema seria organizado. Os detalhes

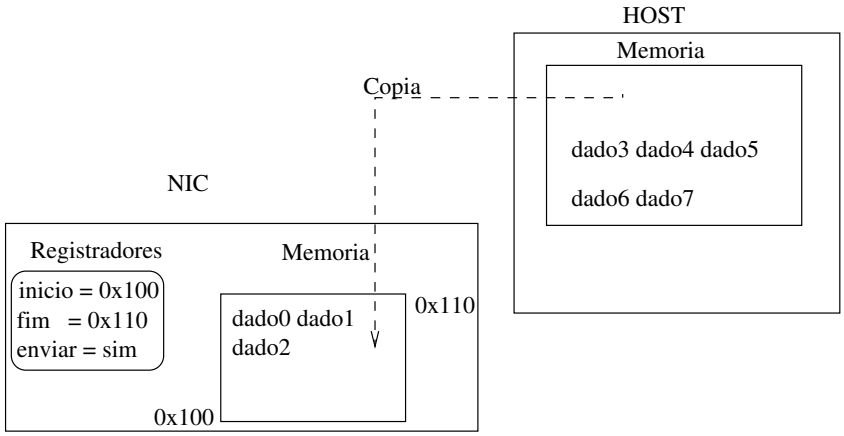

Figura 3.2: Esquema de emissão de um quadro

de organização do quadro (*frame*) serão explicados futuramente na descrição do protocolo.

Quando os dados chegam no destino o processo para receber os dados é semelhante ao do envio. Os registradores de recepção da placa devem ser ajustados para uma região da memória da placa que esteja disponível para receber, sendo este ajuste feito pelo sistema de comunicação, e deve-se informar o tamanho máximo que se espera receber (aonde termina, mesmo processo do emissão de dados) e ao chegarem, os dados são enfileirados nessa região.

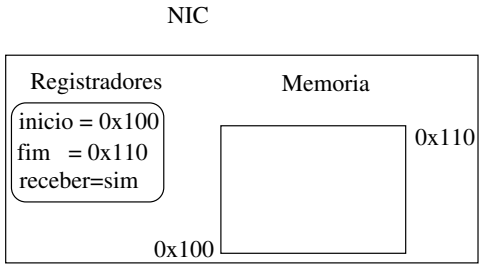

Figura 3.3: Esquema de recepção de um quadro

Agora fazendo algumas modificações no protótipo já existente este poderá combinar tanto as qualidades do hardware existentes, processador de 133MHz, *DMA* etc, quanto metodologias de otimização de software, como *ULC* por exemplo, para

buscar um melhor aproveitamento dos recursos do sistema e assim melhorando o sistema de comunicação através da combinação das qualidades disponíveis nesses dois mundos.

Aqui um ponto importante vale ser comentado que é sobre a escolha e utilização de um processador de rede como um dos componentes do sistema de comunicação. A maior razão para a escolha de um processador de rede é a garantia de maior autonomia do sistema de comunicação, possibilitando o melhor aproveitamento de suas qualidades tanto da rede quando do NIC. Essa autonomia vem do fato de algumas tarefas serem implementadas em hardware como é o caso do *checksum* ou mesmo de filas. Já outras tarefas como cópia de dados, que como mostrado a figura 3, pode ser um gargalo para o desempenho, pode ser feita com a utilização de canais de *DMA*, sendo possível que a própria placa manipule estes canais e a transferência de dados. Outro fator importante é a existência de grande quantidade de memória disponível na placa (mais de 2 Mbytes de memória *SRAM*) [Cohen et al. 1993].

Na figura 10 é apresentado um esquema de como seria a interação entre a aplicação e o MCP.

Como primeiro um passo de otimização é necessário tratar o NIC como um processador de rede e no caso da myrinet, é preciso entender o software de controle (que é carregado na memória da placa e é executado seu processador) pois ele é responsável pelo gerenciamento dos dispositivos disponíveis no NIC.

O processador da myrinet, chamado de LANai, por si só não executa as tarefas que o sistema de comunicação pode delegar ao NIC, ele precisa ser combinado com um software chamado *Myrinet Control Program* - MCP. O MCP é capaz de controlar parte do hardware da placa e interagir com o software de comunicação de várias formas. Por exemplo, um MCP pode organizar os dados a serem enviados (protocolo) , possui a capacidade de enviar e, no caso de ter recebido um datagrama , sinaliza o software de comunicação quando uma mensagens está pronta para ser recebida. Também possui autonomia para copiar dados tanto para o aplicação quanto para o NIC, é capaz de ignorar

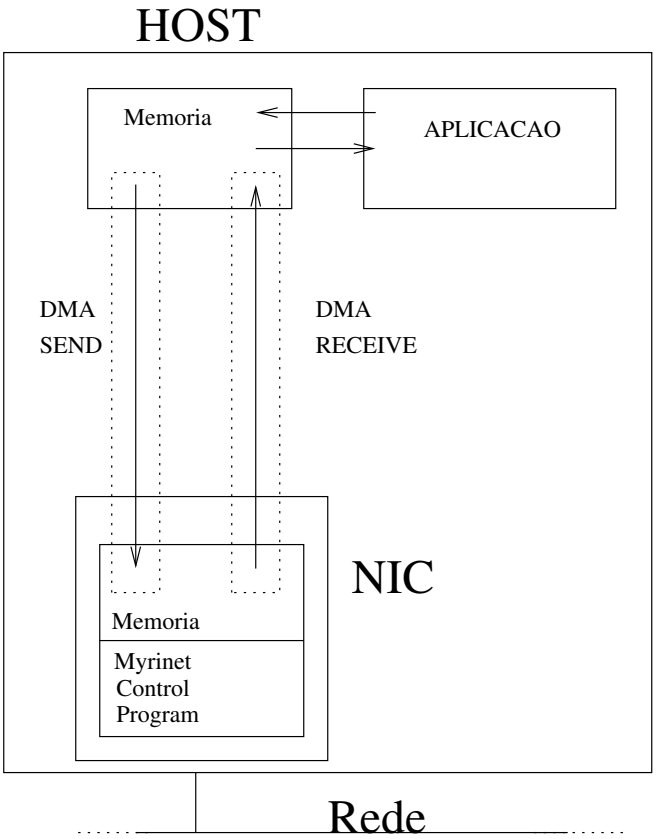

Figura 3.4: Funcionamento de um processador de rede

pacotes que achar defeituosos entre outras inúmeras funções possíveis [Boden 1995].

Seu desenvolvimento acaba sendo bastante demorado pois existe o problema de separar o sistema de comunicação em partes e distribuir essas partes entre o MCP e o software de controle de maneira que a execução dessa parte se dê de forma mais eficiente.

Tendo em vista que essa tarefa pode melhorar o ganho de desempenho um levantamento dos quesitos da aplicação poderia mostrar partes do que o sistema de comunicação que ficaria mais eficiente se implementado no *Host* ou se implementados no NIC. Para tal, precisaria-se fazer um breve estudo dos elementos que compõe o sistema de comunicação, tais como DMA, protocolo etc.

De todos os elementos que compõe o sistema de comunicação, os

seguintes elementos apresentaram condições de serem implementados no NIC ou no *Host*, esses elementos são: transferência de dados com o uso de DMA, protocolo e tarefas relacionadas, validação, envio e recepção.

Para utilizar o DMA como forma de transferência de dados é preciso definir quem deve se preocupar com a transferência se é o NIC ou o *Host*. Como ambos podem fazer DMA dos dados o problema é saber qual o lugar mais apropriado para tal. Uma possível abordagem seria a de deixar somente um dos lados responsável pelas execução das transferências. O grande problema nessa abordagem é o que uma das partes precisaria ficar constantemente verificando se um dos lados quer enviar ou está apto a receber. Poderia-se perder um certo tempo com rotinas para essa verificação. Então como uma solução intermediária optou-se por separar a cópia como segue: Primeiro,o *Host* copia os dados para o NIC quando aquele quiser enviar e o NIC por sua vez, copia os dados que foram recebidos para o *Host*. Essa é uma medida mais objetiva neste caso, pois tanto o *Host* quando o NIC sabem quando é necessário efetuar uma transferência. Assim, as transferências de dados com a utilização de DMA podem vir a ser executadas com um certo grau de liberdade mas indicadores devem ser utilizado para não haver sobrescrita de dados. Já a quebra da mensagem e seu empacotamento (aplicação de um protocolo sobre os dados a serem enviados), é necessário saber em que etapa a mensagem deve ser quebrada e empacotada. A quebra serve para otimizar o fluxo de pacotes que circulam pela rede. Aqui, o problema é saber aonde é mais vantajoso efetuar essas duas operações. Alguns pontos devem ser levantados em conta para tomar essa decisão como, por exemplo, se quebrarmos a mensagem no *Host* e empacotá-la deveremos realizar uma operação de transferência para cada pacote resultante do particionamento dessa forma pode-se perder muito tempo fazendo DMA de pequenos pacotes. Em contrapartida se a quebra e o empacotamento forem feitos dentro no NIC pelo MCP, pode-se fazer cópias maiores e assim pode-se copiar mensagens grandes para a memória do NIC, e o MCP se encarrega de quebrá-la e empacotá-la. A grande vantagem aqui é que efetua-se apenas uma cópia para

cada mensagem mas pode-se perder em tempo de quebra devido a diferença na velocidade do processador, seja qual for o tamanho dela. Se pensarmos na utilização máxima dos recursos disponíveis então a segunda alternativa é mais vantajosa pois pode-se fazer uma melhor utilização da largura de banda do barramento, neste caso 266MB/s, e a quantidade de interrupções geradas para cada mensagens também diminui. O único limitador aqui é o tamanho da mensagem a ser enviada pois essa não poderá ser maior que o tamanho máximo de memória disponível. Mesmo assim uma análise de tempo deveria ser efetuada devido as grandes diferenças de velocidade dos processadores e a utilização de um sistema operacional dedicado. Como forma de verificar isso na prática ambos os modelos foram modelados e desenvolvidos.

Quanto a validação do pacote esta implica em verificações sobre a integridade do pacote decidindo assim se o pacote é aceito ou descartado. Na execução de tal tarefa pode-se criar um gargalo em um ponto do sistema de comunicação que venha a deteriorar todo o desempenho. Aqui é preciso verificar qual a maneira mais eficiente de verificar se um pacote é ou não aceito. Como medida de verificação de erro foi adotado apenas a verificação do tipo de pacote. Como o tipo de pacote é composto por 1 *byte* apenas, assume-se que um erro mínimo na transmissão pode vir a afetar o pacote como um todo danificando esse *byte* e assim ser detectado de maneira simples mas eficiente.Um ponto importante é que uma rede myrinet é praticamente livre de erros. Assim, erros não irão ocorrer frequêntemente e uma verificação muito rigorosa não seria necessária e caso existisse, poderia criar um gargalo.

Por fim é importante decidir quando enviar ou quando receber. Como o envio é uma tarefa mais simples deixa-se a cargo do NIC tomar tal decisão já que ele saberá quando é a hora mais apropriada para essa tarefa fazendo com que o software de controle seja, neste caso, uma interface entre à aplicação e o NIC . Já para o recebimento deve-se levar em conta alguns fatores. Antes de mais nada, é importante verificar o quanto o processo de receber é dependente da aplicação e do software de comunicação. Neste

caso, o software de comunicação serve como uma interface entre a aplicação e o NIC, logo pode-se usar duas abordagens. Na primeira, não há necessidade do software de controle intermediar a transferência dos dados deixando para o MCP o esse encargo com o uso de DMA. Neste caso poderia-se exigir uma certa complexidade da aplicação para tratar as mensagens recebidas por esta razão essa abordagem foi estudada para um uso futuro como um "aspecto" a ser implementado no EPOS.

Numa segunda abordagem pode-se usar o software de controle como intermediário, fazendo com que a aplicação receba a mensagem através de métodos do tipo "receive". Esse método utiliza referências para *buffers* da aplicação evitando a utilização de cópias sendo equivalente a primeira abordagem proposta. Assim o software de controle trata de receber a mensagem e entregar para a aplicação agindo como uma interface para receber os dados.

Para a prototipação foi utilizou-se a segunda abordagem, mas durante a implementação do modelo final tentar-se-á a outra solução para verificar o impacto de cada uma das abordagens na performance.

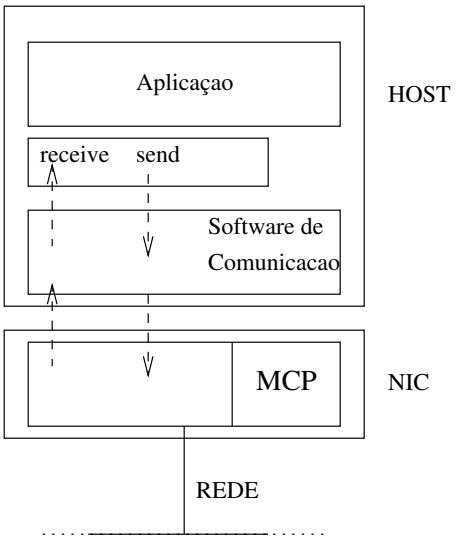

Figura 3.5: Aplicação se comunica com o software de controle que por sua vez se comunica com o MCP

Depois de definirmos as etapas do sistema de comunicação e como se-

riam subdivididos entre o software de comunicação e o MCP, temos o seguinte cenário: O software de comunicação servirá como uma interface entre a aplicação e o NIC oferecendo serviços de “send” e “receive”. O MCP trata as mensagens recebidas empacotando e desempacotando-as, já é responsável pelo protocolo. Ele também faz a validação dos datagramas recebidos e avisará o software de controle sobre a chegada de mensagens.

Aqui termina a etapa de modelagem com a existência de um modelo que servirá como base para o desenvolvimento do membro da família de comunicação que será desenvolvido no EPOS.

3.2 Mediador

Na metodologia utilizada no EPOS, um mediador é a interface entre a aplicação e o hardware, sendo equivalente a um controlador de hardware ou *device driver* de outros sistemas operacionais. O mediador da myrinet para o EPOS é muito parecido com o controlador de hardware que foi desenvolvido para o Linux na etapa de prototipação, com exceção de que esta nova implementação é muito mais simples que a versão do Linux dadas as características do EPOS que facilitam a criação deste tipo de software.

O mediador será responsável por tirar o NIC do estado *halt* e inicializá-la de forma com que este esteja pronto para operar no final da inicialização. Para tal é necessário utilizarmos estruturas de dados que representem os registradores e a memória do NIC que estão mapeados na memória do *Host* como mostrado na figura 6. Estas estruturas de dados são uma representação fiel das unidades de controle e serão utilizadas para realizar todas as tarefas relativas ao NIC, tanto operações de envio e recebimento quanto operações básicas de inicialização quanto DMA. Sua utilização consiste , basicamente, em leitura e escrita de registradores que irão informar o estado atual do NIC e da rede ou irão ajustar um novo estado de execução. Para tal é necessário apenas acessar o campo da

estrutura de dados correspondente ao registrador desejado e então alterar seu estado para um novo estado que seja válido.

```
int Myrinet9::init(System_Info * si){

   PC_PCI pci;
   PC_PCI::Locator loc = pci.scan(VendorID, DeviceID, 0);

   for(Reg32 i=0; i< si−>iomm_size; i++)
       if (si−>iomm[i].locator == (loc.bus << 8 | loc.dev_fn))
         log_addr = si−>iomm[i].log_addr;

   layout = (L9*)log_addr;

    ...
  }
```

Figura 3.6: Processo de localização e mapeamento das estruturas do NIC no EPOS. Utilizadas para fazer com que a estrutura de dados do *Host* seja um espelho dos registradores do NIC

Como forma de exemplificar, para verificar o endereço físico da placa utiliza-se a estrutura de dados que representa os registradores , neste caso “layout”, e acessa-se então o campo referente ao endereço físico, aqui a EEPROM, e então faz-se referência ao campo desejado, como segue: layout->eeprom.board_id.

Com relação a organização do código para que este fosse o mais simples e rápido possível utilizou-se duas técnicas, primeiro, todos os métodos seriam sem retorno e quando o retorno fosse necessário utilizaria-se referência a dados para obter os valores desejados. Por exemplo, toda a parte de inicialização da placa é composta por métodos simples que tem a função de escrever um determinado valor em algum registrador para alterar o estado do *NIC*. Uma maneira de modelar esses métodos é através de métodos sem retorno (*void*). Assim não seria necessário se preocupar com chamadas a funções que poderiam gerar algum atraso. No caso de funções que exigiam retorno, como é o caso do “receive “ e do “get_node_id” optou-se por utilizar referências como parâmetros e métodos sem retorno. Assim poderia-se obter o valor desejado sem , novamente, se preocupar com atrasos causados por chamadas de funções.

O mediador também será responsável pelas tarefas mais importantes do

sistema de comunicação que é o envio e o recebimento de datagramas da rede. Tanto o envio quanto o recebimento dependem da estratégia utilizada para colocar ou retirar os dados nos *buffers* de envio podendo-se escolher entre cópia de dados ou transferência por DMA.

Na etapa de prototipação foram levantados os pontos importantes na escolha de um ou de outro método, por isso nesta etapa irá se fazer apenas a descrição do modo DMA pois o modo de cópia é um subconjunto daquele modo. Utilizando o DMA como forma de transferência de dados precisamos apenas nos preocupar com aonde é mais benéfico que o DMA seja acionado, se do NIC ou do *Host*. Na myrinet, toda a transferência de dados por este método precisa que uma estrutura de dados seja preenchida, com valores como memória de destino, memória de origem, tamanho, etc. sendo que esta estrutura de dados deve estar localizada na memória do NIC. Assim, ao analisar com mais cuidado percebe-se que esta estrutura de dados terá todos os seus campos menos um, o tamanho, modificados apenas uma vez, na sua inicialização. Agora basta saber aonde que seria mais interessante que este dado fosse modificado. Tendo em vista que o *Host* precisa informar de alguma maneira ao NIC qual o tamanho dessa transferência, então não importa quem irá executar o DMA pois o preço para tal é o mesmo em ambos os casos. O único ponto porém é que o se essa execução fosse realizada no *Host* esse precisaria informar ao NIC no seu término, encarecendo a realização de operações de DMA sempre em uma cópia adicional, sendo o NIC portando o lugar ideal para realizar tais operações.

Como citado anteriormente o MCP é um software característico da myrinet que funciona como um *firmware* dinâmico, que é carregado na memória da myrinet em algum determinado momento ,normalmente na inicialização. A função do MCP é servir como uma ferramenta para a manipulação do processador da myrinet assim como dos seus recursos de hardware e como visto no capítulo de prototipação algumas funcionalidades do sistema de comunicação foram escolhidas para serem implementados no MCP. Isso para que fosse possível realizar algumas operações de forma otimizada e

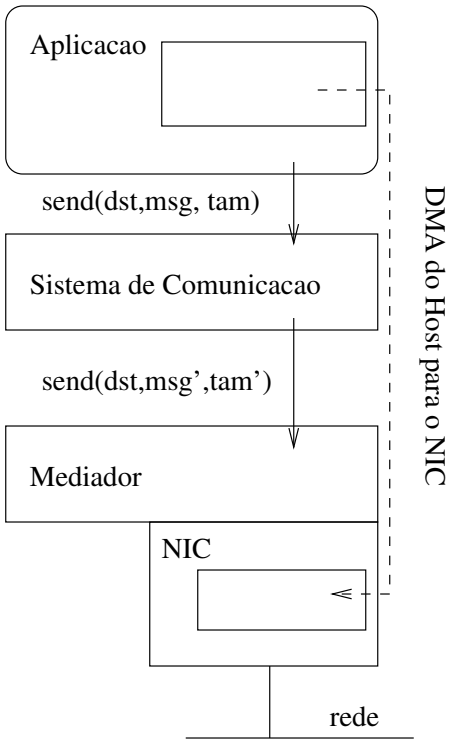

Figura 3.7: Execução de um DMA do *Host* para o NIC

assim ganhar desempenho. No protótipo foram levantadas algumas partes do sistema de comunicação deveriam ser implementadas no MCP e as razões. Agora para o modelo final, deve-se utilizar a base criada no protótipo, julgar suas qualidades e aplicar as modificações necessárias de forma que o desempenho do novo modelo se aproxime ao máximo do almejado.

Quanto à parte de protocolo na etapa de prototipação foi levantado a dúvida de onde a construção do pacote para envio seria mais barato em custos de tempo de execução. Neste caso novamente o lugar ótimo para tal tarefa não foi encontrado mas dois pontos devem ser ressaltados. No caso do *Host* este conta com um processador mais de 10 vezes mais rápido do que o NIC (1900MHz contra 133MHz) e tendo em vista que o EPOS estará rodando como uma aplicação dedicada, ou seja não existe concorrência para a utilização do processador principal,a construção de pacotes no mediador ou no network com sua respectiva transferência poderia otimizar a criação e envio de pacotes. Vale a

pena lembrar que não haverão cópias pois a maioria dos dados são utilizados através de referências e os que não são referências são variáveis constantes não gerando *overhead* adicional. Já no segundo caso, com o protocolo rodando diretamente no NIC, teríamos uma transparência muito maior por parte do mediador, neste caso o mediador ou network aquele seria apenas um controlador de hardware com métodos que sinalizassem o envio e a espera para receber e este tomaria as providências relacionadas com a rede. O NIC receberia os dados "crús" e trabalharia para empacotá-los e desempacotá-los, recebendo e devolvendo apenas dados relativos à aplicação. Como em ambos os casos a quantidade de operações a ser realizada é a mesma vale a pena verificar aonde seria mais rápido para a execução de tais operações.

Um ponto importante a ser levantado é a vantagem que o EPOS oferece para este tipo de aplicação. No caso da myrinet deveríamos ter um endereço lógico para a aplicação que equivaleria a um endereço físico no qual o NIC possa acessar ao ser traduzido pelo SO. Sendo EPOS, como já citado anteriormente, um sistema dedicado, utilizando seus recursos para alto desempenho é possível que o sistema de comunicação utilize DMA para todas as transferências de dados existentes, já que, o sistema disponibiliza recursos de tal forma que cada novo buffer de memória para dados que a aplicação requisitar este já estará apto a ser utilizado para DMA. Isso é possível graças a utilização da memória principal no modo "flat". Assim todos os endereços de memória utilizados pela aplicação já são endereços físicos disponíveis para o NIC, diminuindo o overhead existente em outros modos de mapeamento de memória. Caso o sistema ofereça gerenciamento de memória por paginação também poderíamos utilizar qualquer endereço de memória para DMA, mas seria necessário a tradução de endereços lógicos em físicos acarretando um atraso adicional. Em sistemas genéricos como é o caso do Linux, também existe a possibilidade de termos endereços aptos a DMA mas para que isso seja possível, deve-se fazer invocações ao sistema para alocar *buffers* de DMA e a cada nova mensagem seria necessário que houvesse a alocação de novos *buffers* acarretando em

mais atraso pela invocação ao sistema e pela tradução de endereços físicos em lógicos. Um outro ponto a ser levantado é a necessidade de memória contígua para a utilização de DMA e assim a memória na forma*flat* apresenta a vantagem de que todos os *buffers* alocados nela serão contíguos. Essa é uma outra desvantagem dos outros gerenciamentos de memória como no caso do gerente de memória com paginação já que a alocação de um buffer não resulta, necessariamente, em paginas contíguas podendo resultar em várias páginas espalhadas pela memória.

Uma outra estratégia para acelerar o envio de mensagens é a criação de um *pipeline* de comunicação. Este *pipeline* funciona semelhantemente ao que é usado em processadores com a diferença de são usados datagramas no lugar de instruções e cada estágio do pipeline é uma etapa do processo de envio ou recebimento. Um *pipeline* de rede consiste em uma rotina que dado um tamanho de uma mensagem este analisa um tamanho ótimo para efetuar o particionamento. Para tal utiliza-se uma técnica baseada em cálculos matemáticos e que define o valor ótimo, como mostrado em [Fröhlich, Tientcheu e Schröder-Preikschat].

Esse particionamento deve ocorrer antes da mensagem ser "empacotada" (organizada na forma de um datagrama) pois cada fragmento agora na forma de um pacote irá entrar seqüencialmente nas várias etapas do sistema de comunicação, empacotamento, transferência e envio, de forma a formar um *pipeline* com uma mensagem em cada um dos estágios existentes.

Na etapa de prototipação foi levantada a possibilidade de todas as transferências de dados do *Host* para o NIC serem efetuadas através de DMA. Também levantou-se a possibilidade da quebra e o empacotamento sejam feitos no *Host* e então transferidos. Um ponto importante a ser observado aqui é qual o tamanho ótimo de uma mensagem para que ela realmente ganhe desempenho utilizando DMA. Aqui precisa-se verificar se existe ou não um tamanho mínimo para uma buffer utilizar DMA de maneira eficiente. Como visto em [Prylli e Tourancheau 1997,

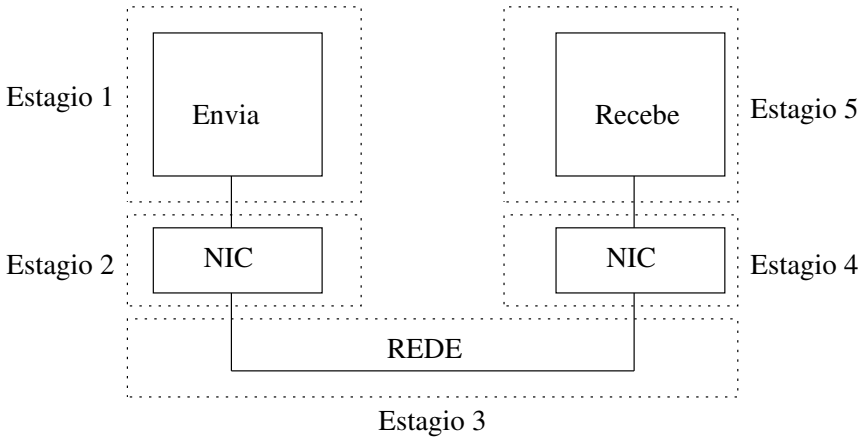

Figura 3.8: Execução de um Pipeline

Frölich e Schröder-Preikschat 2001] para mensagem pequenas, até 256 bits, operações tradicionais de cópia (I/O) oferecem uma menor latência sendo mais adequadas para tratar dados pequenos, já o DMA seria uma ótima opção para diminuir o atraso para mensagens maiores que 256 bits já que as operações de I/O neste caso começam a ter um atraso muito alto.

Tendo em vista que as operações de cópia até 256 bits são vantajosas e que no EPOS qualquer endereço de memória é possível de ser utilizado para DMA não seria mais necessário nos preocuparmos a criação de um pacote no *Host*, como proposto na prototipação, e sua transferência para o NIC. Seria muito mais vantajoso se o header da mensagem, que tem 24 bits, fosse copiada para os *buffers* do NIC e a mensagem, propriamente dita fosse anexada ao final desse header, ou por DMA ou por I/O, dependendo do tamanho dessa mensagem. Dessa forma os datagramas seriam construídos dinamicamente na memória da placa e então transferidos.

Agora chega a parte que envolve o envio e o recebimento de mensagens propriamente dito. Como explicado na prototipação toda a parte de envio e recebimento é feito com base na escrita de registradores. Mais uma vez uma análise se torna necessária para encontrar o local mais apropriado para este tipo de tarefa. Novamente temos duas opções que é a de utilizarmos o *Host* ou o NIC para fazer tal tarefa, precisa-se agora

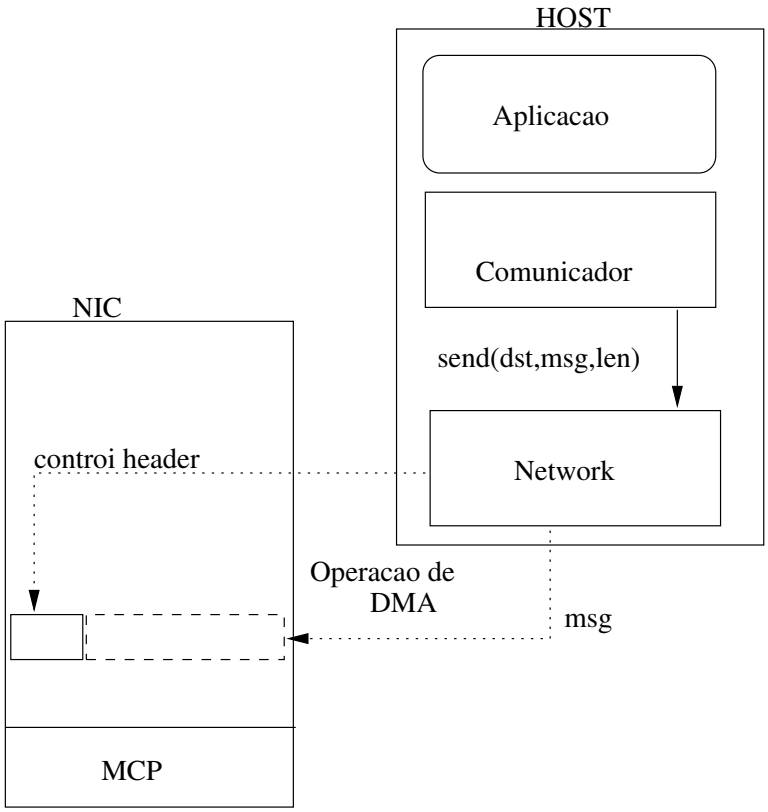

Figura 3.9: Esquema de criação de um pacote utilizando o EPOS

verificar aonde é mais barato realizar essas operações. Para o envio é necessário que haja escrita em 3 registradores (SMP, SA, SMLT como mostrado na etapa de prototipação). O SMP define aonde começa o datagrama, SA indica o alinhamento do cabeçalho de rota e o SMLT inicia a transmissão dos dados e o tamanho total a ser enviado. Como é possível observar o SMP e o SA podem ser sempre constantes, isso se a organização do protocolo assim permitir como é o caso do protocolo definido para este sistema. Dessa forma apenas o disparo da transferência pelo SMLT sofrerá alterações portanto, basta verificar aonde é menos custoso que o SMLT seja ajustado. Se colocado no *Host* para acessar o SMLT ele deverá acessar a região de memória aonde o SMLT se encontra e alterar seu valor sendo necessário acessar o NIC e alterar um valor no caso do SML ser alterado pelo MCP é necessário apenas modificar um valor tornando a operação menos custosa.

Semelhantemente ao processo de envio o recebimento fica mais barato

quando executado como parte do MCP pois ao receber um quadro o NIC irá gerar uma interrupção e um contador é iniciado no NIC. Caso o tempo passe do limite estabelecido o quadro é perdido. Com o MCP rodando no NIC e estado preparado para receber quadros tem-se a vantagem de que ao ser recebido o NIC toma as providências necessárias para o seu recebimento, ou seja,e já possui os *buffers* de recebimento pré-alocados e como o recebimento possui uma prioridade maior que o envio (pelas especificações da myrinet, como pode ser visto em [Myrinet-on-VME Protocol Specification Draft Standard]) o NIC está sempre apto a receber. Caso esta tarefa fosse executada do *Host* teríamos o problema de ter que sempre ler o registrador que indica que um quadro foi recebido, *pooling* e logo que detectado esse recebimento ajustar os registradores através de escritas, o que seria muito custoso já que o NIC através do MCP faz isso praticamente de graça.

Uma outra vantagem de ter o recebimento de dados localizado no NIC é que é possível que ao ser recebido um quadro pode ter o conteúdo relativo a mensagem transferido diretamente para a aplicação. Assim seria possível, por exemplo, a aplicação criar *buffers* de espera e informar ao NIC aonde determinado tipo de mensagem a ser recebida pode ser transferida. Dessa maneira evitaria-se que a aplicação tivesse trabalho adicional para decifrar a mensagem recebida e ajudaria a dar mais agilidade e autonomia ao sistema de comunicação criando uma funcionalidade semelhante à uma mailbox como as utilizadas pelo SO.

3.3 Formalização do Sistema de Comunicação e Protocolo

Quando duas aplicações se comunicam, como as que ocorrem em redes de computadores, é necessário a criação de um mecanismo que possibilite às aplicações envolvidas entender de forma clara o que está sendo transmitido por outra aplicação. RFCs (*Request for Comments*) são criados com a finalidade de definir padrões para proto-

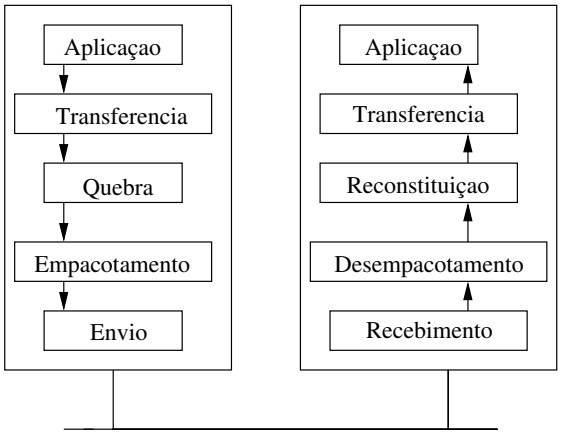

Figura 3.10: Modelo funcional do pipeline

colos de comunicação de dados, através deles usuários e desenvolvedores do mundo todo discutem, aprimoram e fazem as modificações necessárias até o ponto estes se estabilizem e tornem-se um padrão.

Porém, antes de mais nada, devemos ter certeza de que os padrões existentes apresentam as características necessárias e suficientes para a aplicação alvo. Tendo em vista que os modelos convencionais de redes utilizam protocolos extremamente complexos para se comunicar (centenas de aplicações diferentes trocam mensagens entre pontos distantes do mundo criando assim a necessidade de que estas mensagens cheguem ao seu destino com a máxima certeza e exista um mecanismo de retransmissão que se encarregue de completar a mensagem caso ela se perca ou danifique no decorrer do caminho). Sendo assim os protocolos convencionais de rede precisam dotar suas mensagens e protocolos de forma com que qualquer aplicação que queira transmitir dados possa utilizá-las mesmo que, em muitas vezes, apenas uma pequena parte seja suficiente para a comunicação. A complexidade faz com que estes protocolos nem sempre sejam adequados a todos os tipos de aplicação existentes.

3.3.1 Organização dos dados

Ao falar sobre o protocolo aqui proposto é preciso em primeiro lugar levar-se em conta algumas considerações. Primeiro, Myrinet utiliza um sistema de roteamento chamado *wormhole*, o que significa que uma parte da mensagem ,a rota para ser mais exato, será consumida quando passar pelo *switch* e o resto dela será encaminhado ao seu destino; Em segundo ludar existe uma topologia de rede pré-definida, neste caso, conhece-se todos os nós conectados a ela e suas rotas pode-se assim utilizar um protocolo com cabeçalhos de mensagens (*headers*) mais simples e, em terceiro lugar, a simplicidade na "forma" do protocolo pode influenciar em muito no desempenho pois quanto mais complexa for a forma de codificação (entenda-se codificação como a organização dos dados dentro de um pacote) dos dados para o emissão mais tempo será necessário para criar uma mensagem na fonte e para que esta seja decodificada no destino. Assim, como objetivo deste tipo de rede é atingir o máximo de desempenho, opta-se pela simplicidade de suprir apenas as necessidades da aplicação que irá utilizar a rede.

Para a criação do protocolo em primeiro lugar levantou-se as necessidades básicas da rede e das aplicações. Um pacote myrinet (camada 2 do modelo de referência OSI) consiste no seguinte : Aonde o o corpo do

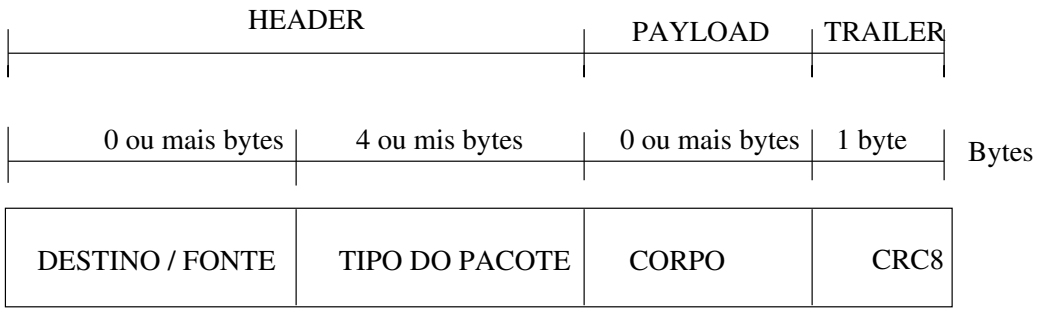

Figura 3.11: Formato de um quadro myrinet

quadro, o *payload*, tem uma unidade máxima de transmissão (Maximum Transmission Unit - MTU) de 2MBytes por quadro pela limitação física da placa, ver

[Myrinet-on-VME Protocol Specification Draft Standard].

Para a composição desta mensagem optou-se por disponibilizar o seguinte:

- rota : destino;
- cabeçalho: fonte, tipo da mensagem, tamanho total da mensagem;
- corpo : parcela do total que esta sendo enviado, mensagem.

Como a rota é importante apenas para quem está enviando e esta será consumida pelo *switch*. Logo uma mensagem de emissão é composta por rota, cabeçalho e corpo, enquanto que uma mensagem no receptor é composta apenas por cabeçalho e corpo. A rota consiste em uma porta válida no *switch*. Já no cabeçalho tem-se: fone que é a informação de quem emissão, o tipo da mensagem para identificar o software que deverá manusear o pacote, tamanho total da mensagem para fins de controle. E, por fim, corpo que consiste na mensagem, ou parte dela, propriamente dita. O corpo possui dados que são importantes apenas para a aplicação requisitante (camada 3 do modelo de referência OSI), sendo assim, esta aplicação é responsável pela organização dos dados no corpo da mensagem.

3.3.2 Formalização

Quando precisa-se criar uma nova linguagem é necessário que esta seja criada de uma maneira com que não apresente problemas a seus usuários uma maneira formal, como foi citado na introdução deste capítulo. O mesmo se aplica à protocolos de comunicação. No caso deste estudo, os protocolos convencionais não satisfazem os requisitos levantados para este tipo de rede, sendo necessário a criação de um protocolo próprio. Para tal é preciso que seja feita sua formalização para seja possível evitar problemas como ambiguidade, laços de repetição (*loops*), estouro de pilha (*buffer overflow*)

entre outro. Para a formalização foi utilizada a ferramenta formal conhecida como Redes de Petri.

Redes de Petri [Cardoso e Valette 1997] foram criadas por Carl Adam Petri em 1962 na Universidade de Darnstadt , na alemanha. É uma ferramenta gráfica e matemática que pode ser adaptada a uma grande diversidade de aplicações com o propósito de analise, avaliação de desempenho e validação formal de sistemas discretos. Mesmo esse metodo formal apresentando limitações como a não existencia de uma maneira de simular todos os estados possíveis das variáveis envolvidas no processo ainda assim é possível verificar o comportamento do sistema para configurações previstas. Sendo esta ultima condição suficiente para formalizar o sistema proposto.

A formalização poderia ser efetivada atravéz de testes exaustivos de troca de menssagens, mas como poderia-se garantir que os testes iriam cobrir todas as possibilidades da máquina de estados?

Asim, para verificar as propriedades da rede que garantem a confiabilidade do protocolo foi utilizada a ferramenta de modelagem PEP [group 2004] e a de simulação PIPE [Bloom et al. 2004]. A escolha dessas ferramentas foi feita com base nos quisitos de disponibilidade e eficiência. Não fora levado em conta a latência nas tarefas de cópia (IO), acesso direto a memória (DMA) e acesso a rede pois o proposito deste é a procura de possíveis problemas e não a da verificação do desempenho pois este requer a adição de diversos fatores o que tornaria a simulação e a análise demais complexas. Tanto a análise estrutural quanto a do modelo foram separadas em duas partes. Primeiro fora simulado o envio e o recebimento de menssagens de forma separada e em seguida montou-se o modelo completo da rede com a inclusão do *switch* e da rede.

3.3.3 O resultados da simulação e análise da rede

Send

- Propriedades do modelo

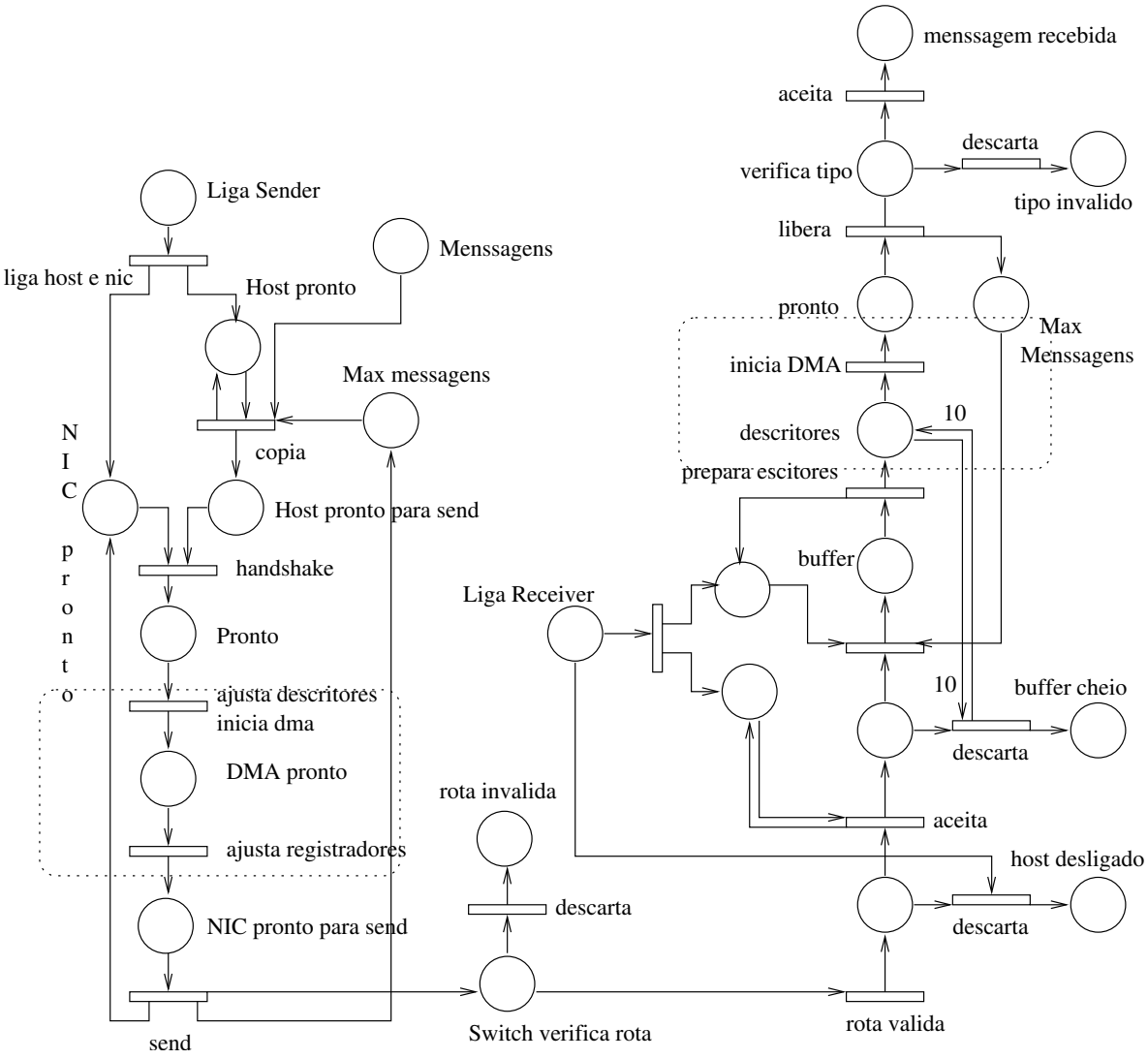

Figura 3.12: Sistema de comunicação no formato da Rede de Petri

- Modelo é simples e estendido
- Livre escolha simples e estendida
- Todos os estados são alcançáveis

- Propriedades estruturais
  - limitada
  - possui *deadlocks*

  - não possui lugares k-ilimitados

- Analise das Propriedades
  - nenhum T-Invariante
  - possui 8 P-Invariantes positivos

Receive

- Propriedades do modelo
  - Estendido
  - livre escolha estendida
  - todos os estados são alcançáveis
- Propriedades estruturais
  - limitada
  - possui *deadlocks*
  - não possui lugares k-ilimitados
- Analise das Propriedades
  - nenhum T-Invariante
  - nenhum P-Invariante

Rede (Envio, Recebimento e adição da rede)

- Propriedades do modelo
  - simples
  - livre escolha estendida
- Propriedades estruturais

  - limitada
  - não possui *deadlocks*
  - não possui lugares k-limitados

- Analise das Propriedades
  - nenhum T-Invariante
  - 21 P-Invariantes positivos

Ao analisar os resultados obtidos podemos concluir:

- No Envio e no Recebimento o deadlock é esperando pois não existe uma conecção com a rede;
- A não existência de estados k-limitados é esperada em alguns estados, como *buffers* por exemplo;
- A livre escolha estendida nos mostra que existem conflitos efetivos, isto é, sempre que chegarmos a um lugar onde existam dois caminhos precisar-se-a escolher entre um deles.
- Os P-Invariantes são condições necessárias mas e suficientes para julgar a alcançabilidade da rede;
- Os P-Invariantes não cobrem todos os lugares, pois o sistema não foi projetado para ser conservativo, tendo em vista que esse é um sistema de troca de mensagens já da quantidade de mensagens a serem enviadas muda com o passar do tempo.
- A não existência de T-Invariantes está ligada com a não reinicialização do sistema.

Por fim, a não existência de *deadlocks*, a limitação da rede, a alcançabilidade e os conflitos efetivos são condições suficientes para garantir que o protocolo assim como o sistema de comunicação que o envolve são funcionais e estão formalizados.

3.4 Diagramas de Classe

A criação de um novo membro para a família de Network do EPOS vem do fato de que este utiliza um alto grau de decomposição dos aspectos relativos ao SO. Os elementos derivados dessa decomposição são então agrupados por semelhança e desses grupos vem as famílias [Fröhlich, Tientcheu e Schröder-Preikschat].

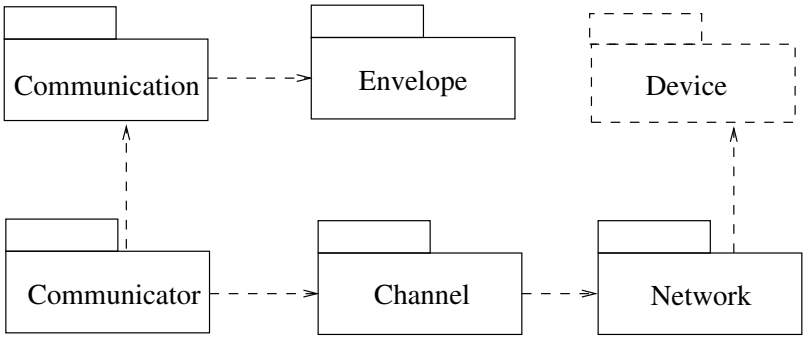

Figura 3.13: Familias de abstrações de comunicação entre processos

Por exemplo, Communicator possui membros responsáveis pela comunicação fim a fim, enquanto que o Channel membros encarregados de estabelecer canais de comunicação entre dois pontos, podendo utilizar Network como uma interface para comunicação com a rede. Já os membros do Network ,por sua vez, utilizam o mediador de hardware representado por Device para se comunicarem com a rede.

Olhando mais detalhadamente a família de Network podemos perceber que esta é formada por vários tipos de rede.

Um dos tipos de rede existentes para esta família é redes do tipo myrinet, assim sendo, ao desenvolver toda uma estrutura para suportar um novo tipo de myrinet o que se está fazendo é nada menos que incorporar um novo membro a esta família.

Todos os membros de uma mesma família se apresentam para o mundo através de uma interface comum chamada de *Inflated Interface* ou interface inflada.

Abaixo seguem os diagramas de classes do membro Myrinet9 para

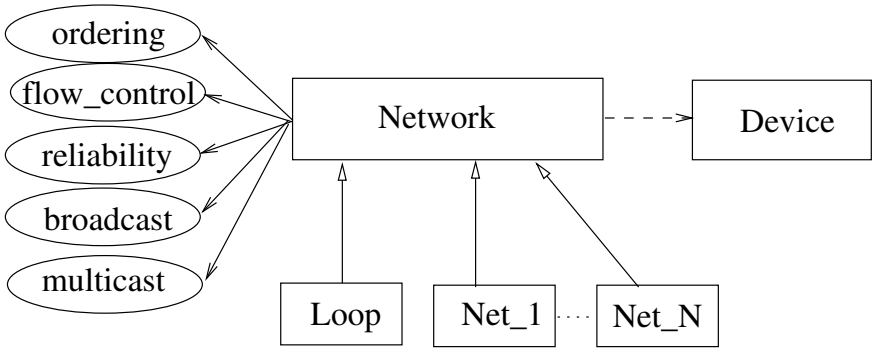

Figura 3.14: Familia de Network e seus membros

EPOS. O membro Myrinet9 é representado pela classe com o respectivo nome, PC_PCI é a interface que o EPOS disponibiliza para acessar o barramento PCI e Basic_Network é a interface comum a todos os membros da família de rede que serão visíveis ao Channel ou o Communicator. .

Network
+ send(dst : Node_id, data : void *, size : unsigned int, id : unsigned short) : void + receive() : void + registerProtocol(type : unsigned short , buff : void *) : void

Figura 3.15: Diagrama da interface inflada

No caso da myrinet ainda teremos uma dependência estabelecida com o PCI. Esta dependência é fundamental para que seja possível utilizar o dispositivo anexado ao controlador de PCI.

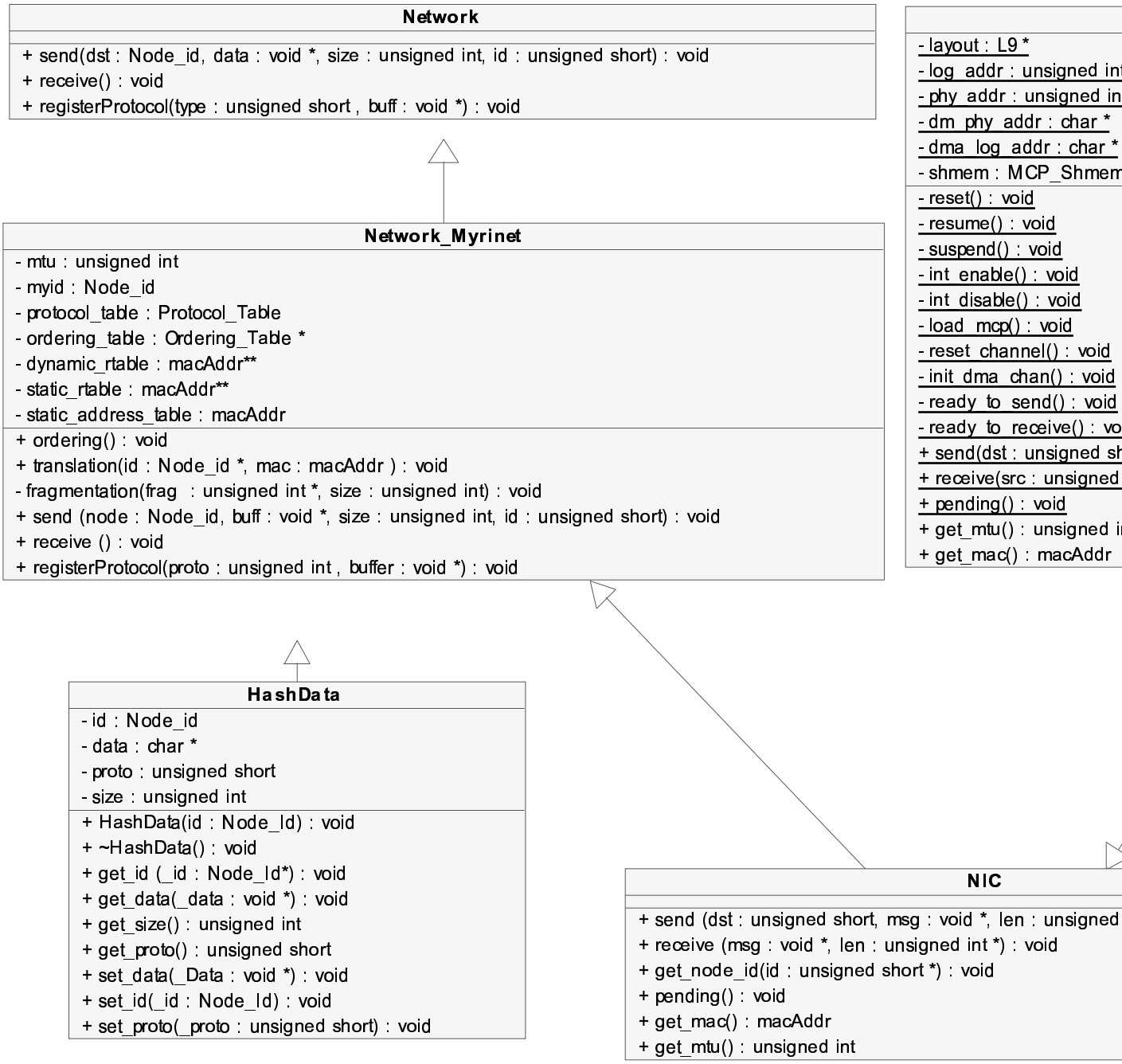

Figura 3.16: Diagrama de dependências

Capítulo 4

Medidas de Performance

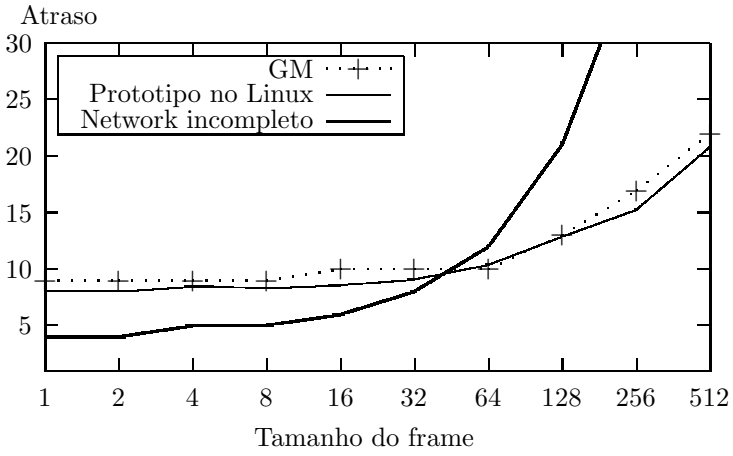

Figura 4.1: Performance com o sistema de comunicação proposto na troca de mensagens.

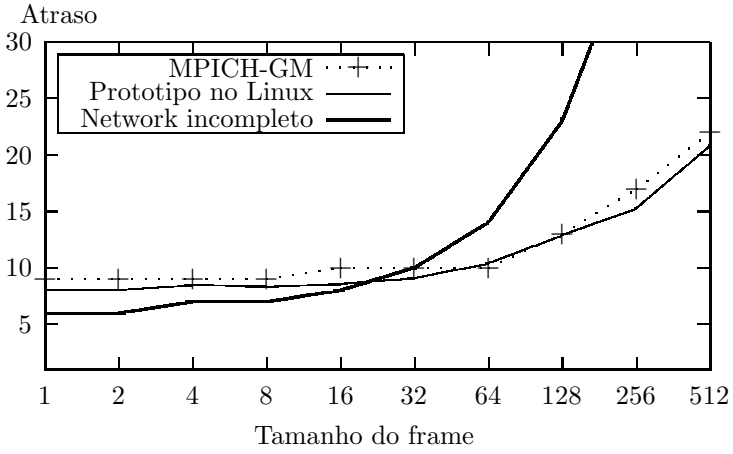

Figura 4.2: Performance com a utilização de uma aplicação.

Capítulo 5

Trabalhos Futuros

Como este trabalho é parte integrante de um sistema de comunicação que envolve uma biblioteca paralela, a *Message Passing Interface* (MPI) existe bastante modificações no sistema de comunicação que podem ser feitos para diminuir o atraso na troca de mensagens da MPI [Sanches 204]. A princípio a idéia mais próxima seria verificar qual o suporte real que o processador LANai apresenta para aplicações, delegando a ele mais tarefas e tarefas mais complexas como o tratamento de cabeçalhos de mensagens MPI, que foi citado no capítulo de implementação. Para dar uma melhor visão dessa idéia imagine o seguinte cenário. Toda mensagem MPI é enviada com algum trecho de código a ser processado em outro nodo, um *Header* MPI é criado como forma de controlar as mensagens recebidas e saber a qual parte dos dados que estão sendo processados ela pertence. Normalmente, ao chegar o Network entrega para a MPI um pacote MPI, com *Header* e dados. O header recebido é comparado com outros *Headers* em espera até encontrar um acerto, ai então os dados são tratados. Os demais que não estivessem sido requisitados, os sem um *Header* acerto, são armazenados para uso futuro.

Agora caso se fosse possível colocar toda a resolução desse *Header* diretamente no Network, mais precisamente no NIC rodando como parte do MCP. Ao enviar uma mensagem MPI, a própria MPI se encarrega de "anotar" nos *buffers* da myrinet que existe um *Header* pendente e que quando esse *Header* tiver um acerto utilize os canais

de DMA existentes no NIC para transferir os dados do pacote MPI diretamente para a aplicação em um buffer reservado para aquela mensagem.

Em termo de performance teríamos um grande ganho teórico pois existiriam 3 operações muito rápidas envolvias: uma cópia para o NIC de 16 bytes, uma execução de DMA e uma escrita no buffer da MPI para informá-la do recebimento de uma mensagem, contra 3 do modelo anterior: transferir os dados do Network para a MPI, comparação de cabeçalho e cópia do conteúdo.

Existem vários outros pontos relativos a MPI que poderiam ser explorados nesta mesma idéia mas que precisam de um maio amadurecimento.

Outra boa opção para um trabalho seria a utilização do processador da myrinet, o LANai como uma nova arquitetura para utilização do EPOS. Poderia-se utilizar o EPOS no lugar do MCP como um SO dedicado a rede.O EPOS teria recursos para um micro-kernel e este seria capaz de dar uma grande autonomia para a interface de rede, fazendo com que tarefas relativas a rede sejam todas resolvidas no NIC. A grande vantagem do EPOS sobre o MCP seria a de uma maior utilização dos recursos do hardware, como por exemplo a myrinet dispões de quatro canais de DMA, utilizando-se quatro emphthreads seria possível deixar uma dedicada para transferência de dados relativos a envio e outra para recebimento de dados. Outras duas poderiam ser usadas para cuidar de operações de envio e recebimento ou mesmo organização de filas. Sem contar tarefas típicas que o MCP poderia executar como tratamento de protocolos, resolução de *Headers* MPI, entre outros.

Capítulo 6

Conclusão

Clusters de computadores pessoais são uma ótima opção para substituir os supercomputadores em relação a custo benefício. A qualidade excepcional do hardware disponível no mercado e que pode ser utilizado em *clusters* não garante a performance desejada. Como foi visto no princípio a utilização de software padronizado e de fácil aquisição se torna um problema e não uma solução. É necessário também que haja uma análise maior dos quesito do hardware para que este software possa utilizar todos os recursos disponíveis de forma mais adequada.

Técnicas aplicadas a software padronizado ajudam a garantir o ganho de performance e a manter uma interface comum entre aplicações, mas essas otimizações possuem seus limites. A mudança de paradigma de desenvolvimento pode trazer novas perspectivas a essa área. A AOSD e o EPOS mostram que é possível desenvolver software básico de altissímo nível e que se, desenvolvido de forma correta, podem garantir a qualidade do produto final. Utilizando a organização em famílias é possível que haja a escolha somente dos membros de famílias que satisfaçam as necessidades da aplicação e dessa forma é que a AOSD garante a qualidade do software gerado.

Outro grande problema encontrado está no que diz respeito aos protocolos utilizados em *clusters*. A utilização de protocolos padronizados traz a vantagem da facilidade mas em compensação traz problemas sérios no que diz respeito a performance.

Uma escolha mau feita pode acarretar numa degradação do sistema. A escolha de protocolos corretos ou o desenvolvimento de novos modelos visando aplicações específicas tornam-se as melhores opções.

O hardware utilizado em *clusters* vem se tornando cada vez mais complexo e eficiente. Com a modernização das interfaces de rede em processadores de rede e com a otimização de determinadas tarefas em hardware é possível utilizar os recursos do NIC e da rede de maneira mais eficientes e dando tanto ao sistema de comunicação quanto à aplicação maior liberdade na escolha dos métodos a serem utilizados na troca de mensagens.

Sendo a myrinet um processador de rede esta oferece várias vantagens em relação à outros tipos de rede. Uma de suas grandes vantagens é a utilização de um processador e memória disponíveis no NIC ajudam nas tarefas de envio e recebimento aumentando ainda mais o desempenho do sistema. Mas delegar todas as tarefas ao NIC também não é a melhor solução, deve-se sempre ser levantado as vantagem de utilizar um coprocessador para rede e seus limites.

Assim,com a combinação dos fatores hardware de alta qualidade, sistemas operacionais, protocolos e de técnicas de engenharia de software é possível a construção de um ambiente mais adequado para *clusters* de maneira com que sistema computacional esteja voltado inteiramente para a tarefa escolhida e que possibilite à aplicação uma melhor interação com o sistema, podendo trazer resultados extremamente satisfatórios comparáveis ao desempenho que os supercomputadores apresentam.

Referências Bibliográficas

[Anderson et al. 1995]ANDERSON, T. E. et al. A case for networks of workstations: Now. *IEEE Micro*, fev. 1995.

[Baker 2000]BAKER, M. *Cluster Computing White Paper*. [S.l.], 2000.

[Basu et al. 1995]BASU, A. et al. U-net: A user-level network interface for parallel and distributed computing. In: . [s.n.], 1995. Disponível em: <citeseer.nj.nec.com/eicken95unet.html>.

[Bloom et al. 2004]BLOOM, J. et al. *PIPE - Open Source Petri Net Editor and Analyzer*. february 2004. Disponível em: <http://petri-net.sourceforge.net/>.

[Boden 1995]BODEN, e. a. N. J. Myrinet – a gibabit-per-second local-area network. *IEEE MICRO*, 1995.

[Cardoso e Valette 1997]CARDOSO, J.; VALETTE, R. *Redes de Petri*. [S.l.]: Editora da UFSC, 1997.

[Cohen et al. 1993]COHEN, D. et al. *ATOMIC: A High–Speed Local Communication Architecture*. 1993. Disponível em: <citeseer.ist.psu.edu/cohen94atomic.html>.

[Computer Networks 2004]COMPUTER Networks. [S.l.]: Prentice Hall PTR, 2004.

[Eicken et al. 1992]EICKEN, T. von et al. Active messages: A mechanism for integrated communication and computation. In: *19th International Symposium on Com-*

puter Architecture. Gold Coast, Australia: [s.n.], 1992. p. 256–266. Disponível em: <citeseer.nj.nec.com/eicken92active.html>.

[Fröhlich 2001]FRöHLICH, A. A. Application-oriented operating system. In: ____. [S.l.]: GMD Reseach Series, 2001. cap. 5, p. 149–166.

[Fröhlich, Tientcheu e Schröder-Preikschat]FRöHLICH, A. A.; TIENTCHEU, G. P.; SCHRöDER-PREIKSCHAT, W. *EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations*. Disponível em: <citeseer.nj.nec.com/502253.html>.

[Frölich e Schröder-Preikschat 2001]FRöLICH, A. A.; SCHRöDER-PREIKSCHAT, W. Component-based communication support for parallel applications running on workstation clusters. In: . [S.l.]: Fourth International Workshop on Advanced Parallel Processing Technologies, 2001. Ilmenau, Germany.

[Geoffray, Prylli e Tourancheau 1999]GEOFFRAY, P.; PRYLLI, L.; TOURANCHEAU, B. Bip-smp: High performance message passing over a cluster of commodity smps. 1999.

[group 2004]GROUP, H. *PEP - Programming Environment based on Petri Nets*. february 2004. Disponível em: <http://parsys.informatik.uni-oldenburg.de/ pep/>.

[Myrinet-on-VME Protocol Specification Draft Standard]MYRINET-ON-VME Protocol Specification Draft Standard. [S.l.].

[Patterson 2000]PATTERSON, J. L. H. D. A. *Organização e Projeto de Computadores - A interface HARDWARE/SOFTWARE*. [S.l.]: Morgan Kaufmann Publishers, Inc, 2000.

[PCI64 Programming]PCI64 Programming. [S.l.].

[Prylli e Tourancheau 1997]PRYLLI, L.; TOURANCHEAU, B. A new protocol design for high performance networking: a myrinet experience. 1997.

[Rubini e Corbet 2001]RUBINI, A.; CORBET, J. *Linux Device Drivers*. [S.l.]: O'Reilly, 2001.

[Sanches 204]SANCHES, A. L. G. *Implementação de MPI Orientada às Aplicações*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 204.

[Wong 2003]WONG, R. F. V. d. W. P. Nas parallel benchmark i/o version 2.4. In: *NAS Technical Report NAS-03-002*. [S.l.: s.n.], 2003.

Capítulo 7

Anexos

7.1 Anexo a - Código Fonte

7.1.1 myrinet9.h

```
// EPOS Myrinet9 Declarations
//
// Author: secco
// Documentation: $EPOS/doc/nic   Date: 17 Feb 2004

#ifndef __myrinet9_h
#define __myrinet9_h

#include <nic.h>
#include "../common.h"
#include "mcp.h"
#include "mcp_shmem.h"

__BEGIN_SYS
//-------------------------------
// Myrinet9 CLASS DECLARATION
//-------------------------------

struct MCP_Packet_Header {
```

```
  unsigned short type;
  unsigned short source;
  int length;
};
//typedef struct _MCP_Packet_Header MCP_Packet_Header;

struct  MCP_Packet {
  MCP_Packet_Header header;
  unsigned int burst;
  unsigned int unused;
  char body[];
};
//typedef struct _MCP_Packet MCP_Packet;

struct MCP_Datagram{
  char route[8];
  MCP_Packet_Header header;
  unsigned int burst;
unsigned int unused;
  char body[];
};
//typedef struct _MCP_Datagram MCP_Datagram;

class Myrinet9: public __INT(Myrinet9), protected NIC_Common
{
private:
    typedef Traits<Myrinet9> Traits;
    static const Type_Id TYPE = Type<Myrinet9>::TYPE;

    // Myrinet9 private imports, types and constants

public:
    Myrinet9();
    ~Myrinet9();

    //world interface
```

```
    static void send(Node dst,  void * msg, Reg32 len);
    static void receive(Node * const src, void * msg, Reg32 * const len);
    static void get_node_id(unsigned short * const id);

    static void net_dma(Node dst,  void * msg, Reg32 len);
    static void host_dma(Node dst,  void * msg, Reg32 len);

    static void host_receive(Node * const src, void * msg, Reg32 * const len);


    static int init(System_Info *si);

private:
// Myrinet9 implementation methods
    static void reset();
    static void resume();
    static void suspend();
    static void int_enable();
    static void int_disable();
    static void load_mcp();
    static void reset_channel();
    static void init_dma_chan();
    static void best_size(Reg32 * const size , Reg32 len) ;

    static void ready_to_send( );
    static void ready_to_receive( );

static void print();
static void printshm();
static void printdtg();
  static void printdd();

    static void handshake();
    static void delay ();
    static void set_shmem();
```

```
static void set_node_id();

static void build_datagram(Node dst,  Reg32 len, Reg32 burst);

static void trigger_send(Reg32 burst);


// Myrinet9 attributes
static L9 *layout;
static Reg32 log_addr;
static Reg32  phy_addr;
static char * dma_phy_addr;
static char * dma_log_addr;
static Reg32* send_buff[2];
static Reg32* recv_buff[2];
static MCP_Shmem * shmem;//<

static MCP_Datagram * dtg;
static DMA_Descriptor * desc;

static Node node_id;
static const unsigned long n_nodes;
static Myrinet_Address address_table[8] ;           //number of nodes
static const char routing_table[8][8] ;
};

__END_SYS

#endif
```

7.1.2 myrinet9.cc

```
// EPOS Myrinet9 Implementation
//
// Author: secco
// Documentation: $EPOS/doc/nic   Date: 17 Feb 2004

#include <mediator/nic/myrinet9/myrinet9.h>
#include "mcp/mcp_executable.h"
#include "getus.h"

__BEGIN_SYS

//before we use a static var, we have to redeclare it.
L9 * Myrinet9::layout;
Reg32 Myrinet9::log_addr;
Reg32 Myrinet9::phy_addr;
char * Myrinet9::dma_phy_addr;
char * Myrinet9::dma_log_addr;
Reg32 * Myrinet9::send_buff[2];
Reg32 * Myrinet9::recv_buff[2];
MCP_Shmem * Myrinet9::shmem;
Node Myrinet9::node_id  ;

MCP_Datagram * Myrinet9::dtg;
DMA_Descriptor * Myrinet9::desc;

const unsigned long Myrinet9::n_nodes =8;

Myrinet_Address  Myrinet9::address_table[8] = {
  0x60dd7f5d8dLL,
  0x60dd7f5dbcLL,
  0x60dd7f5d5eLL,
  0x60dd7f5d91LL,
  0x60dd7f5dbeLL,
  0x60dd7f69ebLL,
```

```
  0x60dd7f5d93LL,
  0x60dd7f5dbdLL,
};

// ROUTING TABLE is equivalent to this : ->  ( rota  & 0x3f) | 0x80
const char Myrinet9::routing_table[8][8] = {
  {    0,  -127, -126, -125, -124, -123, -122, -121},
  { -65, 0, -127, -126, -125, -124, -123, -122},
  { -66, -65, 0, -127, -126, -125, -124, -123},
  {-67,-66,-65,0,-127,-126,-125,-124},
  {-68,-67,-66,-65,0,-127,-126,-125},
  {-69,-68,-67,-66,-65,0,-127,-126},
  {-70,-69,-68,-67,-66,-65,0,-127},
  {-71,-70,-69,-68,-67,-66,-65,0},
};

// Class attributes
// type Myrinet9::attribute;

// Constructors
Myrinet9::Myrinet9()
{
 db<Myrinet9>(TRC) << "Myrinet9()\n";

//  db<Myrinet9>(TRC)  <<"dma logical address : "<< (void*)dma_log_addr << "\n";
//  send_buff[0] = dma_log_addr;
//  db<Myrinet9>(TRC)  <<"send buffer logical address : "<< (void*)send_buff[0]  << "\n"
//  send_buff[1] = send_buff[0] + MCP_PACKET_LEN;
//   recv_buff[0] = send_buff[1] + MCP_PACKET_LEN;
//  recv_buff[1] = send_buff[2] + MCP_PACKET_LEN;
//  db<Myrinet9>(TRC) << "done constructor\n";

dtg  = (MCP_Datagram*)   &layout->sram[SEND_DMA_OFFSET];
desc  = (DMA_Descriptor *) &layout->sram[DMA_DESC_OFFSET];
```

```
}

Myrinet9::~Myrinet9()
{
db<Myrinet9>(TRC) << "~Myrinet9()\n";
}

// Class methods
void Myrinet9::reset(){
  Reg32 bus_rate = 0;
  Reg32 clockval = 0;
db<Myrinet9>(TRC) <<  "reseting" ;
  // 1) Turn OFF pci_config_dma_master bit

  // 2) Turn ON board_reset
  layout->control_reg = BOARD_RESET;

  // 3) Turn OFF board_reset << (void*)  ON LANai reset and ON EBus reset
  //layout->control_reg = ((1<<27) | (1<<30) | (1<<28))
//
//
//MUDEI AQUI ADICIONEI  LANAI_RESET
//
//
  layout->control_reg = (BOARD_RUN | LANAI_RESET | EBUS_RESET );
  //suspend();

  // 4) Write a temporary clockval to the LANai clockcharval register
  //     (lanai_special->clockval = 0x80)
  layout->regs.clock = htonl(0x80);

  // 5) Delay 10millisec
  delay();

  // 6) Read the clockval from the EEPROM
  clockval = htonl(layout->eeprom.lanai_clockval);
```

```
db<Myrinet9>(TRC) << "clockval " << (void*)clockval <<"\n";

  // 7) Write a 1x clock_value into the LANai
  clockval &= (unsigned int) 0xFFFFFF8F;
  layout->regs.clock = htonl(clockval);

  // 8) Delay 10millisec
  delay();

  // 9) Turn OFF EBus reset -- layout->control_reg = (1<<29)
  layout->control_reg = EBUS_RUN;

  // 10) Delay 10millisec
  delay();

  // 11) Measure the speed of the LANai processor (For LANai9 << (void*)  use CPUC and R
  bus_rate = 133;

  // 12) Calculate a new clock_value multiplier using the LANai rate
  //      and the max_lanai_rate from the EEPROM
  clockval |= 0x80;

  // 13) Turn ON EBus reset -- layout->control_reg = (1<<28)
  layout->control_reg = EBUS_RESET;

  // 14) Write the new clock_value into the LANai
  //      lanai_special->clockval = calculated_clockval
  layout->regs.clock = htonl(clockval);
db<Myrinet9>(TRC) << "new clock is " << htonl(clockval) << "\n";

  // 15) Delay 10millisec
  delay();

  // 16) Turn OFF EBus reset -- layout->control_reg = (1<<29)
  layout->control_reg = EBUS_RUN;
```

```
  // memsetting SRAM to 0
  CPU::memset((void *) layout->sram ,  0 ,  MYRINET_SRAM_SIZE);
db<Myrinet9>(TRC) << "memory zeroed\n";

  // 17) Load code into the LANai SRAM
  // load_mcp();

  // 18) Turn ON pci_config_dma_master_bit

  // 19) Turn OFF lanai_reset -- layout->control_reg = (1<<31)
  resume();

db<Myrinet9>(TRC) << "reset process completed \n";
}

void Myrinet9::resume(){
  layout->control_reg = LANAI_RUN;
  layout->regs.myrinet = MYRINET_DEFAULT_OPTIONS;
  delay(); //1 sec
  delay(); //1 sec
for(long long i =0;i<0x0fffffff;i++);
//db<Myrinet9>(TRC) << "running, control register:" << (void*)layout->control_reg <<"\n"
}

void Myrinet9::suspend(){
  layout->control_reg = LANAI_RESET;
db<Myrinet9>(TRC) << "halted, control register:" << (void*)layout->control_reg <<"\n";
}

void Myrinet9::int_enable(){
  layout->control_reg = INTERRUPT_ENABLE;
db<Myrinet9>(TRC)  << "control register is " << (void*) layout->control_reg <<"\n";
}

void Myrinet9::int_disable(){
  layout->control_reg = INTERRUPT_DISABLE;
```

```
db<Myrinet9>(TRC) << "int disabled, control register:" << (void*)layout->control_reg <<"
}


void Myrinet9::load_mcp(){
 CPU::memcpy( (void*)layout->sram, (void*)lanai_executable, lanai_executable_length);
db<Myrinet9>(TRC) << "MCP Loaded\n";
}


void Myrinet9::reset_channel(){
  layout->regs.rml = 0;
  layout->regs.rml = 0;
layout->regs.smlt = 0;
db<Myrinet9>(TRC) << "fetched the reset flit\n";
}

void Myrinet9::delay(){
  for(long long i = 0 ; i <  0x00ffffff  ; i++ );
}


void Myrinet9::set_shmem(){
  shmem = (MCP_Shmem * ) &layout->sram[MCP_SHMEM_ADDR];
  CPU::memset((void*)shmem, 0, sizeof(MCP_Shmem));
shmem->status = htonl(HOST_READY);//NIC_RESET);
  shmem->n_nodes = htonl(n_nodes);
  shmem->node_id = htonl(node_id);
  shmem->mtu = htonl(Myrinet_MTU);
  shmem->phy_addr = htonl( reinterpret_cast<unsigned int>(dma_log_addr) );
db<Myrinet9>(TRC) << "shmem ok \n";
}


void Myrinet9::handshake(){
//shmem->status = htonl(HOST_READY);
```

```
db<Myrinet9>(TRC) << "HOST is ready awaiting for NIC\n"   ;
db<Myrinet9>(TRC) << "status: " <<  htonl(shmem->status) << "\n" ;
while(shmem->status != htonl(NIC_READY));
db<Myrinet9>(TRC) << "nic is ready" <<  htonl(shmem->status) << "\n" ;
}


void Myrinet9::init_dma_chan(){
  //lanai memory that will hold the dma control block
  layout->bank_pointers.DMA_channel_0 =  DMA_DESC_OFFSET ;
  layout->bank_pointers.DMA_channel_1 =  DMA_DESC_OFFSET ;
  layout->bank_pointers.DMA_channel_2 =  DMA_DESC_OFFSET ;
  layout->bank_pointers.DMA_channel_3 =  DMA_DESC_OFFSET ;

db<Myrinet9>(TRC)  << "dma channels initialized \n";
}


void Myrinet9::set_node_id(){
  node_id = 0;
  Myrinet_Address addr = ntohs(*((unsigned short *)&layout->eeprom.lanai_board_id[0]));
  addr <<= 32;
  addr |= ntohl(*((unsigned long *)&layout->eeprom.lanai_board_id[2]));

  Reg32 i;
  for(i = 0; (i < n_nodes) && (address_table[i] != addr); i++);
  if(i == n_nodes) {
db<Myrinet9>(TRC) << "This NIC is not in address table! \n";
    return;
  }
node_id = i;
db<Myrinet9>(TRC) << "id asigned, id is:" << node_id << "\n"  ;
}


void Myrinet9::get_node_id(unsigned short  * const id){
```

```
*id = node_id;
}

inline void Myrinet9::best_size(Reg32 * const size , Reg32 len) {
if(len <= 512)
    *size = 256;
  else if(len <= 1024)
    *size = 512;
  else if(len <= 4096)
    *size = 1024;
  else if(len <= 8192)
    *size = 2048;
  else
    *size = 4096;

}

inline void Myrinet9::
build_datagram(Node dst, Reg32 len, Reg32 burst){
dtg->route[7]       = routing_table[node_id][dst] ;
  dtg->header.source  = htons(node_id) ;
dtg->header.type    = htons(SNOW_PACKET_TYPE);
  dtg->header.length  = htonl(len) ;
  dtg->burst          = htonl(burst);
//for NIC
}

inline void Myrinet9::trigger_send(Reg32 burst){
layout->regs.smp    = htonl ( SEND_DMA_OFFSET );
  layout->regs.sa     = htonl( header_align - 1 );
  layout->regs.smh    = htonl ( SEND_DMA_OFFSET + header_align);
  layout->regs.smlt   = htonl ( SEND_DMA_OFFSET + sizeof(MCP_Datagram) + burst  ) ;
  while ( !  ( htonl ( layout->regs.isr ) & SEND_INT_BIT )  ) ;
}

inline void Myrinet9::ready_to_send(){
```

```
  shmem->send_status = htonl(PACKET_READY_TO_SEND);
//printdd();
// kout << "aqui\n";
  while( htonl(shmem->send_status) != SEND_OK);
}




inline void Myrinet9::net_dma(Node dst, void * msg, Reg32 len){
build_datagram(dst,len,len);
CPU::memcpy(&(dtg->body),msg,len);
trigger_send(len);
}

inline void Myrinet9::host_dma (Node dst, void * msg, Reg32 len){
  Reg32 burst;
  Reg32 total;

shmem->smsg = htonl( (unsigned int)msg);

//best_size(&burst,len);

build_datagram(dst,len,len);
//build_datagram(dst,len,burst);

  //shmem->burst =   htonl(burst) ;
shmem->burst =   htonl(len) ;

  ready_to_send();
// kout << "done\n";
//  total = burst;

//  while(total < len){
//  msg = ( char*) msg + burst;
//  shmem->smsg = htonl( (unsigned int) msg);
```

```
//  ready_to_send();
//  total+= burst;
//  }

// kout << "sent\n";
}

void Myrinet9::send(Node dst, void * msg, Reg32 len){

if(len <= 256) //pipeline first stage
net_dma(dst,msg,len);
else
host_dma(dst,msg,len);

//printdtg();
}


inline void Myrinet9::ready_to_receive(){
  shmem->recv_status = htonl(HOST_READY_TO_RECEIVE);
//printshm();
//printdd();
  while(shmem->recv_status != htonl(PACKET_READY_TO_RECEIVE));
}


void Myrinet9::receive(Node * const src, void * msg , Reg32 * const len){
  Reg32 received;
MCP_Packet *mpkt  = (MCP_Packet*) &layout->sram[RECV_DMA_OFFSET];


shmem->rmsg = htonl((unsigned int)msg);
  //kout << "receive \n";
  ready_to_receive( );
```

```
received = htonl( shmem->burst );

*len = htonl(mpkt->header.length);
*src = htons(mpkt->header.source);


//  while(received < *len ){
//  kout << "pipe?\n";
//  shmem->rmsg = htonl((unsigned int)msg + received);
//  ready_to_receive ( );
//  received += received;//htonl( shmem->burst );

//  }


//kout << "exiting receive\n";
}
```

```
// void Myrinet9::receive(Node * const src, void * msg , Reg32 * const len){
//   Reg32 received =0;
//  Reg32 burst=0;
//  MCP_Packet *mpkt = (MCP_Packet*) &layout->sram[RECV_DMA_OFFSET ] ;

//  layout->regs.rmp = htonl( RECV_DMA_OFFSET );
//   layout->regs.rml = htonl( RECV_DMA_OFFSET + Myrinet_MTU ) ;
//  while ( ! (    htonl (layout->regs.isr ) &  RECV_INT_BIT )   );

//  received  = htonl(mpkt->burst);

//  CPU::memcpy ( msg , &(mpkt->body) , received );
```

```
// /*
//  while(received < htonl (mpkt->header.length) ){

//  layout->regs.rmp = htonl( RECV_DMA_OFFSET );
//  layout->regs.rml = htonl( RECV_DMA_OFFSET + Myrinet_MTU) ;
//  while ( ! (    htonl (layout->regs.isr ) &  RECV_INT_BIT )   );

//  CPU::memcpy ( (char*) msg + received , &(mpkt->body) , received );
//  received += burst ;

//  }*/
//  *src     = htons(mpkt->header.source);
//  *len     = htonl(mpkt->header.length);
//  // kout << "received " << *len << "\n";
// }


 void Myrinet9::printdd(){
kout << "====================descriptor\n";
kout << "next " << (void*) htonl(desc->next) << " starts at offset " << (void*)DMA_DESC_
kout << "len " << htonl(desc->len) <<"\n";
kout << "lar " << (void*)htonl(desc->lar) << " starts at offset " << (void*)SEND_DMA_OFF
kout << "eah " << htonl(desc->eah) <<"\n";
kout << "eal " << (void*) htonl(desc->eal) <<"\n";

}

inline void Myrinet9::printdtg(){
kout << "====================datagram\n";
kout << "dst " << (void*) dtg->route[7]  << "\n";
kout << "src " <<  htons(dtg->header.source)  << "\n";
kout << "type "<< (void*) htonl( dtg->header.type ) << "\n";
kout << "len  "<<  htonl( dtg->header.length ) << "\n";
kout << "burst " << htonl( dtg->burst ) << "\n";
// for(long long i = 0; i< 0x0fffffff;i++);
```

```
}

void Myrinet9::printshm(){
kout << "====================shmem\n";
kout << "status " << htonl(shmem->status) << "\n";
kout << "n_nodes " << htonl(shmem->n_nodes) << "\n";
kout << "node_id " << htonl(shmem->node_id) << "\n";
kout << "mtu " << htonl(shmem->mtu) << "\n";
kout << "phy_addr " << htonl(shmem->phy_addr) << "\n";
  kout << "send_status " << htonl(shmem->send_status) << "\n";
  kout << "recv_status " << htonl(shmem->recv_status) << "\n";
  kout << "burst " << htonl(shmem->burst) << "\n";
}


// void Myrinet9::print(){
//  db<Myrinet9>(TRC) <<"\n\n--- EEPROM INFO ---\n";
//  db<Myrinet9>(TRC) <<"LANai fpga version:" << reinterpret_cast<void*>(const_cast<unsi
//  db<Myrinet9>(TRC) <<"LANai more version:" << (void*)layout->eeprom.more_version<<"\n
//  db<Myrinet9>(TRC) <<"LANai delay line:" <<   reinterpret_cast<void*>( ntohs(layout->
//  db<Myrinet9>(TRC) <<"LANai board type:" << reinterpret_cast<void*>( ntohs(layout->ee
//  db<Myrinet9>(TRC) <<"LANai bus type:" << reinterpret_cast<void*>( ntohs(layout->eepr
//  db<Myrinet9>(TRC) <<"LANai product code:" << reinterpret_cast<void*>( ntohs(layout->
//  db<Myrinet9>(TRC) <<"LANai board label:" <<  (void*)layout->eeprom.board_label<<"\n"
//  db<Myrinet9>(TRC) <<"LANai sram size: " <<  (void*)ntohl(layout->eeprom.lanai_sram_s
//  db<Myrinet9>(TRC) <<"LANai voltage code: " << reinterpret_cast<void*>( ntohs(layout-
//  db<Myrinet9>(TRC) <<"LANai pad voltage: " << reinterpret_cast<void*>( ntohs(layout->
//  db<Myrinet9>(TRC) <<"LANai serial number:" <<   ntohl(layout->eeprom.serial_number)<
//  db<Myrinet9>(TRC) <<"LANai max speed:" <<   ntohs(layout->eeprom.max_lanai_speed)<<"
//  db<Myrinet9>(TRC) <<"LANai clockval:" <<  (void*)ntohl(layout->eeprom.lanai_clockval
//  db<Myrinet9>(TRC) <<"LANai board id:" <<  layout->eeprom.lanai_board_id << "\n";
//  db<Myrinet9>(TRC) <<"LANai cpu version:" << reinterpret_cast<void*>( ntohs(layout->e
//  db<Myrinet9>(TRC) <<"-------------------\n";
}

//void Myrinet9::printextra(){
```

```
//  db<Myrinet9>(TRC) << "control register is " << (void*) layout->control_reg << "\n";

//  db<Myrinet9>(TRC) << "node id is " << node_id <<"\n" ;
// }

// Methods

__END_SYS
```

7.1.3 myrinet9_init.cc

```
// EPOS Myrinet9 Initialization
//
// Author: secco
// Documentation: $EPOS/doc/nic Date: 17 Feb 2004

#include <mediator/nic/myrinet9/myrinet9.h>
#include <mediator/nic/common.h>
#include <mediator/pci/pc_pci.h>
#include <mediator/pci/common.h>
#include <mediator/cpu/ia32.h>
#include <mmu.h>
#include <utility/malloc.h>

__BEGIN_SYS

// Class initialization
int Myrinet9::init(System_Info * si)
{
  db<Myrinet9>(TRC) << "Myrinet9::init()\n";
  PC_PCI pci;
  PC_PCI::Locator loc = pci.scan(VendorID, DeviceID, 0);

  if (! loc ){
    db<Myrinet9>(TRC)  << "Holy Shit! Device not found!\n";
    return -1;
  }

  for(Reg32 i=0; i< si->iomm_size; i++) {
    if (si->iomm[i].locator == (loc.bus << 8 | loc.dev_fn)) {
      log_addr = si->iomm[i].log_addr;
      phy_addr = si->iomm[i].phy_addr;
    }
  }
```

```
  db<Myrinet9>(TRC) << "Log: " << (void*) log_addr << " "
<< "Phy: " << (void*) phy_addr << " \n";

  layout = (L9*)log_addr;

//dma_log_addr =  (char *)(sysalloc(64*1024));
//dma_phy_addr =  dma_log_addr;


//  dma_log_addr = new char[64*1024];
// dma_phy_addr = reinterpret_cast<char*> ((unsigned int) MMU::physical( MMU::Log_Addr(d

db<Myrinet9>(TRC) << "Dma Log: " << (void*) dma_log_addr  << " " << "DMA Phy: " << (void

  db<Myrinet9>(TRC) << "Myrinet9 initialization process\n";

  reset();
init_dma_chan();
reset_channel();
set_node_id();

bool usemcp = true ;

if(usemcp){
reset();
suspend();
    set_shmem();
  int_disable();
  load_mcp();
    int_enable();
    resume();
reset_channel();
handshake();
}
```

```
  kout << "Initialization Completed\n";

  return 0;
}

__END_SYS
```

7.1.4 myrinet9_test.cc

```
// EPOS Myrinet9 Test Program
//
// Author: secco
// Documentation: $EPOS/doc/nic Date: 17 Feb 2004

#include <utility/ostream.h>
#include <nic.h>
#include <framework.h>
#include "getus.h"
#include "mcp/mcp_executable.h"

__USING_SYS

int main()
{
  OStream cout;

  cout << "Myrinet9 test\n";
  Myrinet9 nic;
  unsigned short id;
  nic.get_node_id ( &id );

unsigned long long START, STOP;



unsigned short from;
unsigned int buff_size = 4096;

char *sbuf = (char *)malloc(buff_size);
char *rbuf = (char *)malloc(buff_size);

unsigned int  len;
  CPU::memset(sbuf,'R',buff_size);
```

```
  cout << "my id is " << id << "\n";
int recv=0,send=0;
if(id == 1){
for(;;){
nic.receive( &from ,rbuf, &len);
nic.send(from,sbuf,len );
}
} else{
from = 1;
int j = 1  ;
for(; j <= buff_size ; j *=2){
if(j != 128){
GETUS(&START);
for(int i=0; i< 10;i++){
nic.send( from ,sbuf , j );
nic.receive ( &from, rbuf, &len);
}
GETUS(&STOP);
cout << "I/O Size x Time: "<< j << " x "  << (unsigned long) (  ( (STOP - START)  / (  1
}else
kout << "will jump the fucky 128!!\n";
}

}

kout << "done \n";
for(;;);



return 0;

}
```

7.1.5 lanai9_def.h

```
#ifndef __lanai9_def_h
#define __lanai9_deh_h
#define __myrinet_nic_h =1;


/*
 *  Configuration Space
 */

#define VendorID    0x14C1
#define DeviceID    0x8043
#define RevisionID  0x3


/*
 * Myrinet NIC Bank of Pointers.
 */

typedef struct {
 volatile unsigned int DMA_channel_0; // 0x00 used by nic
 unsigned int DMA_channel_1; // 0x04
 volatile unsigned int DMA_channel_2; // 0x08 used by nic
 unsigned int DMA_channel_3; // 0x0C
 unsigned int FIFO_0;        // 0x10
 unsigned int FIFO_1;        // 0x14
 unsigned int FIFO_2;        // 0x18
 unsigned int FIFO_3;        // 0x1C
 unsigned int unused[8];
} Myrinet_Bank_Pointers;



/*
 * Control register (Offset 0x00800040)
```

```
 */

#define LANAI_RUN            (1<<31)
#define LANAI_RESET          (1<<30)
#define EBUS_RUN             (1<<29)
#define EBUS_RESET           (1<<28)
#define BOARD_RUN            (1<<27)
#define BOARD_RESET          (1<<26)
#define INTERRUPT_ENABLE     (1<<25)
#define INTERRUPT_DISABLE    (1<<24)

#define DMA_TRIGGER_3        (1<<23)
#define DMA_TRIGGER_2        (1<<22)
#define DMA_TRIGGER_1        (1<<21)
#define DMA_TRIGGER_0        (1<<20)
#define FORCE_32BITS         (1<<6)

/*
 * ===============  DMA Controller definitions and structures  ===============
 */


/*
 * DMA control block
 * since myrinet can change this filds they need
 * to be volatile
 */

typedef struct
{
  volatile unsigned int next;  /* next pointer */
  volatile unsigned short csum[2]; /* 2 16-bit ones complement checksums of this block *
  volatile unsigned int len;   /* byte count */
  volatile unsigned int lar;   /* lanai address register */
  volatile unsigned int eah;   /* high 32bit external (PCI) address */
  volatile unsigned int eal;   /* low 32bit external (PCI) address */
```

```
} DMA_Descriptor;


/*
 * Bits in the NEXT field of a DMA block
 */

#define DMA_L2H       0x0 /* LBUS to EBUS (lanai to host) */
#define DMA_H2L       0x1 /* EBUS to LBUS (host to lanai) */
#define DMA_TERMINAL  0x2 /* terminal block? */
#define DMA_WAKE      0x4 /* wake when block completes */


/*
 * ===============  LANai 9.1 definitions and structures  ===============
 */


/*
 * Bits in the IMR and ISR
 */

#define DEBUG_BIT        (1<<31)
#define HOST_SIG_BIT     (1<<30)
#define LANAI_SIG_BIT    (1<<29)

#define LINK_DOWN_BIT    (1<<16)
#define PAR_INT_BIT      (1<<15)
#define MEM_INT_BIT      (1<<14)
#define TIME2_INT_BIT    (1<<13)
#define WAKE0_INT_BIT    (1<<12)
#define NRES_INT_BIT     (1<<11)
#define ORUN4_INT_BIT    (1<<10)
#define ORUN2_INT_BIT    (1<<9)
#define ORUN1_INT_BIT    (1<<8)
#define TIME1_INT_BIT   (1<<7)
```

```
#define TIME0_INT_BIT    (1<<6)
#define WAKE2_INT_BIT    (1<<5)
#define WAKE1_INT_BIT    (1<<4)
#define SEND_INT_BIT     (1<<3)
#define BUFF_INT_BIT     (1<<2)
#define RECV_INT_BIT     (1<<1)
#define HEAD_INT_BIT     (1<<0)


/*
 * Bits in the MYRINET register
 */

#define  NRES_ENABLE_BIT (1<<0)
#define  CRC32_ENABLE_BIT (1<<1)
#define  TX_CRC8_ENABLE_BIT (1<<2)
#define  RX_CRC8_ENABLE_BIT (1<<3)
#define  ILLEGAL_ENABLE  (1<<4)
#define  BEAT_ENABLE  (1<<5)
#define  TIMEOUT0  (1<<6)
#define  TIMEOUT1  (1<<7)
#define  WINDOW0   (1<<8)
#define  WINDOW1   (1<<9)
#define MYRINET_DEFAULT_OPTIONS  (TIMEOUT0 | TIMEOUT1 | WINDOW0 | WINDOW1 | RX_CRC8_ENAB


/*
 * Myrinet NIC special registers.
 */

typedef struct
{
    volatile unsigned int  pad0x0;         /* 0x00 */
    volatile unsigned int  pad0x4;
    volatile unsigned int  pad0x8;         /* 0x08 */
    volatile unsigned int  pad0xc;
```

```
volatile unsigned int  pad0x10;        /* 0x10 */
volatile unsigned int  pad0x14;
volatile unsigned int  pad0x18;        /* 0x18 */
volatile unsigned int  pad0x1c;
volatile unsigned int  pad0x20;        /* 0x20 */
volatile unsigned int  pad0x24;
volatile unsigned int  pad0x28;        /* 0x28 */
volatile unsigned int  pad0x2c;
volatile unsigned int  pad0x30;        /* 0x30 */
volatile unsigned int  pad0x34;
volatile unsigned int  pad0x38;        /* 0x38 */
volatile unsigned int  pad0x3c;
volatile unsigned int  pad0x40;        /* 0x40 */
volatile unsigned int  pad0x44;
volatile unsigned int  pad0x48;        /* 0x48 */
volatile unsigned int  pad0x4c;
volatile unsigned int  isr;            /* 0x50 */
volatile unsigned int  pad0x54;
volatile unsigned int  eimr;           /* 0x58 */
volatile unsigned int  pad0x5c;
volatile unsigned int  it0;            /* 0x60 */
volatile unsigned int  pad0x64;
volatile int           rtc;            /* 0x68 */
volatile unsigned int  pad0x6c;
volatile unsigned int  lar;            /* 0x70 */
volatile unsigned int  pad0x74;
volatile unsigned int  cpuc;           /* 0x78 */
volatile unsigned int  pad0x7c;
volatile unsigned int  pad0x80;        /* 0x80 */
volatile unsigned int  pad0x84;
volatile unsigned int  pad0x88;        /* 0x88 */
volatile unsigned int  pad0x8c;
volatile unsigned int  pad0x90;        /* 0x90 */
volatile unsigned int  pad0x94;
volatile unsigned int  pad0x98;        /* 0x98 */
volatile unsigned int  pad0x9c;
```

```
volatile unsigned int  pad0xa0;        /* 0xA0 */
volatile unsigned int  pad0xa4;
volatile unsigned int  pad0xa8;        /* 0xA8 */
volatile unsigned int  pad0xac;
volatile unsigned int  pad0xb0;        /* 0xB0 */
volatile unsigned int  pad0xb4;
volatile unsigned int  pulse;          /* 0xB8 */
volatile unsigned int  pad0xbc;
volatile unsigned int  it1;            /* 0xC0 */
volatile unsigned int  pad0xc4;
volatile unsigned int  it2;            /* 0xC8 */
volatile unsigned int  pad0xcc;
volatile unsigned int  pad0xd0;        /* 0xD0 */
volatile unsigned int  pad0xd4;
volatile unsigned int  pad0xd8;        /* 0xD8 */
volatile unsigned int  pad0xdc;
volatile unsigned int  rmp;            /* 0xE0 */
volatile unsigned int  pad0xe4;
volatile unsigned int  rml;            /* 0xE8 */
volatile unsigned int  pad0xec;
volatile unsigned int  smp;            /* 0xF0 */
volatile unsigned int  pad0xf4;
volatile unsigned int  smh;            /* 0xF8 */
volatile unsigned int  pad0xfc;
volatile unsigned int  sml;            /* 0x100 */
volatile unsigned int  pad0x104;
volatile unsigned int  smlt;           /* 0x108 */
volatile unsigned int  pad0x10c;
volatile unsigned int  smc;            /* 0x110 */
volatile unsigned int  pad0x114;
volatile unsigned int  sa;             /* 0x118 */
volatile unsigned int  pad0x11c;
volatile unsigned int  pad0x120;       /* 0x120 */
volatile unsigned int  pad0x124;
volatile unsigned int  pad0x128;       /* 0x128 */
volatile unsigned int  pad0x12c;
```

```
volatile unsigned int  myrinet;        /* 0x130 */
volatile unsigned int  pad0x134;
volatile unsigned int  debug;          /* 0x138 */
volatile unsigned int  pad0x13c;
volatile unsigned int  led;            /* 0x140 */
volatile unsigned int  pad0x144;
volatile unsigned int  pad0x148;       /* 0x148 */
volatile unsigned int  pad0x14c;
volatile unsigned int  mp;             /* 0x150 */
volatile unsigned int  pad0x154;
volatile unsigned int  pad0x158;
volatile unsigned int  pad0x15c;
volatile unsigned int  pad0x160;
volatile unsigned int  pad0x164;
volatile unsigned int  pad0x168;
volatile unsigned int  pad0x16c;
volatile unsigned int  pad0x170;
volatile unsigned int  pad0x174;
volatile unsigned int  pad0x178;
volatile unsigned int  pad0x17c;
volatile unsigned int  pad0x180;
volatile unsigned int  pad0x184;
volatile unsigned int  pad0x188;
volatile unsigned int  pad0x18c;
volatile unsigned int  pad0x190;
volatile unsigned int  pad0x194;
volatile unsigned int  pad0x198;
volatile unsigned int  pad0x19c;
volatile unsigned int  pad0x1a0;
volatile unsigned int  pad0x1a4;
volatile unsigned int  pad0x1a8;
volatile unsigned int  pad0x1ac;
volatile unsigned int  pad0x1b0;
volatile unsigned int  pad0x1b4;
volatile unsigned int  pad0x1b8;
volatile unsigned int  pad0x1bc;
```

```
    volatile unsigned int  pad0x1c0;
    volatile unsigned int  pad0x1c4;
    volatile unsigned int  pad0x1c8;
    volatile unsigned int  pad0x1cc;
    volatile unsigned int  pad0x1d0;
    volatile unsigned int  pad0x1d4;
    volatile unsigned int  pad0x1d8;
    volatile unsigned int  pad0x1dc;
    volatile unsigned int  pad0x1e0;
    volatile unsigned int  pad0x1e4;
    volatile unsigned int  pad0x1e8;
    volatile unsigned int  pad0x1ec;
    volatile unsigned int  pad0x1f0;
    volatile unsigned int  pad0x1f4;
    volatile unsigned int  clock;           /* 0x1F8 */
} Myrinet_Special_Regs;


/*
 * Myrinet NIC EEPROM.
 */

typedef struct
{
    /*  byte offset - meaning */

    /*  0 - 32-bit value to be used to initialize the LANai chip */
    unsigned int lanai_clockval;
    /*  4 - 0x0701 would be LANai 7.1 */
    unsigned short lanai_cpu_version;
    /*  6 - 48-bit MAC Address */
    unsigned char lanai_board_id[6];
    /* 12 - LANai SRAM size in bytes */
    unsigned int lanai_sram_size;
    /* 16 - Used to encode information about FPGA, if any */
    unsigned char fpga_version[32];
```

```
    /* 48 - Used to encode information about FPGA, if any */
    unsigned char more_version[16];
    /* 64 - Obsolete - unused */
    unsigned short delay_line_value;
    /* 66 */
    unsigned short board_type;
    /* 68 */
    unsigned short bus_type;
    /* 70 */
    unsigned short product_code;
    /* 72 - Myricom serial number */
    unsigned int serial_number;
    /* 76 - Expected board label */
    unsigned char  board_label[32];
    /*108 - Maximum LANai processor speed (MHz) */
    unsigned short max_lanai_speed;
    /* 110 */
    unsigned char voltage_code;
    /* 111 */
    unsigned char pad_voltage_code;
    /* 112 */
    unsigned short future_use[6];
    /* 124 */
    unsigned int unused_4_bytes;
} Myrinet_EEPROM ;


/*
Memory Layout - Byte offset from beginning of the 16MB space:

     =
8Mb  |   0x00000000 - LANai SRAM space: 8 Megabytes maximum allowed
     =
     |    0x00800000 - Configuration register: Remap of config space (64 bytes)
     |    0x00800040 - Control register: (4 bytes)
     |    0x00800100 - Bank of pointers for DMA and Doorbell: (64 bytes)
```

```
8Mb  |    0x00804000 - LANai special registers: (16K bytes)
     |    0x00808000 - Doorbell Region: (8 Megabytes - 32K bytes, write-only)
     |    0x00A00000 - EEPROM board information (4 Megabytes, read-only)
     |    0x01000000 - End of mapped space
     =

The overlap between the EEPROM board information and the doorbell region
is intentional, and is possible because the EEPROM is read-only, while the
Doorbell Region is write-only.

This is the memory of the LANai processor. It should be accessed as
big-endian  memory since the LANai is a big-endian device.
*/

#define SRAM_OFFSET          0x00000000UL  /* 0 */
#define CONFIG_REG_OFFSET    0x00800000UL  /* 8Mb */
#define CONTROL_REG_OFFSET  0x00800040UL  /* 8Mb + 64 bytes */
#define BANK_POINTERS_OFFSET 0x00800100UL  /* 8Mb + 256 bytes */
#define SPECIAL_REGS_OFFSET  0x00804000UL  /* 8Mb + 16 Kb */
#define DOORBELL_OFFSET      0x00808000UL  /* 8Mb + 32 Kb */
#define EEPROM_OFFSET        0x00A00000UL  /* 10Mb */
#define MYRINET_SIZE                0x01000000UL  /* 16Mb */

#define MYRINET_SRAM_SIZE           0x00800000UL  /* 8 Mb */
#define MYRINET_CONFIG_REGS_SIZE    0x00000040UL  /* 64 bytes */
#define MYRINET_CONTROL_REG_SIZE    sizeof(volatile unsigned int)  /* 4 bytes */
#define MYRINET_PAD0_SIZE     BANK_POINTERS_OFFSET  - (CONTROL_REG_OFFSET + MYRINET_CONTI
#define MYRINET_BANK_POINTERS_SIZE   0x00000040UL  /* 64 bytes */
#define MYRINET_PAD1_SIZE           SPECIAL_REGS_OFFSET - (BANK_POINTERS_OFFSET + sizeof
#define MYRINET_SPECIAL_REGS_SIZE   0x00004000UL  /* 16Kb */
#define MYRINET_DOORB_SIZE          0x007F8000UL  /* 8Mb - 32Kb */
#define MYRINET_PAD2_SIZE           EEPROM_OFFSET - (SPECIAL_REGS_OFFSET + sizeof(Myrinet
#define MYRINET_EEPROM_SIZE         0x00400000UL  /* 4Mb */
#define MYRINET_PAD3_SIZE           MYRINET_SIZE - (EEPROM_OFFSET + sizeof(Myrinet_EEPROI
```

```
typedef unsigned long long DWORD;
typedef unsigned long WORD;

typedef unsigned long long Myrinet_Address;
typedef unsigned short Node;
typedef unsigned int Reg32;

typedef volatile unsigned char Myrinet_SRAM[MYRINET_SRAM_SIZE];
typedef volatile unsigned int Myrinet_Config_Regs[MYRINET_CONFIG_REGS_SIZE / sizeof(vola
typedef volatile unsigned int Myrinet_Control_Register;

//max size of incoming / outgoing message  0x0001000  4KB  or 0x00400000 4MB?
#define Myrinet_MTU  768000



/*
 * Myrinet NIC memory layout.
 */
typedef struct {
    Myrinet_SRAM sram;
    Myrinet_Config_Regs config_regs;
    Myrinet_Control_Register control_reg;
    unsigned char pad0[MYRINET_PAD0_SIZE];
    Myrinet_Bank_Pointers bank_pointers;
    unsigned char pad1[MYRINET_PAD1_SIZE];
    Myrinet_Special_Regs regs;
    unsigned char pad2[MYRINET_PAD2_SIZE];
    const Myrinet_EEPROM eeprom;
    unsigned char pad3[MYRINET_PAD3_SIZE];
}L9;

#endif
```

7.1.6 mcp.h

```
//========================================================================
// MYRINET CONTROL PROGRAM
//
// Desc:
//
// Date: 13 Dec 1999 Auth: Guto
// Changes 17 Jan 2004 Auth: Secco
//========================================================================
#ifndef __mcp_h
#define __mcp_h 1

//========================================================================
// DEPENDENCIES
//========================================================================
#include "lanai9_def.h"

//========================================================================
// CONSTANTS
//========================================================================
// Maximum packet length in bytes (for the transfer pipeline)
#define MCP_PACKET_LEN        0x00010000UL  // 64 Kb

// MCP shared memory address (relative to LANai)
// This should be MCP code + stack
//#define MCP_SHMEM_ADDR 0x00008000UL  // 32 Kb

// MCP communication buffer address (relative to LANai)
//#define MCP_BUFFER_ADDR 0x00010000UL  // 64 Kb

#define SNOW_PACKET_TYPE      0x0666U
//========================================================================
// TYPES
//========================================================================
// Myrinet route between two nodes
```

```
//typedef char Myrinet_Route[CONF_MYRINET_ROUTE_LENGTH];


// Status to synchronize host and NIC
enum MCP_Status
  {
    NIC_RESET = 0,
    NIC_READY = 1,
    HOST_READY = 2,
    NIC_CLEAR_TO_SEND = 10,
    HOST_READY_TO_SEND = 11,
    HEADER_READY_TO_SEND = 12,
    PACKET_READY_TO_SEND = 13,
    SEND_OK = 14,
    SEND_ERROR = 15,
    SEND_REGS_OK = 16,
    DMA_OK = 17,
    STARTSEND =18,
    NIC_CLEAR_TO_RECEIVE = 20,
    HOST_READY_TO_RECEIVE = 21,
    HEADER_READY_TO_RECEIVE = 22,
    PACKET_READY_TO_RECEIVE = 23,
    RECEIVE_OK = 24,
    RECEIVE_ERROR = 25,
    DMA_ENTER = 26,
    DMA_EXIT  = 27
  };

/*we dont need this yet*/
// Myrinet control program executable COFF image
/*struct MCP_Image
{
  static int code_start;
  static int code_size;
  static unsigned short code_seg[];
  static int data_start;
```

```
  static int data_size;
  static unsigned short data_seg[];
  static int bss_start;
  static int bss_size;
  static int shared_start;
  static int shared_size;
};
*/
// Myrinet control program statistics


const unsigned int header_align = 8;

struct _MCP_Statictics {
  int packets_sent;
  int packets_received;
  int packets_droped;
  int routing_errors;
  int header_errors;
  int length_errors;
  int trailer_errors;
  int crc_errors;
  int resets;
};
typedef struct _MCP_Statistics MCP_Statistics;
// Myrinet control program packet



// Myrinet control program shared memory (between host and NIC)
struct _MCP_Shmem {
  volatile unsigned int status;
  volatile unsigned int n_nodes;
  volatile unsigned int route_len;
  volatile unsigned int node_id;
  volatile unsigned int mtu;
```

```
  volatile unsigned int phy_addr;
  volatile unsigned int smsg;
volatile unsigned int rmsg;
  volatile unsigned int send_status ;
  volatile unsigned int recv_status ;
  //MCP_Datagram smsg;
volatile unsigned int burst;
  //MCP_Packet rmsg;
//volatile unsigned int rmsg;
  //volatile MCP_Statistics statistics;
};
typedef struct _MCP_Shmem MCP_Shmem;

/*=========================================================================*/
/* ClASS MCP */
/*=========================================================================*/
//class MCP
//{
// public:
//  MCP();
//void send();
//void receive();
//void reset();

// private:
//void drop();

// private:
//MCP_Shmem *shmem;
//char *sbuf[2];
//char *rbuf[2];
//char *sbuf_ebus_addr[2];
//char *rbuf_ebus_addr[2];
//};

#endif /* __mcp_h */
```

7.1.7 mcp_shmem.h

```
#ifndef __MEM_LAYOUT_H__
#define __MEM_LAYOUT_H__

const unsigned int MAXSIZE           = 0x00200000UL;

/* IF YOU CHANGE HERE UPDATE THE MCP.C!!!!!!!!!!!!!!!!*/

const unsigned int MCP_OFFSET        = 0;       //0
const unsigned int MCP_SHMEM_ADDR    = 0x00008000UL; //32K
const unsigned int DMA_DESC_OFFSET   = 0x00010000UL; //64K
const unsigned int SEND_DMA_OFFSET   = 0x00030000UL; //196K
const unsigned int RECV_DMA_OFFSET   = 0x00050000UL; //327K
const unsigned int FIFO_REGION_OFFSET = 0x00090000UL; //589824

//const unsigned int SAFE_REGION_OFFSET = 0x00090000;
//const unsigned int MCP_BUFFER_ADDR    = 0x00040000; //262144

#endif  /* __MEM_LAYOUT_H__ */
```

7.1.8 mcp_def.h

```
/*****************************
 ** Memory-mapped registers **
 *****************************/
#define MM_REG_FULLWORD_P(X) (*((void* volatile * const ) (0xFFFFFE00 + (X))))
#define MM_REG_FULLWORD(X)   (*((int   volatile * const ) (0xFFFFFE00 + (X))))
#define MM_REG_HALFWORD(X)   (*((short volatile * const ) (0xFFFFFE00 + (X))))
#define MM_REG_BYTE(X)       (*((char  volatile * const ) (0xFFFFFE00 + (X))))

#define CPUC MM_REG_FULLWORD   (0x078)
#define DDEBUG MM_REG_FULLWORD   (0x138)
#define EIMR MM_REG_FULLWORD   (0x058)
#define ISR MM_REG_FULLWORD   (0x050)
#define IT0 MM_REG_FULLWORD   (0x060)
#define IT1 MM_REG_FULLWORD   (0x0C0)
#define IT2 MM_REG_FULLWORD   (0x0C8)
#define LAR MM_REG_FULLWORD_P (0x070)
#define LED MM_REG_FULLWORD   (0x140)
#define MYRINET MM_REG_FULLWORD   (0x130)
#define PULSE MM_REG_FULLWORD   (0x0B8)
#define RMC MM_REG_FULLWORD_P (0x0D8)
#define RML MM_REG_FULLWORD_P (0x0E8)
#define RMP MM_REG_FULLWORD_P (0x0E0)
#define RMW MM_REG_FULLWORD_P (0x0D0)
#define RTC MM_REG_FULLWORD   (0x068)
#define SA  MM_REG_FULLWORD   (0x118)
#define SMC MM_REG_FULLWORD_P (0x110)
#define SMH MM_REG_FULLWORD_P (0x0F8)
#define SML MM_REG_FULLWORD_P (0x100)
#define SMLT MM_REG_FULLWORD_P (0x108)
#define SMP MM_REG_FULLWORD_P (0x0F0)
```

7.1.9 mcp.c

```
#include <myrinet9/mcp_def.h>
#include <myrinet9/mcp_shmem.h>
#include <myrinet9/mcp.h>

void send();
void receive();

/*IF CHANGED UPDATE SHMEM*/
MCP_Shmem * shmem = (MCP_Shmem *)0x00008000;// MCP_SHMEM_ADDR;
DMA_Descriptor * desc = (DMA_Descriptor *)0x00010000; //DMA_DESC_OFFSET

int main(){
int i;
// MCP_Shmem * shmem = (MCP_Shmem *) MCP_SHMEM_ADDR;

shmem->recv_status = NIC_CLEAR_TO_RECEIVE;
  shmem->send_status = NIC_CLEAR_TO_SEND;

if(!(ISR & NRES_INT_BIT)) {
while(shmem->status != HOST_READY);
shmem->status = NIC_READY;
}
/*  else{ */
/*  RML = (unsigned int*)RECV_DMA_OFFSET;  */
/*  RML = (unsigned int*) RECV_DMA_OFFSET;  */
/*  SMLT= (unsigned int*)RECV_DMA_OFFSET;  */
/*  }  */


LED = 1;

for(;;){
if(shmem->send_status == PACKET_READY_TO_SEND)
      send();
```

```
    if( (ISR & RECV_INT_BIT) && (shmem->recv_status == HOST_READY_TO_RECEIVE) )
receive();
}
}

void send() {
  desc->next = ( 0x00010000  | DMA_H2L);
  desc->len  = ( shmem->burst );
  desc->lar  = ( SEND_DMA_OFFSET + 24 );
  desc->eah  = 0;
  desc->eal  = (  shmem->smsg );

  PULSE  = 0x2;

while( ! ( desc->next  & DMA_TERMINAL )  );

SMP  = (unsigned int*) SEND_DMA_OFFSET;
  SA   = header_align - 1;
  SMH  = (unsigned int*)SEND_DMA_OFFSET + header_align ;
  SMLT = (unsigned int*)SEND_DMA_OFFSET +  24 + shmem->burst ;

while(!(ISR & SEND_INT_BIT)) ;

  shmem->send_status = SEND_OK;
}

void receive() {
unsigned int * burst;
  RMP = (unsigned int*) RECV_DMA_OFFSET;
  RML = (unsigned int*) RECV_DMA_OFFSET + Myrinet_MTU;


  while(!(ISR & RECV_INT_BIT));

burst = (unsigned int) (RECV_DMA_OFFSET +8);
```

```
desc->next = ( 0x00010000  | DMA_L2H);
  desc->len  = ( *burst );
  desc->lar  = ( SEND_DMA_OFFSET + 16 );
  desc->eah  = 0;
  desc->eal  = (  shmem->rmsg );

  PULSE  = 0x2;

while( ! ( desc->next  & DMA_TERMINAL )  );

shmem->burst = *burst;

shmem->recv_status = PACKET_READY_TO_RECEIVE;
}
```

7.1.10 nic.h

```
// EPOS NIC Interface
//
// Author: secco
// Documentation: $EPOS/doc/nic Date: 19 Feb 2004

#ifndef __nic_h
#define __nic_h

#include <system/config.h>

__BEGIN_SYS
__BEGIN_INT

class NIC_Common
{
protected:
    NIC_Common() {}

public:
    // NIC common types and ants
};

class NIC: public NIC_Common
{
public:
    NIC() __DEF;
    ~NIC() __DEF;

void send(unsigned short dst, void * msg, unsigned int len);
void receive(unsigned short * const src, void * msg, unsigned int * const len);
void get_node_id( unsigned short * const id);
};

class Myrinet9: public NIC_Common
```

```
{
public:
    Myrinet9() __DEF;
    ~Myrinet9() __DEF;

void send(unsigned short dst, void * msg, unsigned int len);
void receive(unsigned short *  const src, void * msg, unsigned int * const len);
void get_node_id(unsigned short * const id);

};

__END_INT
__END_SYS

#ifdef __MYRINET9_H
#include __MYRINET9_H
#endif

#endif
```

7.2 Anexo b - Artigo

Uma Família de Abstrações Myrinet para EPOS

Fernando R. Secco, Antonio A.M. Frohlich

(guto,secco)@lisha.ufsc.br

12 de julho de 2004

Resumo

Este trabalho apresenta um sistema de comunicação dedicado para redes de baixa-latência myrinet que utiliza o sistema operacional EPOS. Como metodologia foi utilizada a AOSD cujas maiores caracteristicas são a flexibilidade e simplicidade de sua componentização. A proposta do sistema de comunicação é manter o sistema o mais simples possíveis para garantir o menor atraso para mensagens curtas e a melhor utilização da largura de banda para mensagens longas.

É possível ainda, observar o impacto do software de comunicação no desempenho da aplicação. Como um sistema dedicado pode vir a contribuir com melhores soluções utilizando componentes mais simples. Os resultados alcançados com um sistema dedicado em comparação com um genérico encoraja os desenvolvedores a repensarem as metodologias empregadas no desenvolvimento de software básico.

1 Introdução

Antes de mais nada é preciso saber para que alguém utilizaria um *cluster* quais os benefícios e os problemas que este tipo de máquina computacional oferece. *Cluster* de Computadores Pessoais possuem, em primeira instância, um custo pequeno para sua concepção, tanto em tempo de planejamento (escolha do hardware ideal, software, refrigeração etc) e sua concepção é rápida pois normalmente é construído com equipamento disponível no mercado. Outro ponto importante é a facilidade de manutenção e expansão de *clusters*. A facilidade de manutenção vem da existência de vários software livres ou pagos que ajudam na automatização do processo além de ser possível desenvolver ferramentas para este fim com grande facilidade. Na sua concepção é possível utilizar diversos tipos de topologia de rede interligados por diferentes tipos de redes com a vantagem da configuração das máquinas poder ser diferente. Por esta razão é possível expandir um *cluster* para qualquer nova tecnologia emergente de forma bastante simplificada e rápida portanto, *clusters* pode sempre utilizar tecnologia de ponta. Na maior parte das vezes o custo-benefício de um *cluster* pode superar o de um supercomputador, tanto em capacidade de processamento quanto custo [Baker 2000]. Em contrapartida os problemas mais comuns estão relacionados a utilização de espaço, a qualidade do software e desempenho. Dependendo da quantidade de nodos de processamento utilizados o espaço físico utilizado pode superar o espaço reservado para sua implantação. Outro problema grave é a qualidade do software utilizado, tanto o software de gerenciamento quanto o de processamento. Esses, em sua maioria, são projetos de universidades sendo, normalmente, muito instáveis e que muitas vezes são descontinuados. Em outros casos são software livre e possuem um grau de complexidade elevado

que dificulta a sua utilização. Mas um dos maiores problemas é a baixa performance. Dependendo do tipo de hardware adquirido e do tipo de software utilizado pode-se criar vários gargalos e degradar o desempenho.

Portanto, somente a agregação de hardware não é suficiente para que um cluster apresente o resultado esperado. Ao utilizar-se uma tecnologia como essa é necessário saber exatamente o que se espera do supercomputador criado, para que se possa escolher de forma mais eficiente o software e o hardware que será utilizado nesse *cluster*. Mas essa é uma tarefa difícil, pois é necessário tempo para pesquisar tecnologias ou mesmo desenvolver novas. Por esta razão softwares e hardware padronizados são muito utilizados em aglomerados, pois a facilidade de aquisição combinado com uma vasta documentação facilitam o seu emprego e manutenção. Mas a utilização deste tipo de software e hardware apresenta custos quando a maior meta é a performance.

Utilizando-se de hardware comum e com o intuito de utilizar software padronizado diversas pesquisas foram realizadas para encontrar pontos-chave dentro do sistema computacional, que vai desda aplicação até o controlador de hardware, que pudessem afetar de alguma maneira o desempenho do sistema sobre esse hardware, melhorando ou degradando.

1.1 User level networking

No sistema de comunicação, por exemplo, tem-se como meta a máxima utilização da largura de banda da rede (*bandwidth*) com o mínimo atraso possível (*latency*). Entende por atraso, o tempo que leva para a aplicação montar a mensagem, esta passar pelo cerne do sistema, pelo sistema de comunicação e em fim pela rede física e assim chegando ao seu destino em um

ouro computador e entende-se por largura de banda a máxima taxa de transferência suportada pela rede, ou seja, quanta informação pode ser enviada ao mesmo tempo pela mesma banda física(Esta é normalmente medida em bits por segundo, *bits per second* ou bits/s). Neste caso tem-se o compromisso de garantir uma melhor performance mantendo os padrões existentes.

Com esse objetivo desenvolveu-se artifícios de software que garantissem os padrões.*User Level Comunication* (ULC) [Geoffray, Prylli e Tourancheau 1999] é uma dessas técnicas que procura melhorar o desempenho simplificando os caminhos entre as operações de emissão e recepção e afeta diretamente a maneira com a qual a aplicação interage com o sistema operacional.

Através da utilização da ULC procura-se diminuir a interação entre a aplicação e o SO, afetando assim a quantidade de cópias feitas diminuindo-as ao máximo para que tempo necessário para essas operações seja reduzido. Com a ULC reduze-se a intervenção do kernel (chamadas ao sistema (*system calls*), troca de contexto, etc) no processamento de mensagens, diminuindo drasticamente o tempo desperdiçado nessas intervenções.

Outras técnicas desenvolvidas em hardware podem ser utilizadas, como o *write combine*. Em cada cópia, os dados são armazenados numa cache interna e os endereços da cache são marcadas como *write combine*. Quando chega ao limite de 256 bits apenas uma cópia é feita . Esta técnica é mais um exemplo de uma técnica para ganhar tempo.

1.2 Protocolos Leves

A pilha TCP/IP não foi criada para uso em *clusters*. O protocolo IP permite o endereçamento e roteamento em escala global e TCP é um protocolo de

transporte complexo com avançado tratamento de perdas de pacotes. Foi desenvolvido numa época em que as redes apresentavam problemas graves de perdas de dados aonde um protocolo extremamente robusto era necessário. O TCP/IP logo se tornou o padrão para a comunicação em rede, tanto locais quanto remotos (World Wide Web). Por ser o mais popular e acessível foi um dos primeiros protocolos de comunicação a ser utilizados em aglomerados.

Um dos problemas da pilha TCP/IP é com a utilização de mensagens curtas. Isso se deve ao tamanho dos cabeçalhos TCP e IP que são de 20 bytes e o UDP é de 8 bytes [Computer Networks 2004]. Para a transmissão de dados com mensagens curtas, até 256 bytes, teríamos uma perda enorme de taxa de transferência apenas por causa desses 40 bytes (TCP+IP). Com *payloads* de 1 a 256 bytes teríamos um perda que vai de 2000% à 7% para o UDP e de 800% à 2%. Existe também um pequeno *overhead* gerado no processamento desses cabeçalhos mas quando algumas centenas de mensagens forem transmitidas pode haver um significativo aumento do tempo de processamento.

Uma maneira de contornar o *overhead* para o processamento dos cabeçalhos e aumentar a taxa de transferência é a utilização de protocolos leves ou *lightweight procotols*.

Um protocolo leve sofre modificações sobre o *standard* no qual ele se baseia (TCP/IP por exemplo) modificando o formato dos dados ou a maneira com que esses são tratados. Se uma determinada rede mostra-se segura o suficiente o protocolo poderia ignora o *checksum* e a confirmação. Caso o atraso fosse a maior prioridade poderia-se ignorar a fragmentação.

Muitas vezes um novo protocolo é criado de forma que possua apenas o

necessário para transmitir os dados de um ponto ao outro assumindo características da rede (como baixa taxa de perdas ou ordenação) de forma segura e eficiente, normalmente visando alguma aplicação ou sistema específico. Protocolos leves normalmente estão associados com técnicas de ULC visando melhorar ainda mais o desempenho (mais dados transferidos em menos tempo).

2 Sistema de Comunicação

O sistema de comunicação consiste em um mediador, que é a abstração que o EPOS faz do hardware, e um protocolo. O mediador será responsável por tirar o NIC do estado *halt* e inicializá-lo de forma com que este esteja pronto para operar no final da inicialização. Este também é responsável pela operações de envio e recebimento de datagramas, interfaceamento com a aplicação além de operações de leitura e escrita em registradores. Uma grande vantagem de utilizar-se o mediador do EPOS é que o EPOS já possiu ULC nativo, assim sendo é possível fazer com que a aplicação interaja com o NIC sem que seja necessário a criação de um software intermediário.

Para operar o NIC, Myrinet [Boden 1995], utiliza-se um MCP. O MCP é um software característico da myrinet que funciona como um *firmware* dinâmico, que é carregado na memória da myrinet em algum determinado momento ,normalmente na inicialização. A função do MCP é servir como uma ferramenta para a manipulação do processador da myrinet assim como dos seus recursos de hardware. Algumas das funcionalidades do sistema de comunicação podem ser implementadas no MCP, como é o caso do ULC e protocolos de acesso ao meio.

Quando duas aplicações se comunicam, como as que ocorrem em redes de computadores, é necessário a criação de um mecanismo que possibilite às aplicações envolvidas entender de forma clara o que está sendo transmitido por outra aplicação. Para garantir este entendimento foi criado um protocolo que suprisse todas as necessidades da aplicação e que pudesse utilizar ao máximo as qualidades da rede (baixo atraso e alta largura de banda). Para garantir a confiabilidade do procotolo foi feita uma formalização com Redes de Pertri [**?**] e utilizou-se a o simulador PIPE [**?**] como ferramenta de validação.

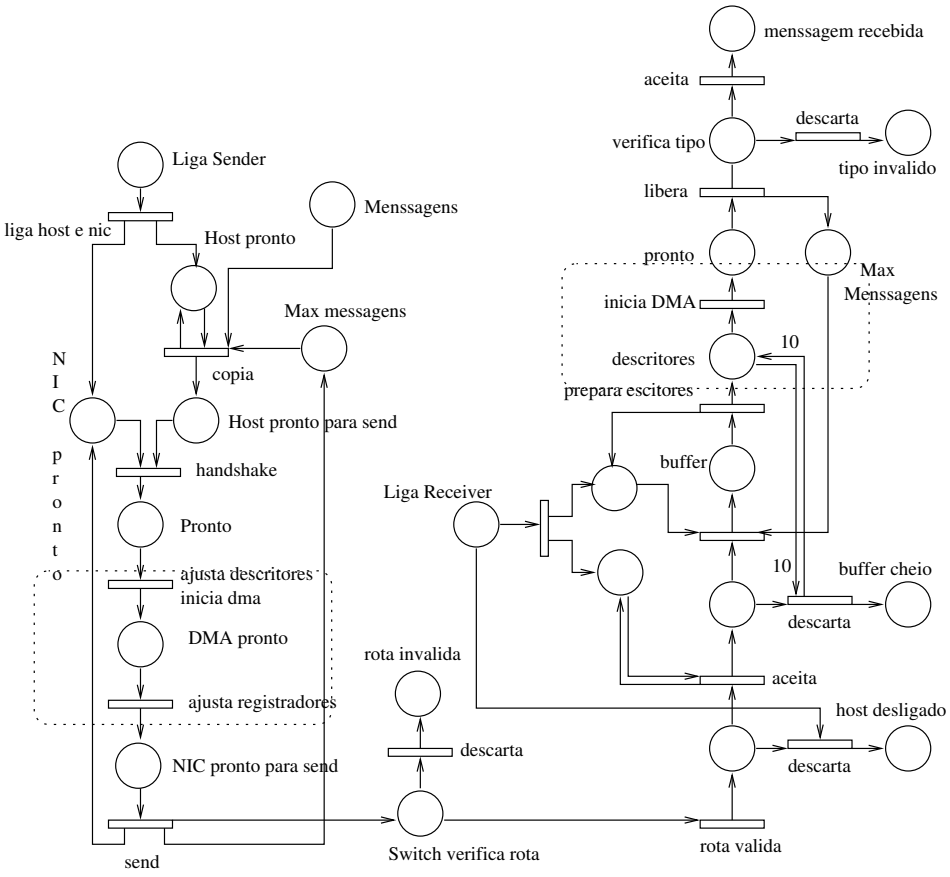

Figura 1: Sistema de comunicação no formato da Rede de Petri

Os resulatados apresentados foram:

- No Envio e no Recebimento o deadlock é esperando pois não existe uma conecção com a rede;
- A não existência de estados k-limitados é esperada em alguns estados, como *buffers* por exemplo;
- A livre escolha estendida nos mostra que existem conflitos efetivos, isto é, sempre que chegarmos a um lugar onde existam dois caminhos precisar-se-a escolher entre um deles.
- Os P-Invariantes são condições necessárias mas e suficientes para julgar a alcançabilidade da rede;
- Os P-Invariantes não cobrem todos os lugares, pois o sistema não foi projetado para ser conservativo, tendo em vista que esse é um sistema de troca de mensagens já da quantidade de mensagens a serem enviadas muda com o passar do tempo.
- A não existência de T-Invariantes está ligada com a não reinicialização do sistema.

Em resumo, a não existência de *deadlocks*, a limitação da rede, a alcançabilidade e os conflitos efetivos são condições suficientes para garantir que o protocolo assim como o sistema de comunicação que o envolve são funcionais e estão formalizados.

3 Resultados

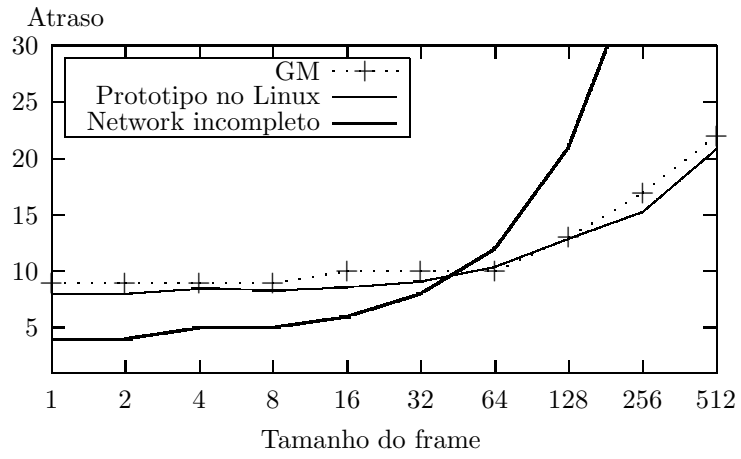

Figura 2: Resultados alcançados com troca de mensagems. Observer que a degradação de desempenhose deve ao fato do EPOS não ter *write-combine* habilitado e o sistema de comunicação do EPOS não utilizar os *dma engines*

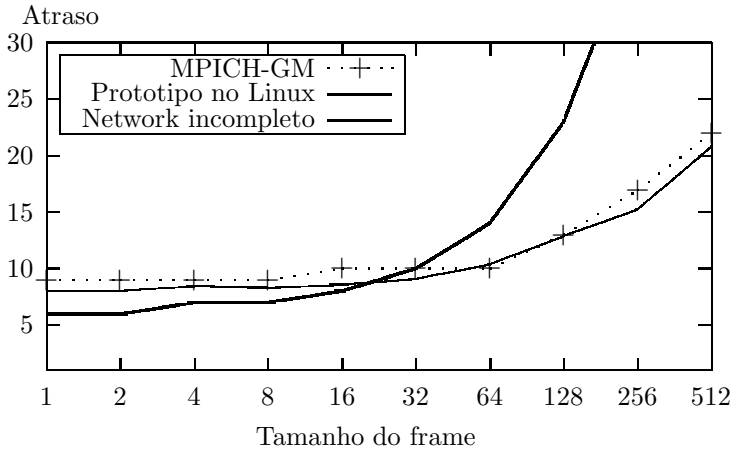

Figura 3: Utilização de uma aplicação, neste caso MPI para

4 Conclusão

A utilização de protocolos padronizados traz a vantagem da disponibilidade mas em compensação, traz problemas sérios no que diz respeito a performance. Uma escolha mau feita pode acarretar numa degradação do sistema. A escolha de protocolos corretos ou o desenvolvimento de novos modelos visando aplicações específicas tornam-se as melhores opções.

Pelos resulados apresentados na seção de resultados é possível verificar que o sistema de comunicação apresenta um pequeno atraso quando se está lidando com mensagens curtas. Essa era uma caracteristica já esperada pois mensagens longas sofrerão um atraso natural pelo atraso gerado pela grande trasnferência de dados e pelas cópias existentes. Assim é interessante que mensagens curtas sofram o mínimo de atraso possível enquanto que, mensagens longas tenham um melhor aproveitamento da largura de banda disponível.

Referências

[Baker 2000]BAKER, M. *Cluster Computing White Paper*. [S.l.], 2000.

[Boden 1995]BODEN, e. a. N. J. Myrinet – a gibabit-per-second local-area network. *IEEE MICRO*, 1995.

[Computer Networks 2004]COMPUTER Networks. [S.l.]: Prentice Hall PTR, 2004.

[Geoffray, Prylli e Tourancheau 1999]GEOFFRAY, P.; PRYLLI, L.; TOURANCHEAU, B. Bip-smp: High performance message passing over a cluster of commodity smps. 1999.