AMILCAR ROSA PEREIRA

UMA AVALIAÇÃO DO DESEMPENHO DE SISTEMAS DISTRIBUÍDOS SOBRE REDES DE ALTA VELOCIDADE

Dissertação apresentada à
Escola Politécnica da
Universidade de São Paulo para
obtenção do Título de Mestre
em Engenharia

São Paulo
Jun/2002

AMILCAR ROSA PEREIRA

UMA AVALIAÇÃO DO DESEMPENHO DE SISTEMAS DISTRIBUÍDOS SOBRE REDES DE ALTA VELOCIDADE

Dissertação apresentada à
Escola Politécnica da
Universidade de São Paulo para
obtenção do Título de Mestre
em Engenharia

Área de Concentração:
Engenharia da Computação

Orientador:
Prof. Dr.
Geraldo Lino de Campos

São Paulo
Jun/2002

AGRADECIMENTOS

Ao Prof. Dr. Geraldo Lino de Campos, pelo apoio e orientação segura em todas as fases deste trabalho;

À Fátima, pela solicitude e gentileza com que sempre me atendeu;

Ao Mário, pela camaradagem demonstrada ao colega recém-chegado;

À Prof. Dr. Liria e a todos os integrantes do LASB, pelo suporte recebido durante a realização da pesquisa;

À todos os meus amigos que contribuíram direta ou indiretamente;

Ao meu pai, pelos dias de convivência e pelo auxílio prestado em momentos cruciais desta jornada;

À minha mãe e irmãs Nádia e Vânia, pela solidariedade em todos os momentos da vida;

À Andreia, pelo carinho e constante incentivo.

Todas estas pessoas tiveram papel fundamental no desenvolvimento deste trabalho.

RESUMO

Este trabalho avalia o comportamento de um sistema operacional distribuído quando este sofre alteração da velocidade de sua rede de interconexão. Com a evolução das redes locais para velocidades de até 1 Gbps surge a dúvida: qual a vantagem que aplicações distribuídas realmente obtém com o aumento da velocidade da rede do sistema? A análise aqui realizada compara o desempenho de um sistema distribuído operando sob uma rede Fast Ethernet e sob uma rede Ethernet com a finalidade de responder a esta questão.

Os dados aqui apresentados foram obtidos através do uso do sistema operacional distribuído Amoeba e da utilização de suas primitivas de comunicação baseadas no protocolo de rede FLIP. Os números mostram que a simples evolução da tecnologia de rede, sem um trabalho paralelo no protocolo de rede, não implica numa melhora significativa da performance das aplicações distribuídas.

ABSTRACT

This work evaluates the behavior of a distributed operating system by changing its internetworking speed. With the evolution of the local network technology to speeds up to 1 Gbps the following question arises: Do distributed applications perform better using faster networks? The analysis presented here compares the performance of a distributed operating system working with a Fast Ethernet network and an Ethernet network to answer this question.

We have got the numbers presented here by using the Amoeba distributed operating system and its communications primitives based on the FLIP network protocol. The results show that the simple improvement of the network speed, without a parallel work in the network protocol, does not lead to a significant enhancement in the distributed applications performance.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Nos últimos anos, uma nova tendência tem sido notada no mercado de computadores. A relação preço/performance das estações de trabalhos e computadores pessoais tem se tornado bem menor que a relação dos computadores de grande porte. A razão para esta redução nos custos dos computadores de pequeno porte é razoavelmente simples. Em qualquer produto manufaturado os custos de desenvolvimento são divididos pelo volume produzido. O preço/performance dos computadores está melhorando 80% ao ano, enquanto que o preço performance dos supercomputadores está melhorando apenas de 20% a 30% ao ano (Anderson; Culler; Patterson, 1997).

Com isto em mãos a comunidade científica internacional vem procurando formas de substituir os supercomputadores por alguma organização agrupada de vários computadores baseados em microprocessadores. Esta forma de se organizar computadores para se formar um único sistema poderoso é chamada de sistema distribuído.

Existem atualmente duas vertentes principais de sistemas que se encaixam na categoria de sistemas distribuídos. Uma delas consiste num conjunto independente de computadores, cada um com seu módulo de memória independente, interface de rede, controladoras de disco, etc, conectados através de uma rede local, formando o que é usualmente chamado de cluster. A outra vertente consiste num agrupamento de conjuntos de processador+memória, ligados através de um barramento (baseado em chaveamento) proprietário, chamados genericamente de processadores massivamente paralelos (MPP). Esta última vertente lembra muito os antigos supercomputadores, uma vez que sua tecnologia é quase que inteiramente proprietária de seu fabricante, e o preço é também relativamente mais alto que os sistema baseados em cluster.

Os sistemas baseados em cluster são então a alternativa mais interessante para sistemas distribuídos, uma vez que a velocidade das redes locais tem conseguido atingir um patamar muito próximo a velocidade dos barramentos internos, principalmente com as redes Gigabit. Por exemplo, um barramento com tamanho de palavra de 4 bytes, funcionando a 66 Mhz, atinge uma taxa de transferência máxima

de 250 Mbytes/s. Uma rede funcionando a 1 Gbit/s consegue atingir uma taxa de transferência máxima de 128 Mbytes/s. A velocidade do barramento interno é portanto apenas duas vezes maior que a velocidade da rede local. Com a próxima padronização de rede a 10 Gbit/s, este número ultrapassará 1000 Mbytes/s. Assim um sistema baseado em rede local pode atingir um desempenho muito próximo ao desempenho de um sistema semelhante baseado em barramentos proprietários, com a vantagem de ser bem mais barato.

Com este cenário formado surge uma dúvida. Aumentar a velocidade da rede de interconexão de um sistema distribuído realmente fornece uma melhora em seu desempenho ao resolver problemas e aplicações reais? É baseado nesta pergunta que se define o principal objetivo deste trabalho, avaliar como se comporta um sistema distribuído depois de ter a velocidade de sua rede aumentada.

Algumas escolhas foram feitas com o objetivo de se definir o ambiente no qual se realizará este estudo. A partir do cluster montado em nosso laboratório formado por oito microcomputadores AMD k6 233Mhz, com 64 Mb de memória cada um, interligados por interfaces Fast Ethernet, através de um HUB 3Com, percebemos que um excelente sistema distribuído para avaliação de nossa experiência seria o sistema Amoeba, que possuí licença livre e código fonte aberto para Universidades. O Amoeba se encaixa perfeitamente em nosso cluster, uma vez que ele também foi desenvolvido para plataforma x86.

O sistema operacional Amoeba (Tanenbaum et.al, 1991) é um sistema distribuído que procura aproveitar as vantagens do custo dos computadores pessoais, e de seus processadores. Todo o poder de processamento do sistema fica concentrado numa entidade chamada pool de processadores. Assim, todos os usuários podem ter acesso a um sistema poderoso, mesmo que estejam utilizando terminais X de baixo poder de processamento.

Infelizmente o desenvolvimento do Amoeba foi abandonado em 1994, deixando lacunas de *drivers* para alguns dispositivos de hardware que apareceram durante este período, inclusive para a interface de rede baseada no chip DC21140A presente em nosso cluster.

Este trabalho toma como base portanto o Sistema Operacional Amoeba, incorporando ao seu núcleo (*kernel*) uma camada de software que possibilite a ele

utilizar a tecnologia Fast Ethernet na comunicação entre os diversos nós que formam o sistema. Isto possibilitará um estudo dos ganhos que podem ser obtidos com esta atualização, bem como vai ser possível analisar como se comporta o Amoeba ao utilizar os computadores atuais.

No capítulo 2 deste trabalho serão apresentados alguns conceitos básicos sobre o Amoeba, sobre a organização do chip controlador de rede Digital DC21140A, e sobre o *driver* FLIP desenvolvido neste trabalho para o chip. No capítulo 3 descreve-se o protocolo FLIP e os resultados de performance obtidos ao alterar a velocidade da rede de 10 Mbps para 100 Mbps. O capítulo 4 segue a mesma estrutura do capítulo 3, descrevendo o protocolo de comunicação de grupo do Amoeba, e seus resultados de performance. No capítulo 5 são apresentados os resultados de algumas aplicações distribuídas de *benchmark* que foram desenvolvidas para avaliar como a alteração da velocidade da rede afeta o *speedup* das mesmas.

1.1. Trabalhos Relacionados

Existem vários projetos em andamento que tentam melhorar o desempenho de sistemas distribuídos através do aumento da velocidade das redes de interconexão dos nós do sistema. A maioria destes projetos trabalha com mecanismos que permitem o acesso direto das aplicações às interfaces de rede como uma forma de diminuir a latência na troca de mensagens . Isto permite que estes mecanismos de comunicação alcancem uma performance melhor que protocolos de comunicação completos como o FLIP, porém não contam com funcionalidades tais como roteamento de pacotes, controle do fluxo, facilidades de *broadcast* e outras facilidades citadas no capítulo 3.

O projeto U-Net (Welsh; Basu; Eicken, 1996) da Universidade Cornell fornece às aplicações um mecanismo simples de acesso às interfaces de rede, transferindo para o nível de usuário a maioria do processamento do protocolo de comunicação, o que permite a construção de um sistema mais especializado, que elimina redundâncias e desperdícios de processamento, mas infelizmente, torna mais complexa a construção de aplicações e a manutenção do sistema. Existem diversos

outros projetos (Lauria; Pakin; Chien, 1998); (Martin et.al, 1997); (Pakin; Karamcheti; Chien, 1997) que seguem a mesma idéia do U-Net (“micro rede”), trabalhando com bibliotecas de rotinas de acesso direto às interfaces de rede.

O NOW (Anderson; Culler; Patterson, 1995) é um projeto que tem uma estrutura parecida com a do Amoeba, com a diferença que ele procura aproveitar o software já existente nas organizações, e utiliza os processadores e outros recursos de cada estação de trabalho do sistema quando não houver nenhum usuário utilizando-a. Por este motivo o NOW é chamado de rede de estações de trabalho (Network Of Workstations).

2. CONCEITOS BÁSICOS

2.1. Sistema Operacional Amoeba

O projeto do sistema operacional Amoeba (Renesse; Staveren; Tanembaum, 1989); (Tanembaum et.al, 1990) e (Tanembaum et.al, 1991) teve como foco principal a construção de um sistema totalmente distribuído**.** Procurou-se criar um Sistema Operacional em que as aplicações ficam espalhadas pelas diversas máquinas do sistema, mas o usuário tem a impressão de estar utilizando um poderoso sistema de multiprocessamento centralizado.

O Amoeba é o que se chama de um sistema operacional distribuído transparente. Num sistema deste tipo, o usuário não faz idéia de qual máquina do sistema está rodando seus processos. No Amoeba, a principal função dos terminais onde o usuário tem o acesso ao sistema é apenas fornecer uma interface de entrada e saída entre o usuário e o sistema. O Amoeba decide sozinho onde cada um dos processos disparados pelos usuários deve ser executado. É isto que diferencia um sistema operacional distribuído transparente de um sistema operacional de rede. Neste último, o usuário deve indicar em que máquina do sistema ele quer que seus processos rodem.

O software do Amoeba está divido basicamente em duas partes: um *kernel*, que roda em todas as máquinas do sistema, e uma coleção de servidores especializados que fornecem a maioria das funcionalidades comuns a todos os sistemas operacionais. O *kernel* do Amoeba possui apenas as funcionalidades básicas: entrada e saída simples, gerenciamento de memória, controle de processos e mecanismos de troca de mensagens entre processos. Por esta razão esse tipo de *kernel* é chamado de *microkernel*.

Outros serviços comuns à maioria dos outros sistemas operacionais são disponibilizados no Amoeba através de servidores específicos espalhados pelos diversos nós do sistema. Dentre estes inúmeros servidores destacaríamos como mais importantes os servidores de arquivo, servidores de diretório, servidores de data, e os

servidores de tcp/ip (o Amoeba é baseado em um outro protocolo de rede chamado FLIP que vai receber mais atenção logo adiante).

Na Figura 1 podemos identificar os elementos principais que formam a arquitetura básica do Amoeba. Todo o poder de processamento do sistema fica concentrado no pool de processadores, disponível a todos os usuários do sistema. O acesso ao sistema é fornecido através das estações de trabalho. A finalidade desta organização é aproveitar ao máximo a capacidade dos processadores do sistema.

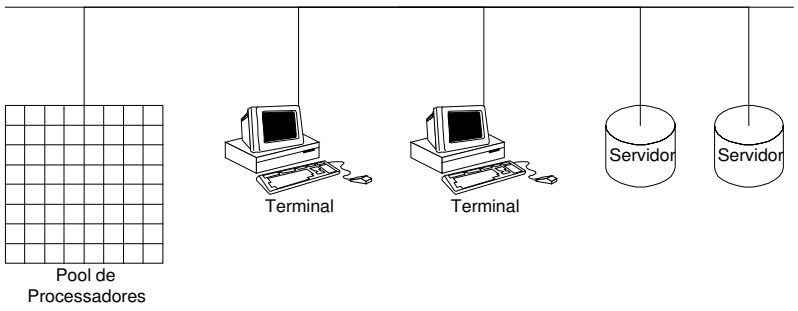

Figura 1 – Organização do Amoeba

2.2. Chip controlador de rede DC21140A

Para avaliar os benefícios de desempenho que o aumento da velocidade da rede em um sistema distribuído acarreta, escolhemos o chip controlador de rede Digital DC21140A (Estados Unidos, 1998). Este chip possui características perfeitas para o desenvolvimento deste projeto, uma vez que funciona tanto a 10 Mbs quanto a 100 Mbs. O objetivo desta seção é fornecer uma visão geral da interface do chip com o software que utiliza e controla seus serviços (em nosso caso o *kernel* do Amoeba).

O chip DC21140A é um controlador Fast Ethernet para redes LAN que funciona tanto em 100 Mbps quanto em 10 Mbps. Ele possui uma interface direta com o barramento PCI, bem como dispõe de uma série de registradores próprios para viabilizar a comunicação com o processador. O diagrama de blocos do chip encontra-se na Figura 2.

Como todo dispositivo projetado em conformidade com a especificação PCI, o controlador DC21140A possui a capacidade de efetuar transferências diretas para a

memória do sistema através do mecanismo DMA, sem nenhuma interferência do processador da máquina. A sua interface de software e suas estruturas de dados foram projetadas para minimizar a carga colocada sobre o processador durante a sua utilização.

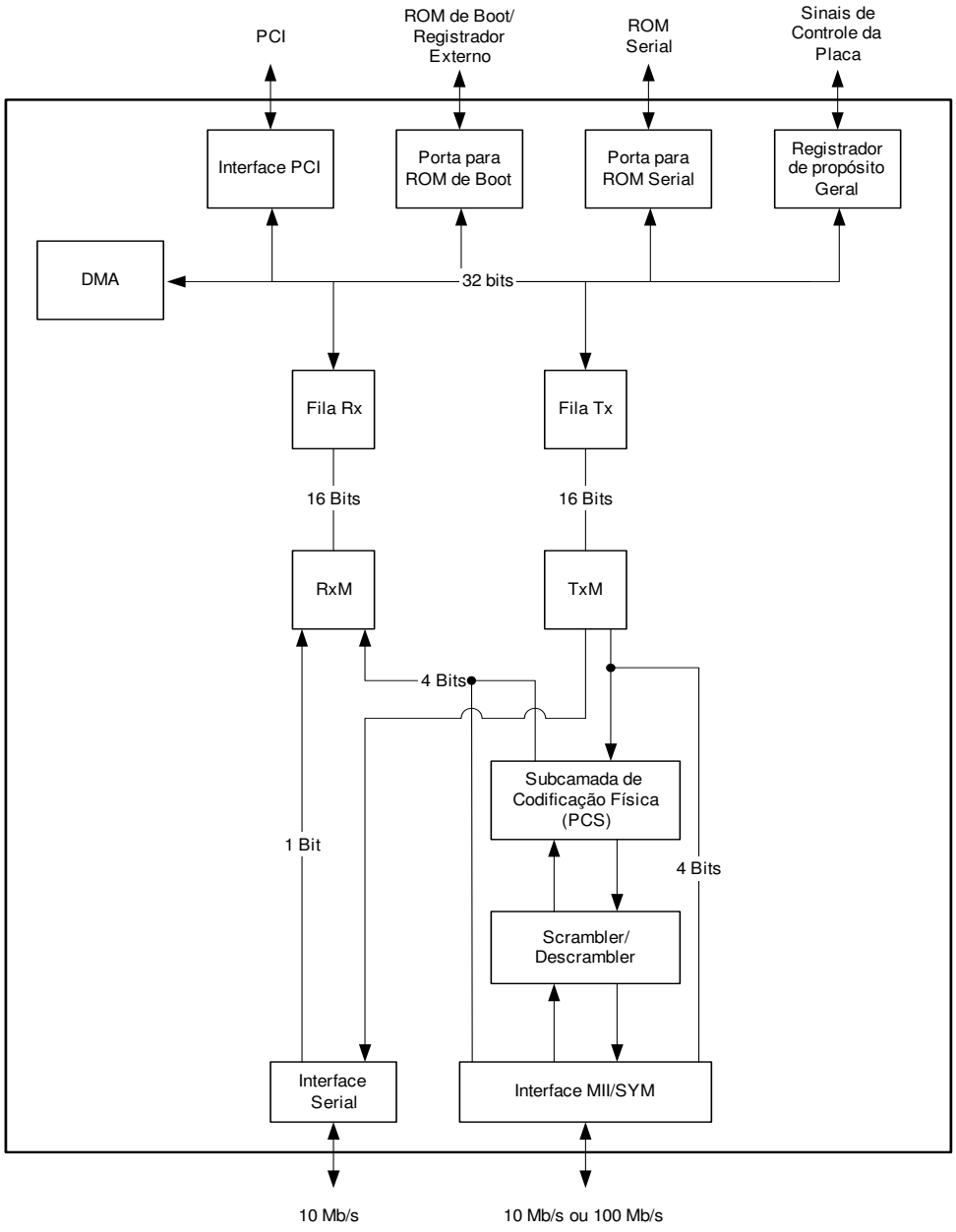

Figura 2 – Diagrama de Blocos DC21140A

O controlador DC21140A possui um conjunto de registradores de controle e status próprios (CSR) que permitem ao *driver* de software controlar seu funcionamento. Ele também se comunica com o *driver* através de um conjunto de descritores localizados em uma área da memória principal compartilhada por ambos. A função destes descritores é indicar a localização dos dados a serem transmitidos e a localização de uma área reservada para a recepção de pacotes. Este conjunto de descritores está separado em duas listas: uma para transmissão de pacotes e outra para recepção de pacotes. A localização do primeiro descritor da lista de recepção é indicada pelo registrador CSR3 e a do primeiro descritor da lista de transmissão é indicada pelo registrador CSR4.

2.3. Driver FLIP para o CHIP DC21140A

O *driver* construído para o chip DC21140A possui basicamente duas funções:

- Gerenciar os *buffers* de recepção e transmissão utilizados pelo DC21140A
- Servir de interface entre o protocolo FLIP e o DC21140A

Ao entrar em operação, o *driver* cria na memória principal da máquina duas filas circulares, sendo uma fila dedicada à transmissão de quadros e a outra dedicada à recepção de quadros. Na fila de transmissão encontra-se uma serie de descritores usados pelo chip para receber do *driver* quadros a serem transmitidos para a rede. Na fila de recepção, a função de cada um dos descritores é oposta, ou seja, eles são usados pelo chip para entregar ao *driver* os quadros que chegam pela rede.

Na figura 3 encontra-se o formato específico de cada descritor, utilizado na estrutura de fila presente na Figura 4. O campo `OWN` indica de quem é a vez de acessar o descritor, ou seja, é o mecanismo de sincronização entre o *driver* e o chip. Entretanto, a interpretação dos outros campos é diferente para descritores de transmissão e descritores de recepção.

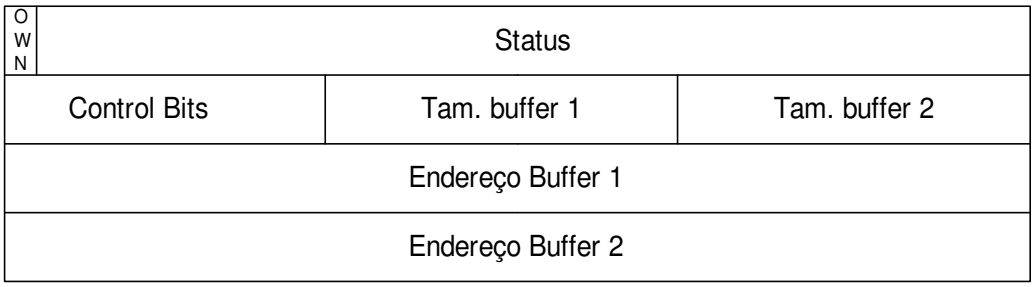

Figura 3- Formato de um descritor

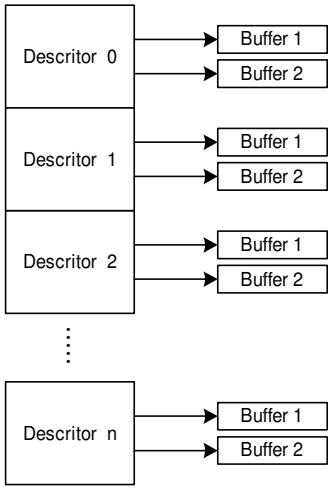

Figura 4 – Lista de Descritores

Para transmissão de quadros, o campo `status` é usado pelo chip para retornar informações ao *driver* sobre o quadro transmitido. O campo `control` é usado pelo *driver* para fornecer informações de controle ao chip sobre o quadro a ser transmitido, por exemplo, se o chip deve interromper o *driver* quando o quadro for transmitido. Os apontadores `Endereço buffer1` e `endereço buffer2` são usados para informar ao chip a localização do quadro. O chip então pode transmitir um quadro que está dividido em duas regiões de memória sem que o *driver* tenha que concatenar estes dois conjuntos de dados. Os indicadores de tamanho `tam buffer 1` e `tam buffer 2` informam ao chip a quantidade de dados existente em cada um dos dois *buffers*.

Quando estes descritores são usados para recepção de quadros, a interpretação é um pouco diferente. `Endereço Buffer 1` e `Endereço Buffer 2` apontam para

regiões de memória previamente alocadas pelo *driver* destinadas a utilização pelo chip com a finalidade de armazenar quadros recebidos da rede. `Tam Buffer 1` e `Tam buffer 2` indicam o tamanho destas duas regiões. Quando recebe um quadro, o chip preenche o campo `status` com informações sobre este quadro, como tamanho, se o quadro é de *multicast*, e também se ocorreram erros na tentativa de receber um quadro.

2.3.1. Transmissão de Quadros

No processo de transmissão de quadros, o *driver* recebe da camada FLIP um quadro a ser transmitido sem o CRC, que é calculado e adicionado pelo chip. Na seqüência, o *driver* preenche o próximo descritor de transmissão livre com informações sobre o quadro a ser transmitido, e seta o bit `OWN` indicando que é a vez do chip acessar o descritor. Na Figura 5 abaixo se encontra o pseudocódigo da rotina `send` do *driver*.

```
void send(pkt)
{
    indice = cur_tx % TX_RING_SIZE;
    tx_ring[indice].buffer1 = pkt.data;
    tx_ring[indice].count1 = pkt.tamanho;
    tx_pkts[indice].pkt = pkt;

    /* Verifica quantidade de quadros enviados
       e permite interrupção se metade da fila está ocupada */
    if(cur_tx -dirty_tx > (TX_RING_SIZE/2)      {
        tx_ring[indice].control |= INTERRUPCAO_AO_TRANSMITIR;

    cur_tx++;
```

Figura 5 – Rotina send

Ao receber o comando avisando que existem pacotes a serem transmitidos, o chip procura o próximo descritor com bit `OWN` ligado e transmite o quadro apontado

por este descritor, desligando em seguida o bit `OWN`. Se o bit interrupção do campo controle do descritor estiver ligado, o chip emite uma interrupção IRQ para o *driver*, informando que existem quadros que foram transmitidos. O *driver* então intercepta esta requisição de interrupção, e depois de verificar através dos registradores de status e controle do chip (CSR), que uma transmissão de quadro gerou a interrupção, executa a rotina da Figura 6 para liberar os descritores de quadro já transmitidos para uso em outras transmissões.

```
void transmitted()
{
    while(cur_tx > dirty_tx)
    {
        indice = dirty_tx % TX_RING_SIZE;
        libera(tx_pkts[indice].pkt); /* Libera memória usada na transmissão */
        verifica_status(tx_ring[índice]); /* Atualiza estatísticas da interface */
        dirty_tx = dirty_tx + 1;
    }
}
```

Figura 6 – Rotina transmitted

2.3.2. Recepção de Quadros

No processo de recepção de quadros, o chip recebe quadros da rede e os armazena em regiões de memória apontadas pelos descritores de recepção. A fila de descritores de recepção é montada pelo *driver* no momento em que o mesmo inicia. A rotina `init_rx_ring` que executa esta operação está listada na Figura 7.

Ao receber um quadro da rede o chip armazena-o no *buffer* apontado pelo próximo descritor livre (com bit `OWN` ligado), e envia uma interrupção IRQ ao *driver* indicando em seus registradores de status e controle (CSR) que o motivo da interrupção é a recepção de um quadro. Recebendo a interrupção, o *driver* verifica o motivo de emissão da mesma nos registradores CSR do chip, e executa a rotina da Figura 8 para repassar o quadro recebido para a camada FLIP. O código fonte completo do *driver* encontra-se no apêndice A.

```
void init_rx_ring()
{
    /* Aloca RCVPOOLSIZE pacotes de 1518 bytes */
    pool = aloca_pacotes(RCVPOOLSIZE, TAM_ETHERNET);
    cur_tx = dirty_tx = 0;
    for(indice = 0; indice < RX_RING_SIZE; indice++)
    {
               pkt = pega_pacote(pool);
               rx_pkts[i] = pkt;
               rx_ring[indice].own = CHIP;
               rx_ring[indice].count1 = TAM_ETHERNET;
               rx_ring[indice].buffer1 = pkt.data;
    }
}
```

Figura 7 – Rotina init_rx_ring

```
void received()
{
    indice = cur_rx % RX_RING_SIZE;

    /* Enquanto descritores pertencerem ao driver*/
    while (rx_ring[indice].own == DRIVER) {
        cur_rx++;
        tamanho = rx_ring[indice].count1;
        status = rx_ring[indice].status;

        newpkt = pega_pacote(pool);
        /* Em caso de falha, devolver descritor ao chip,
           descartando quadro*/
        if(newpkt == NULL || status == ERRO) {
            rx_ring[índice].own = CHIP;
            return;
        }
        pkt = decd->dec_rx_pkts[entry];
        rx_pkts[indice] = newpkt;
        rx_ring[indice].buffer1 = newpkt.data;
        rx_ring[indice].own= CHIP;
        eth_arrived(interface_id, pkt); /* Entrega quadro à camada FLIP */
        indice = decd->cur_rx % RX_RING_SIZE;
    }
}
```

Figura 8 – Rotina received

3. O PROTOCOLO FLIP

O protocolo de rede FLIP ("Fast Local Internet Protocol") foi desenvolvido em conjunto com o sistema operacional Amoeba (Kaashoek et.al, 1993). O FLIP é um protocolo de rede sem conexão, mas com um aumento de funcionalidade para suportar chamadas remotas de procedimento (RPC), ao invés da transferência de uma simples sequência de bytes (como o protocolo TCP/IP).

No projeto do protocolo FLIP uma série de requisitos que um sistema distribuído possui relacionados à comunicação entre seus processos foram observados. Seis requisitos fundamentais foram identificados: transparência, semântica específica para RPC, comunicação de grupo, segurança, gerenciamento da rede, e tratamento de redes WAN. Cada um destes requisitos é discutido abaixo:

1. **Transparência**. Um protocolo de rede transparente permite que seus endereços lógicos especifiquem um processo ao invés de um *host* do sistema. Isto significa que um processo que estava rodando em um processador possa ter sua localização física alterada (passar de uma máquina para outra), sem que os outros processos do sistema que se comunicam com ele tenham que ser notificados deste fato. Assim, o protocolo de rede vincula endereços a processos, e estes endereços são independentes da localização física de cada processo.
2. **Semântica específica para RPC**. Na maioria dos casos, sistemas distribuídos são construídos sobre o paradigma cliente/servidor. Neste paradigma, um cliente passa uma mensagem para um servidor requisitando algum serviço específico e fica aguardando sua resposta. O servidor realiza então o serviço do cliente e retorna uma mensagem com os dados pedidos. O mecanismo de comunicação usado para implementar o paradigma cliente/servidor é a chamada remota de procedimento (RPC).
3. **Comunicação de grupo**. Existe uma outra forma de comunicação além do mecanismo de requisição e resposta muito utilizado em um sistema distribuído. Este mecanismo é a comunicação de grupo, que permite que um processo se comunique com outros N processos. Muitas redes possuem suporte a

comunicação de grupo, comumente chamada de *broadcasting*. Assim, para se obter maior eficiência do protocolo num sistema distribuído, é necessário aproveitar as facilidades de *broadcasting* de uma rede física.

4. **Segurança**. Além de possuir uma boa performance na troca de mensagens, um protocolo de rede deve possuir também mecanismos que facilitem a construção de um sistema distribuído seguro. Portanto, o protocolo deve possuir um mecanismo que permita indicar ao usuário se uma determinada rede é segura ou não. Se a rede não for segura, o usuário pode decidir criptografar os dados a serem enviados naquela rede.
5. **Gerenciamento de rede**. Um outro dispositivo necessário em um protocolo de rede para sistemas distribuídos é o gerenciamento automático da rede. Em um ambiente com diversos processadores e redes, muitas vezes acontece que um processador seja desligado para manutenção ou para que a rede seja reconfigurada. Para a maioria dos protocolos atuais, casos como estes exigem a intervenção do administrador da rede para efetuar esta reconfiguração. Para contornar esse inconveniente, o protocolo de rede deve ser dotado da capacidade de se auto-adaptar a mudanças na configuração da rede e à retirada ou acréscimo de nós do sistema.
6. **Tratamento de redes WAN**. A maioria da comunicação em um sistema deste tipo ocorre entre as máquinas localizados próximas umas das outras. Apesar disso, o sistema precisa contemplar a possibilidade de comunicação com máquinas localizadas em espaços mais distantes. Ao se adotar um suporte para atender a essa eventualidade, deve-se fazê-lo de tal forma que não fique comprometido o desempenho da comunicação em nível local.

Antes do FLIP, observou-se que nenhum dos protocolos de rede mais conhecidos na época contemplava todos os requisitos citados acima. A principal falha encontrada naqueles protocolos, principalmente no protocolo IP, é que mecanismo de endereços de rede identificava uma máquina, e não um processo como o requisito de transparência exige. Assim, fazer um processo servidor migrar de uma máquina para outra se revelou uma tarefa mais complexa do que deveria ser, exigindo intervenção manual em arquivos de configuração. Diante das limitações dos

protocolos de rede existentes naquela época, começou a se conceber um novo protocolo mais eficiente. Seus projetistas tomaram como ponto de partida o atendimento dos seis requisitos já mencionados. Essa é a origem FLIP.

3.1. O FLIP

No protocolo FLIP a comunicação entre processos ocorre através do uso de uma ou mais entidades chamadas de caixa FLIP. Uma caixa FLIP comunica-se com o mundo exterior através de interfaces de rede e interfaces para processos. No centro de uma caixa FLIP encontra-se uma entidade chamada de chaveador de pacotes. Este chaveador é responsável pelo roteamento de pacotes entre as diversas interfaces. A estrutura de uma caixa FLIP é apresentada na Figura 9.

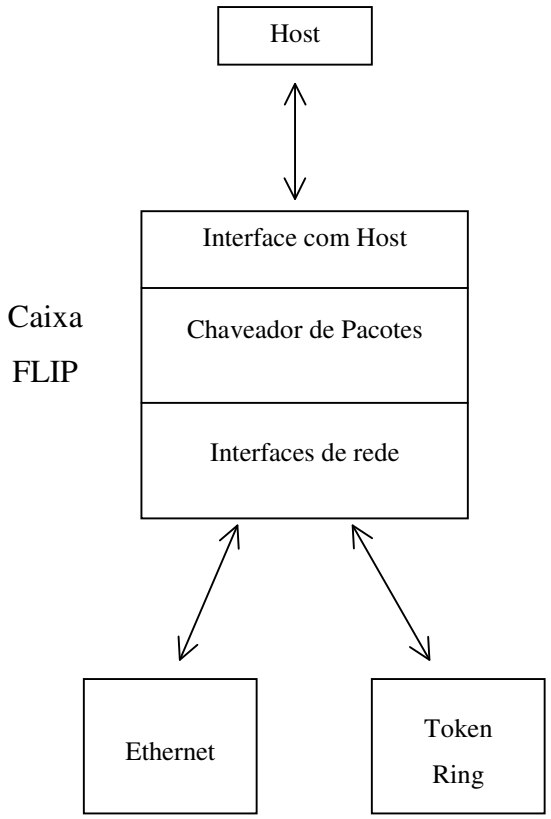

Figura 9 – Caixa FLIP

A comunicação no protocolo FLIP ocorre entre NSAPs (Pontos de Acesso aos Serviços de Rede). Um processo envia e recebe mensagens registrando um endereço NSAP na caixa FLIP através do uso da interface com o *host*. Os processos escolhem os endereços NSAP randomicamente, e cada um dos endereços escolhidos pode ser levado com o processo caso o mesmo seja transferido para uma outra máquina. Um endereço NSAP é composto de 64 bits, o que torna muito difícil a escolha de um mesmo NSAP por dois processos diferentes. Outro detalhe é que um processo pode possuir mais que um endereço NSAP. Como já foi citado anteriormente, o protocolo FLIP é um protocolo sem conexão, e as mensagens transmitidas entre NSAPs não possuem entrega garantida, ordenação, e podem ser entregues de forma fragmentada.

Na visão do chaveador de pacotes uma interface com o *host* não possui qualquer diferença em relação a uma interface de rede. O chaveador de pacotes possui uma tabela interna de roteamento que mapeia endereços NSAP a interfaces. Quando recebe um fragmento de mensagem de qualquer uma das interfaces, o chaveador verifica se o endereço NSAP de destino da mensagem está presente em sua tabela de roteamento e, baseado na entrada encontrada, entrega-o à interface indicada. Desta maneira, um fragmento de mensagem (ou a mensagem inteira) será entregue a um NSAP local, se o mesmo for entregue pelo chaveador de pacotes à interface com o host.

Na Figura 10, encontram-se as primitivas oferecidas pelo protocolo FLIP aos processos usuários através da interface com o Host. A interface consiste de 7 chamadas no sentido aplicação->interface e duas chamadas no sentido interface->aplicação.

Primitivas	Descrição
`Ifno = Flip_init(ident, receive, notdeliver)`	Reserva uma entrada na interface com o *host*. `Ident` é um identificador usado pela própria aplicação quando a interface devolve um pacote através das rotinas indicadas por `receive` (um pacote foi recebido) e `notdeliver`.
`Flip_end(ifno)`	Desaloja uma entrada reserva pela primitiva `flip_init`
`EP = Flip_register(ifno, Private-Address)`	Registra um endereço NSAP no chavedor de pacotes. Este endereço nunca vai ser removido da tabela de roteamento do chaveador de pacotes local.
`Flip_unregister(ifno, EP)`	Remove o endereço NSAP do chaveador de pacotes.
`Flip_unicast(ifno, msg, flags, dst, EP, length)`	Envia a mensagem `msg` para o NSAP `dst`. O endereço de origem da mensagem é indicado pelo parâmetro `EP`.
`Flip_multicast(ifno, msg, flags, dst, EP, length, ndst)`	Envia a mensagem `msg` à pelo menos `ndst` que estão recebendo mensagens no endereço `dst`.
`Flip_broadcast(ifno, msg, EP, length, hopcnt)`	Envia uma mensagem `msg` a todos os NSAPs distantes até `hopcnt` segmentos de rede.
`Receive(ident, fragment description)`	Um fragmento foi entregue pelo chaveador de pacotes para a interface com o host. O aplicativo identifica o endereço NSAP de destino dentre os que ele está utilizando através do parâmetro `ident`.
`Notdeliver(ident, fragment description)`	Por algum motivo um fragmento enviado pelo processo não pode ser entregue a seu endereço NSAP de destino.

Figura 10 – Interface FLIP

Um fato importante a ser destacado aqui é a maneira como um endereço NSAP é registrado por uma entidade (processo usuário). O endereço passado por um processo à interface com o *host* é um endereço privado conhecido apenas por este processo. O endereço utilizado pelos outros processos do sistema para se conectar com o primeiro é um endereço público obtido através de um função de criptografia assimétrica:

```
Endereço Público = Criptografa(Endereço privado)
```

O endereço registrado no chaveador de pacotes é o endereço público. Porém uma aplicação só consegue registrar um endereço público se a mesma possuir seu respectivo endereço privado. É praticamente impossível descobrir um endereço privado através do uso de um endereço público, uma vez que o número de possibilidades para cada endereço público é de 2^{64}.

Um processo pode enviar uma mensagem através do uso de três primitivas: `flip_unicast`, `flip_multicast` e `flip_broadcast`. A diferença entre as três é a quantidade de destinos da mensagem enviada. Se uma mensagem é passada ao chaveador de pacotes, o mesmo verifica se o endereço de destino da mesma está presente em sua tabela de roteamento. Caso positivo, o mesmo adiciona um cabeçalho FLIP a mensagem e a envia para a interface de rede correspondente. Se o endereço não for conhecido, o chaveador tenta localizar o NSAP efetuando o *broadcast* de uma mensagem especial do tipo LOCATE. O chaveador cadastra todas as tuplas <endereço NSAP destino, interface, endereço físico> que responderem a mensagem LOCATE em sua tabela de roteamento. Se a quantidade de destinos ainda não for suficiente, a rotina `notdeliver` cadastrada pela primitiva `flip_init` é invocada para informar aplicação que a mensagem não pode ser entregue.

As primitivas oferecidas pela interface com o *host* fornecem o material básico necessário para a implementação de sistemas de RPC e comunicação de grupo, fundamentais na construção de aplicações distribuídas. Protocolos como o RPC sempre enviam mensagens de reconhecimento, e devido ao fato de que as redes atuais são muito confiáveis, seria um desperdício de banda oferecer estes serviços na camada de rede.

3.2. Descrição do Protocolo

O chaveador de pacotes, unidade principal de cada caixa flip do sistema, é responsável pela manutenção da tabela de roteamento do protocolo, e pelo chaveamento de fragmentos de mensagens que trafegam na rede. Um fragmento FLIP, como em qualquer outro protocolo de rede possui basicamente duas partes distintas: o *header* e o corpo da mensagem. No *header* vão informações como endereço de origem, endereço de destino, tipo de mensagem e tamanho. Na Figura 11 encontra-se a forma geral de um fragmento FLIP.

Os campos `HopCnt Atual` e `HopCnt Max` indicam os pesos dos segmentos de rede pelos quais o fragmento trafegou desde sua origem e o peso do maior caminho permitido para o fragmento. Quando um segmento de rede passa de um segmento de rede para outro através do chaveador de pacotes, o peso do novo segmento é adicionado ao campo `HopCnt Atual` do cabeçalho do fragmento.

Figura 11 – Cabeçalho FLIP

O campo `tipo` indica qual é o tipo da mensagem. `Endereço Destino` e `Endereço Origem` indicam os endereços NSAP de destino e origem da mensagem. `Tamanho` é o tamanho do fragmento atual que pertence a mensagem identificada por

`Identificador da Mensagem`. O fragmento se encaixa na posição `Deslocamento` da mensagem. O tamanho total da mensagem pode ser obtido no campo `Tamanho Total`.

O campo `flags` informa algumas características do fragmento. As mais importantes são:

- Ordem dos bytes nos inteiros multi-bytes (*little endian* ou *big endian*)
- Permissão para rotear através de rede não seguras (requisito de segurança definido pela aplicação)
- Indicação se o fragmento foi roteado sobre redes não seguras.
- Indicação se o fragmento está retornando porque uma rota não pode ser alcançada

Entre os bits inclusos no campo `flags` citados acima, encontram-se alguns que englobam tratamento de segurança sobre as redes por onde um fragmento de mensagem trafega. É responsabilidade do administrador do sistema avaliar se um segmento de rede é confiável ou não. Faz parte da configuração de uma interface a informação que indica se o segmento de rede ao qual ela está conectada é seguro ou não. Por exemplo, a política de administração do sistema de uma determinada organização pode considerar um segmento de rede seguro se as pontas do segmento e seus nós intermediários estão localizados dentro da organização (geograficamente). Segmentos de rede que não se enquadram no caso anterior seriam então considerados não seguros.

O FLIP permite através do uso dos *flags* do protocolo, que uma aplicação defina se os dados que ela está transmitindo podem ser trafegados sobre segmentos de rede que não são seguros. Se a aplicação indica ao protocolo que ele deve trafegar uma mensagem somente sobre segmentos seguros e o protocolo FLIP não encontra um rota na rede que satisfaça este requisito, a mensagem será então devolvida através da rotina `notdeliver` para a aplicação com o *flag* da mesma indicando esta informação. A aplicação pode decidir então criptografar a mensagem e em seguida enviá-la com o *flag* requisitando roteamento sobre redes seguras desligado desta vez. Ou seja, o FLIP fornece um mecanismo que possibilita uma otimização para aplicações que necessitam trafegar mensagens confidenciais.

O funcionamento do protocolo está baseado numa tabela de roteamento mantida pelo chaveador de pacotes. Esta tabela é dinâmica e possui tuplas com os seguintes atributos:

```
<endereço NSAP, interface, endereço físico, hopcnt, seguro, age, local>
```

A tabela de roteamento possui tamanho fixo e em certos momentos pode ficar sem espaço. Quando isto acontecer, ela obtém espaço escolhendo a entrada com o campo `age` mais alto. Entradas na tabela são liberadas também quando o campo age atinge um certo valor predefinido. Uma entrada da tabela correspondente a um endereço local não poderá ser removida. Sempre que uma mensagem chega ao chaveador de pacotes, a entrada da tabela correspondente à sua origem tem seu campo `age` zerado.

Para que a tabela de roteamento possa ser gerenciada, o protocolo trabalha com seis tipos diferentes de mensagens. Na Figura 12 encontra-se uma lista com os tipos de mensagens existentes.

Tipo	Função
`LOCATE`	Encontrar a localização de um endereço NSAP
`HEREIS`	Resposta para um `LOCATE`
`UNIDATA`	Envia um fragmento Ponto-a-Ponto
`MULTIDATA`	Efetua o *multicast* de um fragmento
`NOTHERE`	Endereço NSAP de destino não é conhecido
`UNTRUSTED`	Endereço NSAP de destino não pôde ser alcançado através de redes confiáveis.

Figura 12 – Tipos de Mensagem FLIP

Sempre que o chaveador de pacotes receber um fragmento de mensagem cujo endereço de destino é desconhecido, ele efetua um *broadcast* de uma mensagem `LOCATE` contendo o endereço em questão. Se no caminho do *broadcast* um chaveador de pacotes possuir em sua tabela de roteamento uma entrada que leve ao endereço

consultado, este chaveador responde a consulta com uma mensagem do tipo `HEREIS` indicando o peso do caminho que ele conhece para o endereço.

Quando o chaveador que possui o fragmento de mensagem recebe a resposta a sua consulta através de uma mensagem `HEREIS`, ele cria uma nova entrada em sua tabela de roteamento, com peso igual ao peso retornado pela mensagem `HEREIS` adicionado ao peso do caminho até o chaveador que enviou a mensagem `HEREIS`. Na entrada vai também a interface pela qual a mensagem chegou e o endereço físico de destino da outra ponta do segmento de rede.

Outro tipo de mensagem relacionado à manutenção da tabela de roteamento é o tipo `NOTHERE`. Quando um chaveador de pacotes recebe uma mensagem de dados de uma interface de rede e o endereço de destino não é conhecido (não está na tabela de rotas), ele altera o tipo da mensagem para `NOTHERE` e envia a mensagem de volta para sua origem. Quando a caixa FLIP no caminho recebe uma mensagem do tipo `NOTHERE`, ela invalida a entrada da tabela de roteamento e verifica se existe uma rota alternativa para o destino da mensagem. Se houver uma outra rota, a mensagem é novamente convertida para uma mensagem de dados e enviada para o destino pela rota alternativa. No caso da não existência de uma rota alternativa, a caixa verifica se o endereço de origem da mensagem está na tabela, e passa a mensagem para frente em caso positivo, ou descarta a mesma em caso negativo.

Fragmentos de mensagens que não podem trafegar sobre redes não seguras funcionam de maneira análoga. Quando uma caixa FLIP recebe um fragmento de dados com *flag* de segurança ligado e não possui uma entrada na tabela de roteamento que indica uma rota segura, ela modifica o tipo da mensagem para `UNTRUSTED` e envia o fragmento de volta para sua origem.

Mensagens que chegam a interface com o *host* e possuem tipo `UNTRUSTED` ou `NOTHERE` são entregues a aplicação através da rotina `notdeliver` registrada pela primitiva `flip_init`.

3.3. Performance

Para avaliar como se comporta a performance do protocolo FLIP ao se alterar a velocidade da rede do sistema, testes foram efetuados utilizando chamadas remotas de procedimento (RPC). Os testes efetuados fornecem duas visões de performance: latência e taxa de transferência. No Amoeba a interface do protocolo RPC com as aplicações consiste das 3 primitivas descritas na Figura 13. O protocolo RPC garante entrega e ordem das mensagens. A semântica de no máximo uma execução de requisição é respeitada através do uso do campo `msgid` do *header* do FLIP.

Primitivas	Descrição
`Bufsize = trans( request_hdr, request_buf, request_size, reply_hdr, reply_buf, reply_size)`	Usada por um cliente para enviar uma requisição de serviço para um determinado servidor. A porta onde o servidor responde requisições se encontra no parâmetro `request_hdr`.
`Bufsize = getreq(hdr, buf, size)`	Usada pelo servidor para receber requisições de serviço na porta indicada dentro da estrutura `hdr`. Quando a primitiva retorna, a porta contida dentro da estrutura `hdr` é a porta usada pelo cliente.
`Void = putrep(hdr, buf, size)`	Usada pelo servidor para responder a uma requisição de serviço recebida pela primitiva `getreq`.

Figura 13 – Interface RPC

Os testes foram efetuados através do uso de um aplicativo cliente e de um aplicativo servidor. O servidor consiste de um *loop* infinito que fica bloqueado na primitiva `getreq` aguardando uma requisição de um cliente. Ao receber a requisição do cliente, o servidor envia uma resposta nula ao cliente através da primitiva `putrep`.

O cliente por sua vez mede o tempo necessário para enviar n mensagens de m bytes para o servidor através da primitiva `trans`. Ao final de seu processamento ele informa os dados de latência e taxa de transferência da operação. O parâmetro `request_buf` da primitiva `trans` contém um *buffer* de tamanho m.

3.3.1. Latência

Os testes de latência foram obtidos através do uso dos aplicativos cliente e servidor descritos no início desta seção. Na Figura 14 encontram-se os dados de latência para a rede do sistema operando a 10 Mbps. A latência obtida para mensagens de tamanho 0 é de 350 milisegundos. Para mensagens de 256 bytes a latência sobe para 550 milisegundos. Para mensagens de 8 Kbytes a latência chega ao valor de 7410 milisegundos.

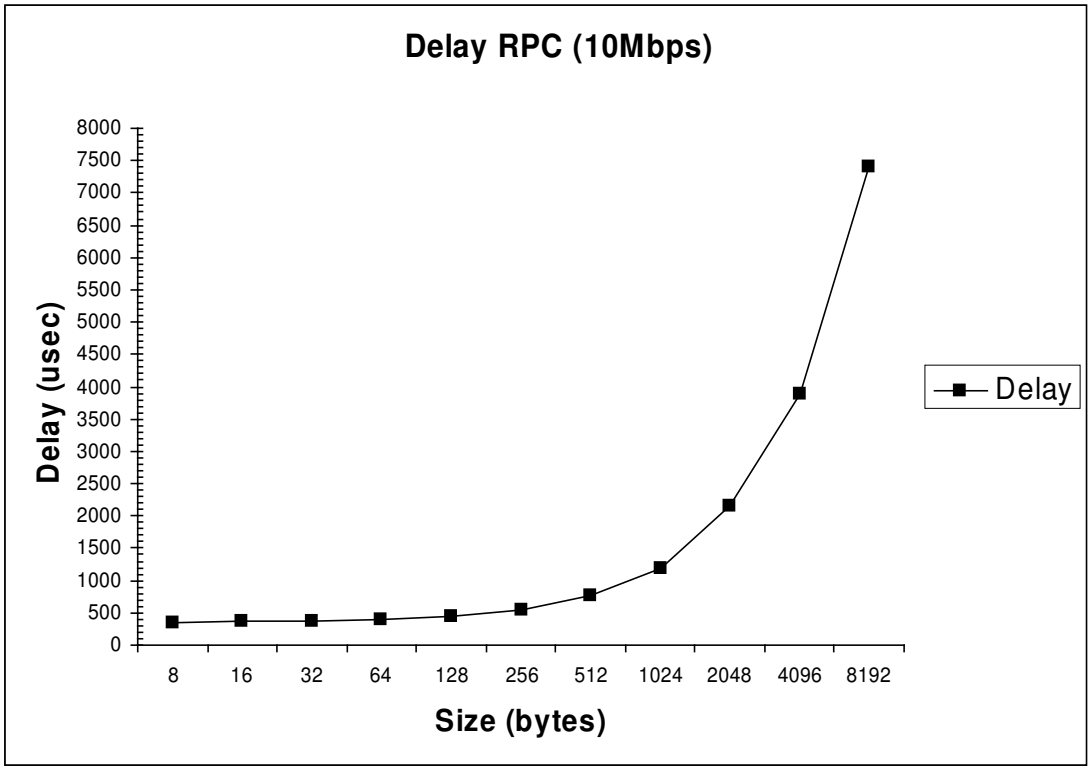

Figura 14 – Delay RPC 10 Mbps

Ao alterar a velocidade da rede para 100 Mbps obtém-se os dados presentes na Figura 15. Nesta velocidade, a latência para mensagens nulas cai para 106

milisegundos. O valor para mensagens de 256 bytes fica em 133 milisegundos, chegando a 922 milisegundos para mensagens de 8 Kbytes.

O que se nota, é que ao aumentar o tamanho das mensagens, o ganho de performance com a rede mais rápida fica mais evidente. Por exemplo, a melhora de performance para mensagens nulas fica em 3,3 vezes, enquanto que para mensagens de 8 Kbytes a melhora de performance sobe para 8 vezes com a rede Fast Ethernet.

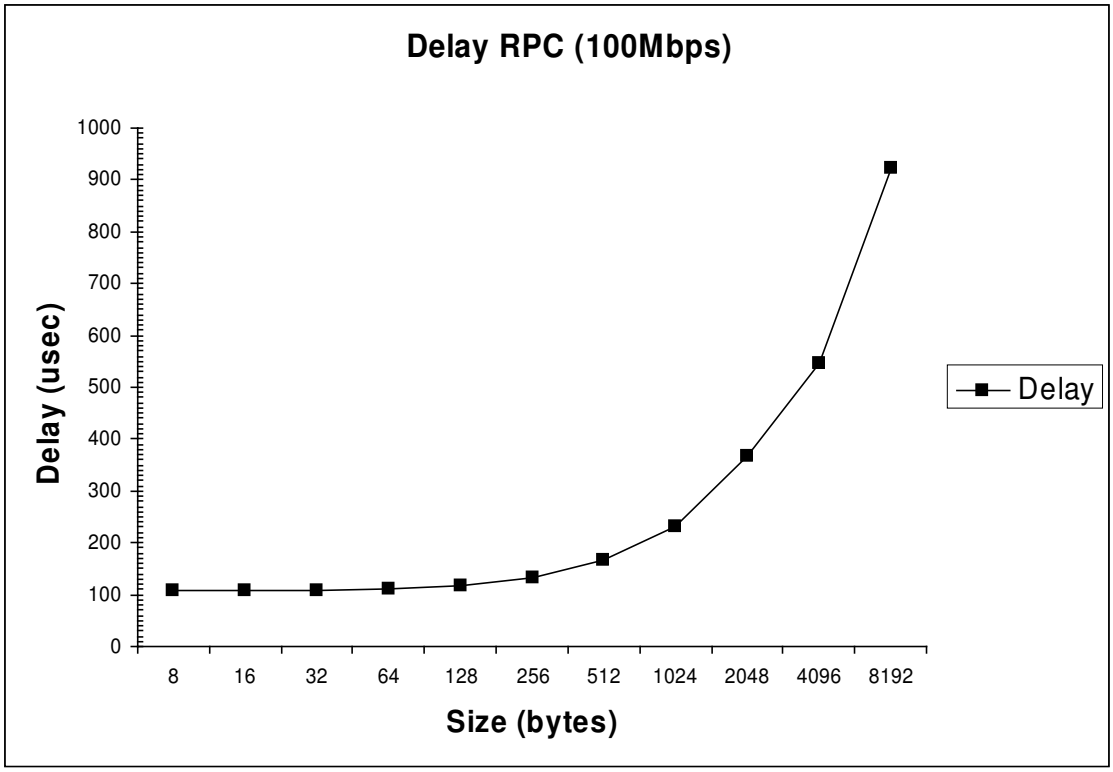

Figura 15 – Delay RPC 100 Mbps

A explicação para este fato é a seguinte. O protocolo RPC além do pacote de requisição de serviço contendo os dados utiliza mais duas mensagens. Uma mensagem enviada pela rotina `putrep` e outra mensagem de reconhecimento enviada pelo *kernel* do cliente informando ao servido que a resposta foi recebida. Assim o tempo de processamento gasto pelo protocolo RPC para as duas mensagens adicionais é o mesmo para pacotes pequenos e pacotes grandes, ou seja, quanto maior a mensagem, menos importante será este tempo de overhead no resultado

final. E com o aumento da velocidade da rede, este tempo de overhead tende a ficar ainda mais diluído (Karamcheti; Chien, 1994).

3.3.2. Taxa de Transferência

Os testes de taxa de transferência foram efetuados com os mesmos aplicativos usados no teste de latência. Com a rede operando a 10 Mbps obtém-se os resultados presentes na Figura 16. Para mensagens de tamanho 32 a taxa de transferência fica em 84 Kbytes/s. Ao dobrar este tamanho a taxa de transferência sobe para 160 Kbytes/s. Com mensagens de 1 Kbyte a taxa de transferência chega a 847 Kbytes/s. O limite alcançado pelo protocolo para 10 Mbps fica em torno de 1120 Kbytes/s com mensagens de tamanho superior a 32 Kbytes.

Ao aumentar a velocidade da rede para 100 Mbps consegue-se um ganho de performance equivalente a latência e os dados obtidos encontram-se na Figura 17. Para mensagens de 32 bytes a taxa de transferência fica em 289 Kbytes/s. Com mensagens de 64 bytes, este valor sobe para 567 Kbytes/s. Com o tamanho das mensagens chegando ao valor de 1 Kbyte a taxa de transferência vai a 4337 Kbytes/s. O limite para taxa de transferência é de 9929 Kbytes/s e é atingido com mensagens maiores que 64 Kbytes.

Algo que se nota em ambas os gráficos de taxa de transferência é que a curva possui um certo desvio com a variação do tamanho da mensagem de 1024 para 2048 bytes. Isto acontece porque é nessa faixa que se encontra o mtu do Ethernet (1536 bytes), que leva a fragmentação de mensagens. Mas como pode se notar a influencia da fragmentação é pequena na performance do protocolo.

Os ganhos obtidos pela alteração da velocidade da rede de 10 para 100 Mbps ficam em 3,5 vezes para mensagens de 64 bytes, 4,5 vezes para mensagens de 512 bytes, 5,8 vezes para mensagens de 2 Kbytes, e 8,6 vezes para mensagens de 16 Kbytes.

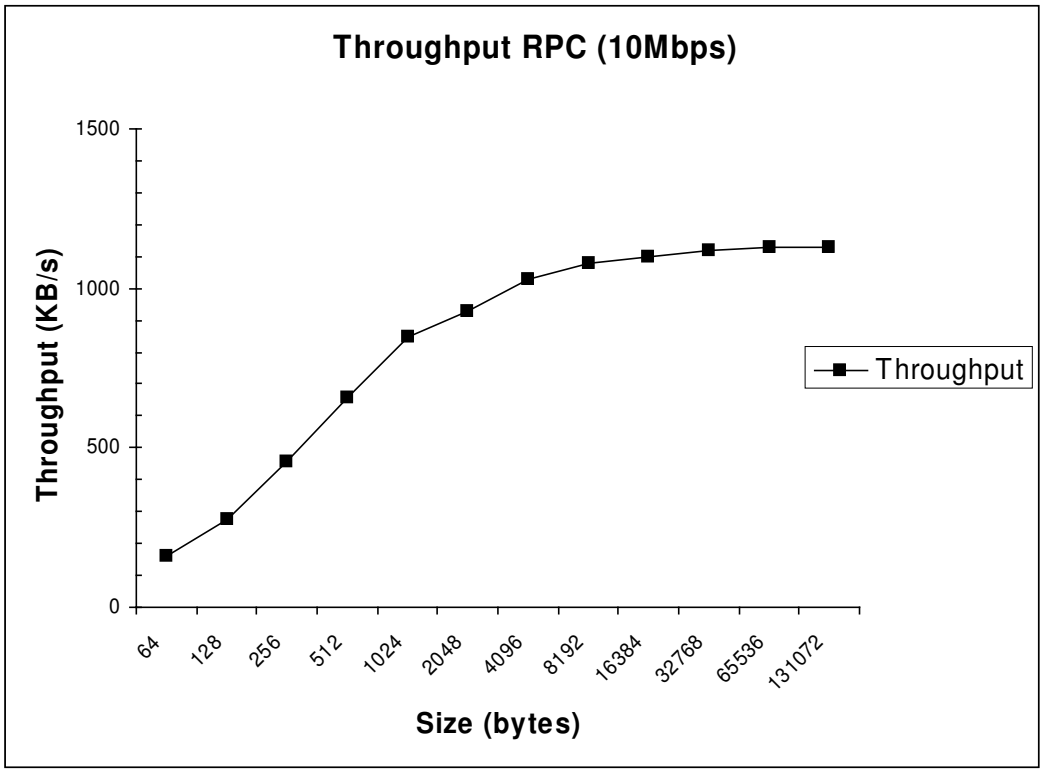

Figura 16 – Throughput RPC 10 Mbps

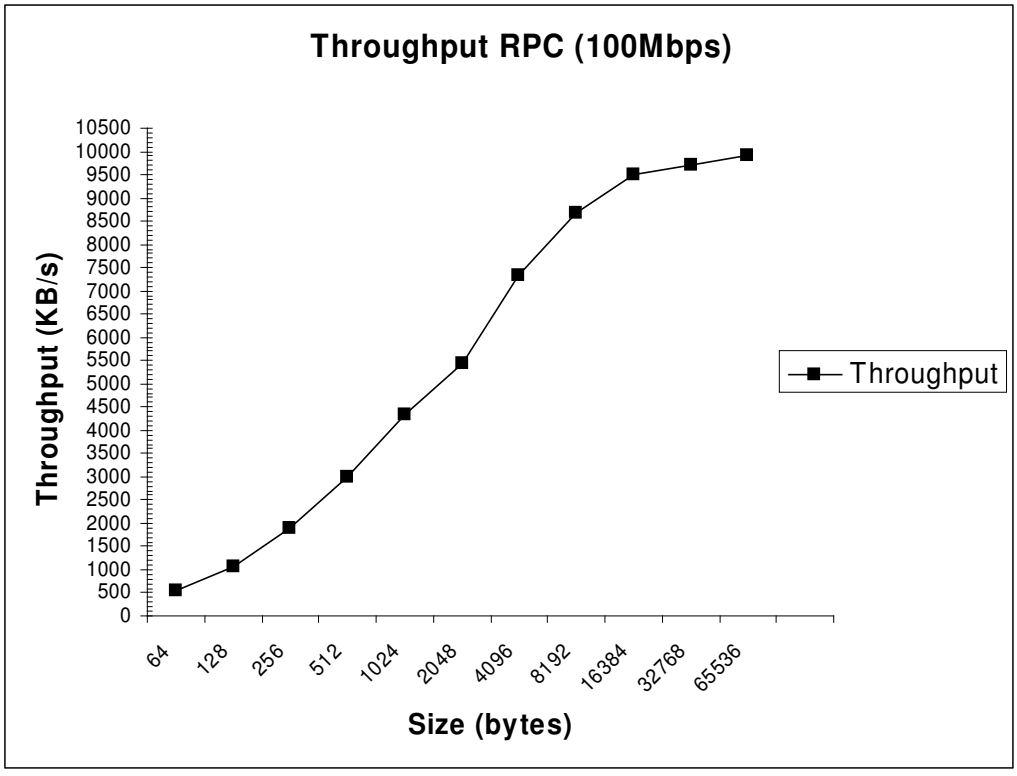

Figura 17 – Throughput RPC 100 Mbps

3.3.3. Avaliação

O que se pode avaliar dos resultados de performance obtidos neste capítulo é que aplicações distribuídas que trafegam mensagens maiores vão obter um melhor benefício da elevação da velocidade da rede do sistema.

Para mensagens pequenas o resultado ficou um pouco aquém do esperado, uma vez que o aumento da velocidade da rede foi de 10 vezes, enquanto que as aplicações que utilizam o protocolo RPC só poderão sentir uma melhora de mais ou menos 4 vezes. A melhora obtida com as mensagens maiores já se torna mais interessante, uma vez que ficou bem próxima do valor esperado de 10 vezes.

O que se nota neste caso é que o overhead do protocolo ainda é grande e portanto não existe linearidade com o aumento de velocidade da rede. Talvez com processadores mais rápidos este overhead fique um pouco mascarado, mas a idéia é que o desenvolvimento de novas interfaces de rede acompanhe o desenvolvimento de novos processadores.

4. COMUNICAÇÃO DE GRUPO NO AMOEBA

A construção de aplicações distribuídas se torna mais fácil se o Sistema disponibilizar alguma forma de se enviar uma única mensagem a ***n*** destinos diferentes. Existem muitos nomes para este tipo de comunicação entre processos. O mais conhecido é sem dúvida *broadcasting*. Mas nomes como *multicasting* e comunicação de grupo são também amplamente utilizados.

Imagine-se diversos nós de uma aplicação distribuída fazendo algum tipo de computação com a finalidade de encontrar um determinado resultado. Este resultado, uma vez computado, poderia ser utilizado por todos os outros nós para o início de uma nova etapa do algoritmo. No momento em que um determinado nó encontrar este resultado, surge a necessidade imediata de informar este valor a todos os outros participantes do processo. Este cenário é um exemplo claro de como a comunicação de grupo facilita o processo de desenvolvimento de aplicações distribuídas. Com uma simples chamada de sistema, todos os participantes do processo receberiam o resultado encontrado e o algoritmo poderia iniciar uma nova etapa.

Um sistema de arquivos distribuído tolerante a falhas também poderia tomar proveito de um sistema de *broadcasting* (Kaashoek; Tanembaum; Verstoep, 1993). Um sistema de arquivos confiável poderia manter cópias espelhadas dos dados em mais de uma máquina. Sempre que recebesse uma requisição para atualizar um determinado dado, o sistema teria que enviar uma mensagem avisando a todos os participantes desta atualização. Se um sistema de comunicação ponto a ponto fosse utilizado, n-1 mensagens teriam que ser enviadas. No caso de um sistema de comunicação de grupo, apenas uma mensagem seria suficiente.

Visando facilitar problemas como os citados acima, um sistema de comunicação de grupo foi implementado no Amoeba (Kaashoek; Tanembaum, 1992). Neste capítulo será apresentado este protocolo, e ao final desta apresentação, encontra-se uma análise de seu desempenho, discutindo seu comportamento ao alterar a velocidade da rede de interconexão.

Os projetistas deste protocolo de comunicação de grupo tentaram aproveitar ao máximo uma característica comum à maioria das redes físicas existentes: a

capacidade de *broadcasting*. Redes como Ethernet, Token Ring, Token Bus, permitem o *broadcasting* de pacotes. Assim, além de criar um protocolo para tornar a implementação de aplicações distribuídas mais trivial, os projetistas do Amoeba tentaram tornar a operação de troca de mensagens entre processos a mais otimizada possível.

A implementação de um protocolo de comunicação de grupo deve considerar uma série de fatores que vão determinar seu funcionamento e sua usabilidade. Fatores como endereçamento e a ordem com que as mensagens são entregues são decisivos para o sucesso ou insucesso de um protocolo. Neste capítulo, faremos uma descrição sumária da maneira como a comunicação de grupo no Amoeba foi projetada, as razões que levaram os projetistas a optarem por determinadas soluções, bem como uma exposição sobre o funcionamento do protocolo.

4.1. Fatores de Projeto em Comunicação de Grupo

Como mencionado anteriormente, existem diversos fatores que influenciam a eficácia do projeto de um sistema de comunicação de grupo. No projeto de comunicação de grupo aqui em estudo, foram identificados seis pontos que caracterizam um protocolo de *broadcasting* como este que estamos descrevendo:

- Endereçamento
- Confiabilidade
- Ordem de entrega
- Semântica de entrega
- Semântica de resposta
- Estrutura do grupo

4.1.1. Endereçamento

Endereçamento é o mecanismo através do qual são especificados os nós de destino de uma determinada mensagem. Pelo menos quatro maneiras de

endereçamento foram identificados durante o desenvolvimento do projeto deste protocolo.

A forma mais simples consiste em obrigar o remetente da mensagem a explicitamente especificar cada um dos destinos em que a mensagem deve ser entregue. Uma segunda maneira consiste do uso de um único endereço para o grupo todo. Além de economizar banda (comparado ao método anterior), esta última maneira também permite que se envie uma mensagem sem saber quais são os processos membros do grupo em questão.

Os outros dois métodos são menos convencionais que os anteriores, e talvez por isso não sejam amplamente utilizados em implementações de *broadcasting*. No método chamado de "endereçamento pela origem", um destino aceita uma mensagem se o endereço de origem da mesma fizer parte do grupo. No outro método, chamado endereçamento funcional, um processo aceita uma mensagem se uma função de usuário, contida na mensagem, resultar em verdadeiro.

4.1.2. Confiabilidade

Um protocolo de comunicação de grupo confiável possui basicamente um requisito: que uma mensagem enviada seja recebida por todos os membros que estejam operacionais (Jalote, 1994). No caso de perda de uma mensagem por um dos nós do grupo, o sistema deve possuir algum mecanismo automático que detecte esta falha e reenvie a mesma mensagem para este nó. O protocolo deve inclusive estar preparado para o caso em que o remetente deixa de funcionar. Se algum nó recebeu a mensagem, todos os outros membros operacionais devem recebê-la. Afora este requisito básico de confiabilidade, é possível implementar uma outra característica para tornar um sistema mais confiável. O protocolo pode conseguir se recuperar, mesmo no caso em que alguns nós membros do grupo deixem de funcionar.

4.1.3. Ordem das mensagens

Este fator determina a ordem na qual as mensagens são entregues pelo protocolo a cada membro do grupo. Ele esta altamente relacionado ao conceito de

confiabilidade, uma vez que este implica na não existência de buracos entre uma mensagem e outra. Os seguintes tipos de ordenação de mensagens são encontrados atualmente em protocolos de *broadcasting* (Vitenberg et.al, 1999):

- ***Multicast* FIFO:** Mensagens enviadas por um mesmo membro são entregues na ordem em que foram enviadas para os outros membros do grupo.
- ***Multicast* causal:** Se duas mensagens m e m' são enviadas de tal modo que m causalmente preceda m' (ou seja, m' depende de m), então todos os membros que receberem ambas as mensagens receberão m antes que m'.
- ***Multicast* totalmente ordenado:** Neste tipo de ordenação todas as mensagens são entregues na mesma ordem para os membros do grupo que estão operacionais.

4.1.4. Semântica de Entrega

O protocolo de comunicação de grupo deve possuir alguma política para chegar a um consenso de quando uma mensagem pode ser considerada entregue com sucesso. Três possibilidades foram levantadas. A primeira delas considera uma mensagem de *broadcast* entregue se pelo menos k membros receberam a mensagem. A segunda possibilidade considera uma mensagem de *broadcast* entregue com sucesso se pelo menos metade dos membros a receberam. E a última considera a mensagem entregue se todos os membros a receberam.

4.1.5. Semântica de Resposta

Semântica de resposta significa o que o membro que efetuou um *broadcast* espera receber como resposta dos membros que receberam o *broadcast*. Quatro semânticas foram identificadas: nenhuma resposta, uma única resposta, diversas respostas e todas as respostas. É importante enfatizar que este fator não tem relação com mensagens internas do protocolo de reconhecimento das mensagens. Ele está mais para o conceito de requisição/resposta utilizado no protocolo RPC.

4.1.6. Estrutura do Grupo

Este fator indica como será o comportamento do grupo com relação a seus membros e seus não membros. Grupos podem ser fechados ou abertos. Na primeira opção somente membros do grupo podem enviar mensagens para o mesmo.

Outra decisão com relação a estrutura do grupo tem a ver com a forma com que os grupos são formados. Um grupo com estrutura estática não permite a adição de novos membros após a criação do grupo. Grupos com estrutura dinâmica funcionam de maneira oposta, permitindo a adição de novos membros a qualquer momento após a criação do mesmo.

4.2. Estrutura do Protocolo

Na implementação do sistema de comunicação de grupo do Amoeba, seus projetistas fizeram diversas escolhas, todas elas feitas dentro do conjunto de opções apresentados na seção anterior. Nesta seção faremos a identificação dessas escolhas, bem como procuraremos esclarecer os critérios e motivos que nortearam os projetistas nesse processo de decisão.

4.2.1. Endereçamento

No Amoeba grupos são endereçados a partir de um endereço escolhido aleatoriamente, chamado de porta. Quando um processo deseja se comunicar com o grupo, ele manda uma mensagem endereçada a esta porta. Uma porta em comunicação de grupo é análoga a uma porta usada pelo protocolo FLIP.

4.2.2. Confiabilidade

O sistema oferece comunicação confiável para as aplicações. Se uma mensagem é perdida por qualquer motivo na rede, o sistema garante que a mensagem será reenviada sem interferência da aplicação.

No caso de falha de um processo ou máquina, o protocolo garante a manutenção do grupo com todos os outros processos restantes no mesmo. Para isto o protocolo passa por uma fase de reconstrução do grupo, durante a qual um novo grupo contendo todos os membros ainda funcionais será criado. Ainda durante a reconstrução, o protocolo garante que todos os membros entrarão em sincronia para entregar todos as mensagens enviadas antes da reconstrução do mesmo.

4.2.3. Ordem de Mensagens

O Amoeba oferece ordem total de mensagens por grupo. Na semântica implementada, as mensagens serão entregues na mesma ordem a todos os processos membros do grupo. Isto garante também a ordem causal. Já se travou um intenso debate para saber se um protocolo de comunicação de grupo deve ou não oferecer mensagens ordenadas (Birman, 1994). No projeto do Amoeba, adotou-se a teoria de que aplicações distribuídas têm implementação mais simplificada com o uso de um protocolo de comunicação de grupo que oferece ordem total de mensagens.

O esquema utilizado para que as mensagens do grupo possam ser ordenadas é baseado num seqüenciador. O *kernel* onde o seqüenciador está rodando não possui nada de especial. A única diferença deste *kernel* para os outros é que este possui um *flag* indicando-o como seqüenciador. Se o *kernel* no qual o seqüenciador está funcionando falhar, o protocolo entra numa fase de recuperação para eleger um novo seqüenciador. A função deste seqüenciador é designar números seqüências para todas as mensagens que trafegam no grupo. Este protocolo, entretanto, não garante a ordem de mensagens entre grupos distintos.

O protocolo também é baseado num esquema negativo de reconhecimento de mensagens. Normalmente o protocolo não emite reconhecimento para uma mensagem recebida; envia uma mensagem especial apenas quando percebe que deixou de receber uma determinada mensagem. Cada mensagem possui um identificador atribuído pelo seqüenciador. Quando o *kernel* de um membro recebe uma mensagem com identificador (n – 1) e posteriormente recebe uma mensagem com identificador (n + 1), ele envia uma mensagem de reconhecimento negativo para a mensagem com número de seqüência n.

4.2.4. Semântica de entrega e semântica de resposta

As primitivas de comunicação de grupo no Amoeba entregam uma mensagem a todos os membros operacionais do grupo. O usuário conta também com a semântica de "todos ou nenhum" para mensagens entregues. Ou seja, se um único membro recebeu uma determinada mensagem, todos os outros membros operacionais deverão recebê-la.

Também foi definido não implementar nenhum mecanismo de resposta no protocolo. Para isto o Amoeba oferece as primitivas de comunicação de grupo.

4.2.5. Estrutura de grupo

Ao contrário da maioria das outras implementações de comunicação de grupo (Amir et.al, 1992); (Birman; Clark, 1994); (Moser et.al, 1996) e (Renesse; Birman, Maffeis, 1996), o Amoeba trabalha com grupos fechados. Ou seja, um processo que não faz parte do grupo não pode enviar uma mensagem de *broadcast* para o grupo. A única maneira que ele tem para fazer isto é efetuar um *join* para o grupo. Outra maneira consiste em se efetuar um RPC para a porta na qual o grupo recebe requisições externas. Um dos membros do grupo vai interceptar esta requisição e vai retransmiti-la para os outros membros. Após a requisição de serviço ter sido efetuada, o membro que a recebeu envia a resposta ao cliente.

Esta implementação segue o conceito de transparência em sistemas cliente/servidor. Um cliente não deve saber como uma operação é efetuada no servidor, mas apenas quais são as operações que são disponibilizadas a ele pelo sistema. A maneira como esta requisição vai ser executada não é relevante para o cliente.

4.3. O Protocolo

O protocolo oferece 8 primitivas básicas para o usuário. Esta interface entre os processos usuários e o protocolo de comunicação de grupo é disponibilizada

através de chamadas de sistema para o *kernel* do Amoeba. Na Figura 18 encontra-se cada uma destas primitivas e uma descrição das mesmas.

Das rotinas apresentadas na Figura 18, `ResetGroup` é a que merece um pouco mais de atenção. Quando o protocolo entende que um determinado membro não está mais funcionando, ele entra num estado de recuperação. O protocolo decide que um membro falhou quando este deixa de responder ou enviar mensagens depois de um certo período. A partir deste momento, todas as chamadas a `ReceiveFromGroup` passam a retornar um erro indicando problema com um membro. Deste momento em diante é de responsabilidade de cada membro chamar `ResetGroup` para que a reconstrução do grupo possa ser iniciada pelo sistema.

O parâmetro `resistência` também tem papel fundamental no funcionamento do grupo. Ele funciona como um requisito de qualidade de serviço (*QoS*) para o protocolo, indicando quantos membros podem falhar sem que o protocolo perca mensagens. Isto é alcançado basicamente da seguinte forma. Quando um membro envia uma mensagem, o protocolo não considera a mensagem entregue até que pelo menos r – 1 membros enviem um *ack*. Assim, existem pelo menos r membros com a mensagem armazenada em seu *buffer* histórico.

O protocolo assume um meio de comunicação não-confiável. Fragmentação, reconstrução e roteamento de mensagens são feitos numa camada mais baixa dentro do *kernel*, mais especificamente pelo protocolo FLIP. O protocolo também utiliza, quando disponível, a capacidade de *broadcasting* da rede física, como uma otimização ao envio de mensagens ponto-a-ponto. Ou seja, quando um membro envia uma mensagem através da primitiva `SendToGroup`, o protocolo verifica como ele pode enviar esta mensagem para cada um dos membros. Ele envia uma mensagem de *broadcast* para os membros que podem ser alcançados desta maneira. Os outros membros recebem a mensagem através de mensagens ponto-a-ponto.

Primitivas	Descrição
`Gd = CreateGroup(port, resistência, max_grp, buf, max_msg)`	Cria um grupo. `Resistência` especifica quantos membros podem falhar sem que o protocolo perca nenhuma mensagem.
`Gd = JoinGroup(hdr)`	Associa o processo ao grupo.
`LeaveGroup(gd, hdr)`	Retira o processo do grupo. O último membro a chamar esta função faz com que o grupo desapareça
`SendToGroup(gd, hdr, buf, bufsize)`	Envia atomicamente uma mensagem para todos os membros do grupo. As mensagens são totalmente ordenadas.
`Size = ReceiveFromGroup (gd, &hdr, &buf, bufsize, &more)`	Bloqueia até a chegada de uma mensagem. `More` é usada pelo sistema para indicar a existência de mais mensagens a serem entregues.
`Group_size = ResetGroup (gd, hdr, nr_members)`	Usado para recuperar o grupo após a detecção de falha de um dos membros. Se o grupo que está sendo restaurando possuir pelo menos `nr_members` sobreviventes a função será bem sucedida
`GetInfoGroup(gd, &state)`	Retorna informações sobre o grupo.
`ForwardRequest(gd, member_id)`	Repassa uma requisição feita ao grupo para um outro membro.

Figura 18 – Interface Comunicação de Grupo

4.3.1. Descrição do protocolo

Para facilitar a descrição do protocolo, será utilizado o cenário genérico apresentado na Figura 19. Cada máquina do sistema possui apenas um membro do grupo. Isto não é um requisito para o funcionamento correto do protocolo, apenas uma forma de simplificar a demonstração do funcionamento do protocolo. O

protocolo possui dois modos de funcionamento: BB (*broadcast* seguindo por *broadcast*) e PB (ponto-a-ponto seguido por *broadcast*).

No modo PB, quando um membro envia uma mensagem M através da primitiva `SendToGroup`, o membro entrega a mensagem para o *kernel* do SO e bloqueia. O *kernel*, por sua vez, encapsula M em uma mensagem ponto-a-ponto e a envia para o seqüenciador. Quando recebe M, o seqüenciador pega o próximo número s de seqüência e faz um *broadcast* contendo s e M para todos os membros do grupo. Ou seja, as mensagens são entregues para os membros na ordem em que chegam ao seqüenciador. Quando recebe a mensagem do seqüenciador, o *kernel* que enviou M sabe que a mensagem foi entregue com sucesso e pode então devolver à aplicação o controle do sistema.

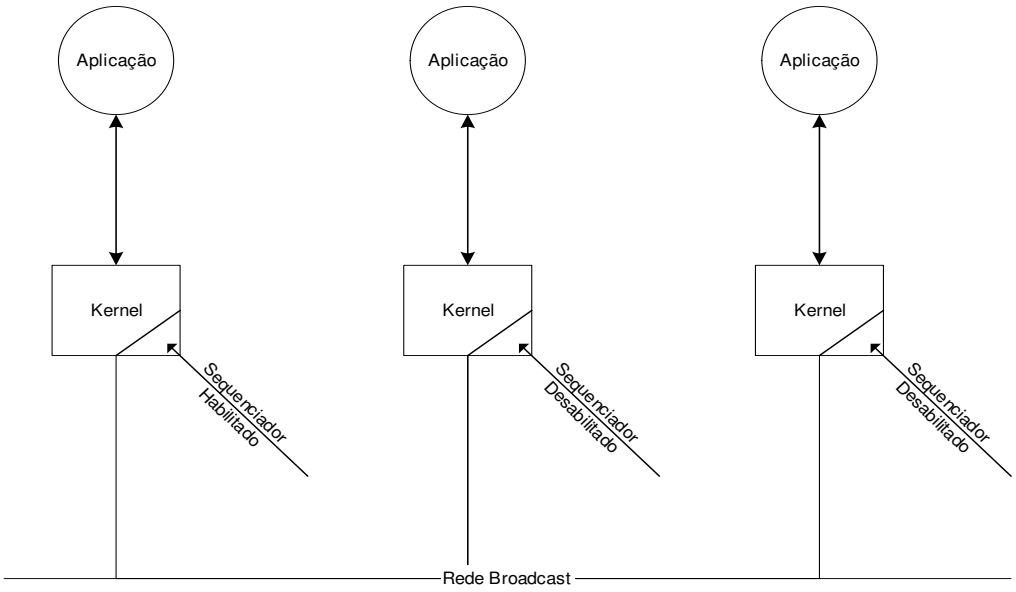

Figura 19 – Cenário Comunicação de Grupo

No modo BB, um membro ao enviar uma mensagem M, efetua primeiro um *broadcast* desta mensagem para todos os membros do grupo. O seqüenciador ao receber esta mensagem imediatamente envia uma outra mensagem contendo apenas o próximo número de seqüência s.

A diferença do método BB para o método PB é que, no último, a mensagem inteira normalmente trafega duas vezes na rede, uma até o seqüenciador e outra do seqüenciador até os membros, gastando 2n bytes de largura de banda. No método

BB, entretanto, cada *kernel* é interrompido pela interface de rede quando chega o *broadcast* e quando chega a mensagem de aceite. Assim, para mensagens grandes, o método BB é o mais indicado, e para mensagens pequenas o método utilizado deve ser o método PB.

Além de entregar mensagens usando a política de ordenação total, o protocolo ainda fornece mensagens confiáveis. As taxas de erros nas redes atuais são pequenas, mas não está totalmente descartada a possibilidade de perda de mensagens na rede. A perda pode ser ocasionada tanto por falha de transmissão na rede, como por *overflow* de *buffer* na camada de rede de enlace.

O *kernel* percebe que perdeu uma mensagem quando está esperando receber uma mensagem n e recebe uma mensagem n + 1. Como resposta à falha detectada, ele envia uma mensagem ponto-a-ponto ao seqüenciador requisitando a(s) mensagem(ns) perdida(s). O seqüenciador recupera então a mensagem n de seu *buffer* histórico e a reenvia para o *kernel* requisitante. O *kernel* dos outros membros também é dotado deste *buffer* histórico com a finalidade de que o grupo possa recuperar mensagens perdidas no caso de falha do seqüenciador.

Mas há ainda um outro problema a ser contornado. O *buffer* histórico é finito, de modo que o protocolo deve contemplar uma maneira de descartar mensagens antigas desnecessárias. Esse problema foi solucionado da seguinte maneira. Sempre que envia uma mensagem para o grupo, o *kernel* insere no cabeçalho da mensagem um pequeno número dizendo qual a última mensagem recebida. Todo *kernel* mantém para cada membro do grupo uma estrutura que, entre outras informações, contém o número de seqüência da última mensagem recebida pelo membro. Portanto, num determinado momento é fácil computar as mensagens que podem ser descartadas do *buffer* histórico. O *kernel* deve percorrer a lista de membros, verificando em cada estrutura o número de seqüência *s* da última mensagem recebida pelo membro. O menor número s encontrado indica que o *kernel* pode apagar de seu *buffer* histórico todas as mensagens com número de seqüência menor ou igual a este número.

O protocolo descrito até agora ainda não se recupera no caso de falha de um ou mais membros do grupo. Ou seja, está faltando algum mecanismo que implemente o comportamento esperado pelo parâmetro resistência existente na primitiva `CreateGroup`.

O parâmetro resistência r força o *kernel* do membro que chamou `SendToGroup` a bloquear até que ele saiba que pelo menos r membros tenham recebido a mensagem M. Para conseguir isto, o *kernel* envia uma mensagem comum ponto-a-ponto para o seqüenciador no caso do uso do método PB, ou efetua um *broadcast* de M no caso do uso do método BB. O seqüenciador, por sua vez, aloca o próximo número de seqüência s, porém ainda não aceita oficialmente a mensagem. Ao receber a mensagem M, o *kernel* de cada um dos outros membros do grupo armazena a mesma em seu *buffer* histórico e envia de volta uma mensagem de reconhecimento (*ack*) para M caso seu identificador de grupo seja menor que r. Assim que receber r reconhecimentos, o seqüenciador sabe que pode oficialmente acusar o sucesso da operação de entrega da mensagem, enviando em seguida uma mensagem de aceite para todos os membros do grupo. O funcionamento do método PB está ilustrado na Figura 20, enquanto que o método BB está ilustrado na Figura 21.

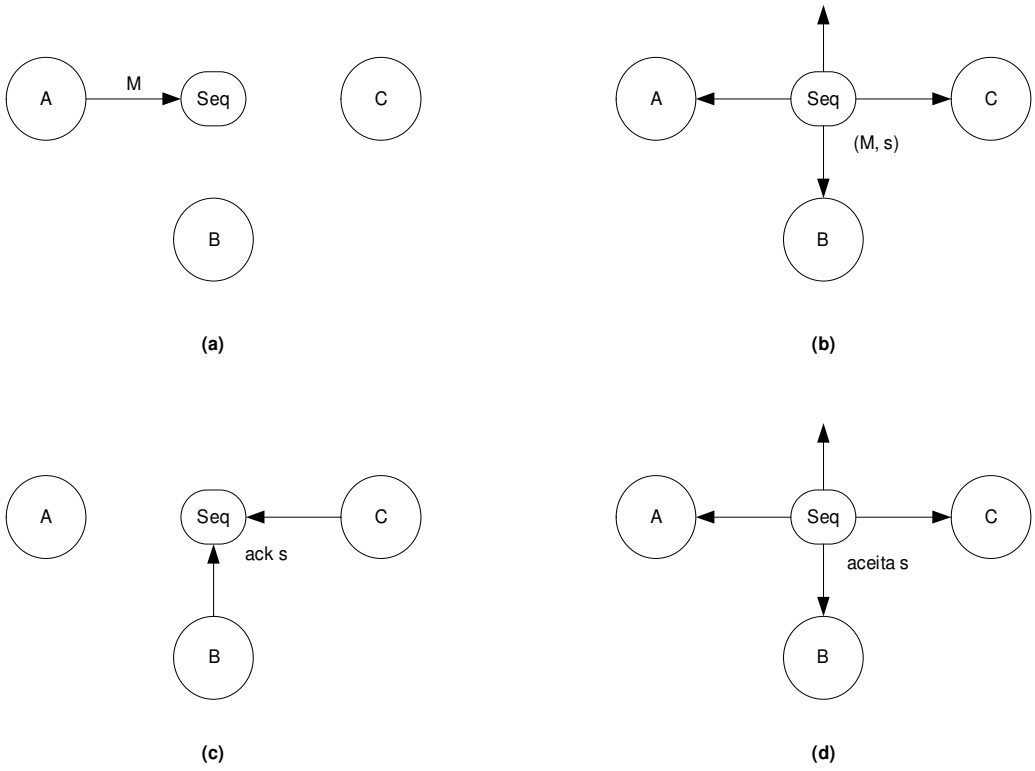

Figura 20 – Método PB

O protocolo garante através do mecanismo explicado no parágrafo anterior, que cada mensagem do grupo está armazenada no *buffer* histórico de pelo menos r membros mais o seqüenciador. Assim, mesmo que haja falhas de até r membros, é garantido pelo protocolo que durante a fase de recuperação grupo, todos os membros restantes terão acesso a todas as mensagens do grupo.

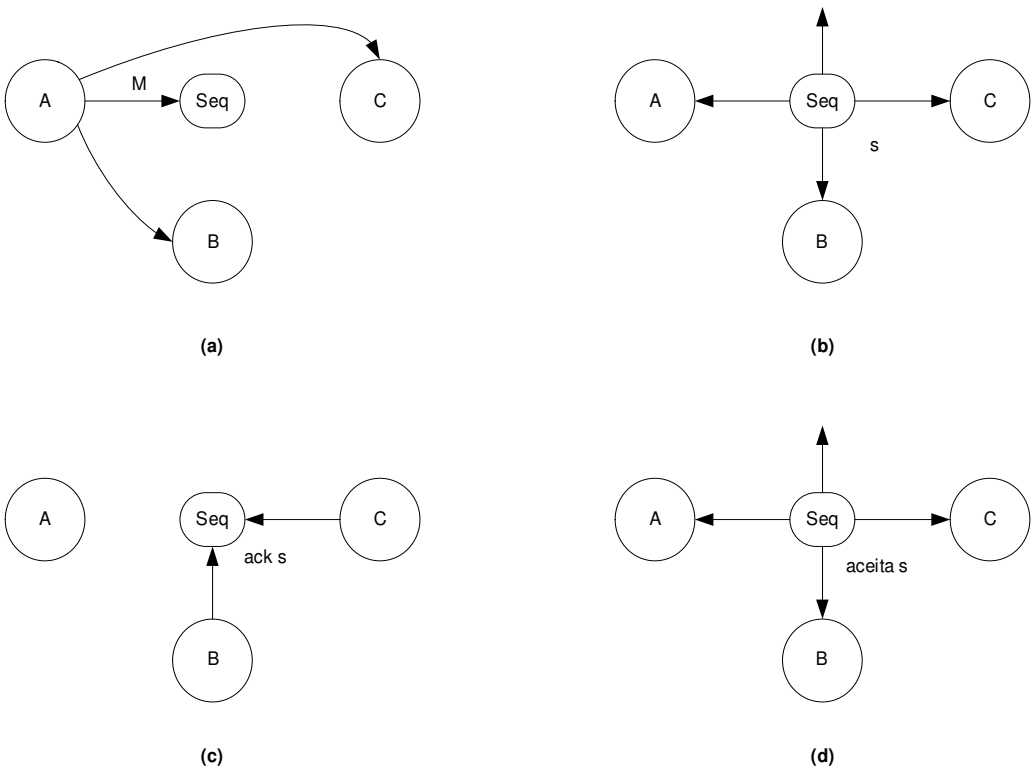

Figura 21 – Método BB

4.4. Performance

Para avaliar como se comporta o sistema de comunicação de grupo do Amoeba numa situação em que a velocidade da rede do sistema foi alterada, executamos testes para avaliar o desempenho do mesmo nas duas condições (10 Mbps e 100 Mbps). Os testes executados forneceram medidas de desempenho de latência e taxa de transferência.

4.4.1. Latência

Para obter dados de latência utilizou-se um aplicativo cuja arquitetura consiste de um membro efetuando chamadas *SendToGroup*, e n membros efetuando chamadas a ReceiveFromGroup com o intuito de receber as mensagens enviadas pelo primeiro membro.

Na Figura 22 encontram-se os resultados de performance da rede funcionando a 10 Mbps e o protocolo de comunicação de grupo funcionando no modo PB (ponto-a-ponto seguido por *broadcast*). Como é possível observar, o *delay* segue a tendência de não variar com o aumento do número de membros receptores. E também conforme o esperado, a latência aumenta linearmente com o crescimento do tamanho dos pacotes. A menor latência obtida é de 302 microsegundos para mensagens com tamanho 0 e dois membros. Para mensagens com 8000 bytes[1] a latência ficou em torno de 14,5 milisegundos.

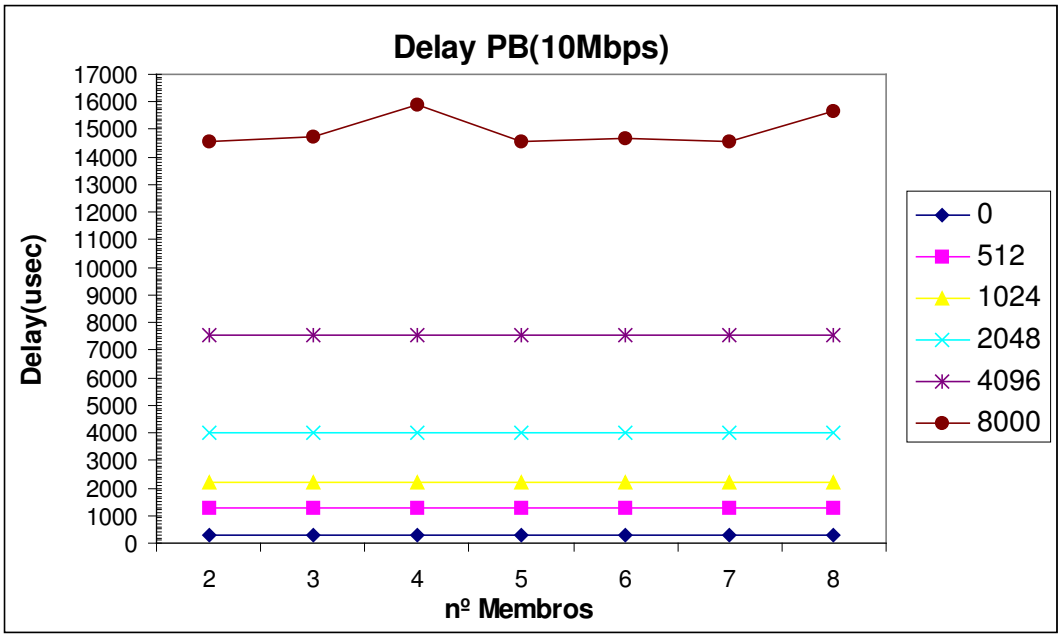

Figura 22 – Delay PB 10 Mbps

[1] 8000 bytes é o tamanho máximo permitido pelo protocolo

Na Figura 23 encontram-se os resultados para este mesmo teste de latência, porém desta vez com o protocolo funcionando no modo BB (*broadcast* seguido por *broadcast*). A latência para mensagens de tamanho 0 ficou um pouco maior que no método PB, em torno de 310 microsegundos. Porém com o crescimento do tamanho das mensagens, o protocolo obtém resultados bem melhores com 7,5 milisegundos de latência para mensagens de 8000 bytes. Na hipótese de alteração do número de membros, o comportamento segue o padrão do caso anterior, com variação praticamente nula.

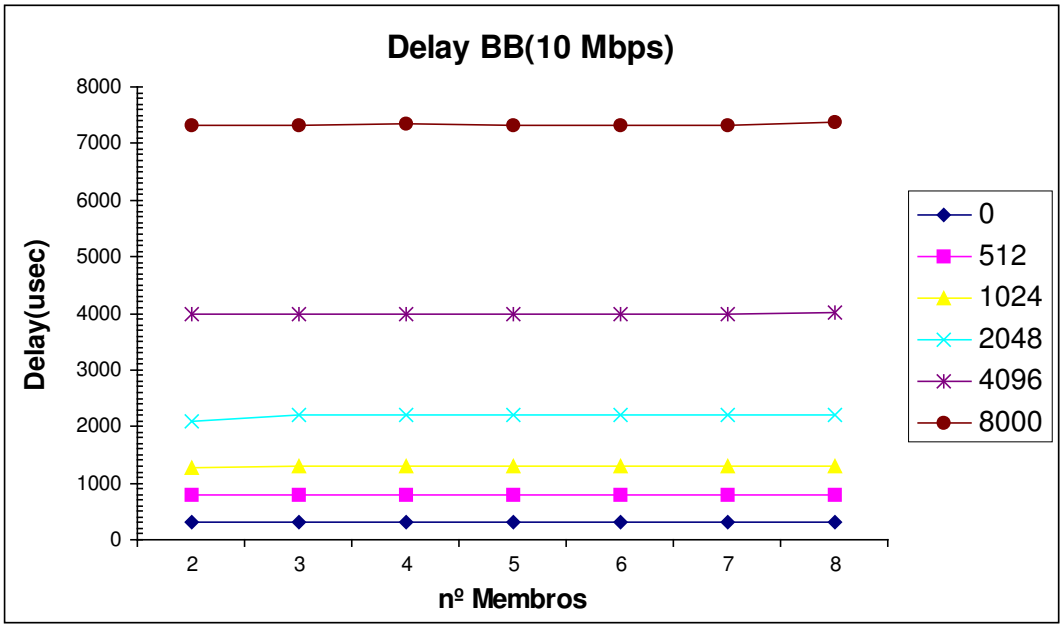

Figura 23 – Delay BB 10 Mbps

Ao efetuar a mudança da velocidade da rede para 100Mbps, obtém-se o resultado apresentado na Figura 24 para o método PB. Para mensagens com tamanho 0, a latência fica em torno de 120 microsegundos. Este resultado é apenas 13% maior que a latência do protocolo RPC. Para mensagens com 8000 bytes, a latência fica em torno de 4 milisegundos. Com o aumento da velocidade da rede, a melhora de performance fica em torno de 2,5 vezes para mensagens nulas, e de 4 vezes para mensagens de 8000 bytes. O melhor resultado fica para mensages de 4 Kbytes, cujo

aumento foi de 5 vezes. O motivo para esta diferença está no *overhead* do protocolo, que fica mascarado para mensagens grandes face ao aumento da proporção de tempo em que a transmissão da mensagem fica dependente do meio físico. Pela mesma razão o aumento de velocidade não é linear com o aumente de velocidade da rede.

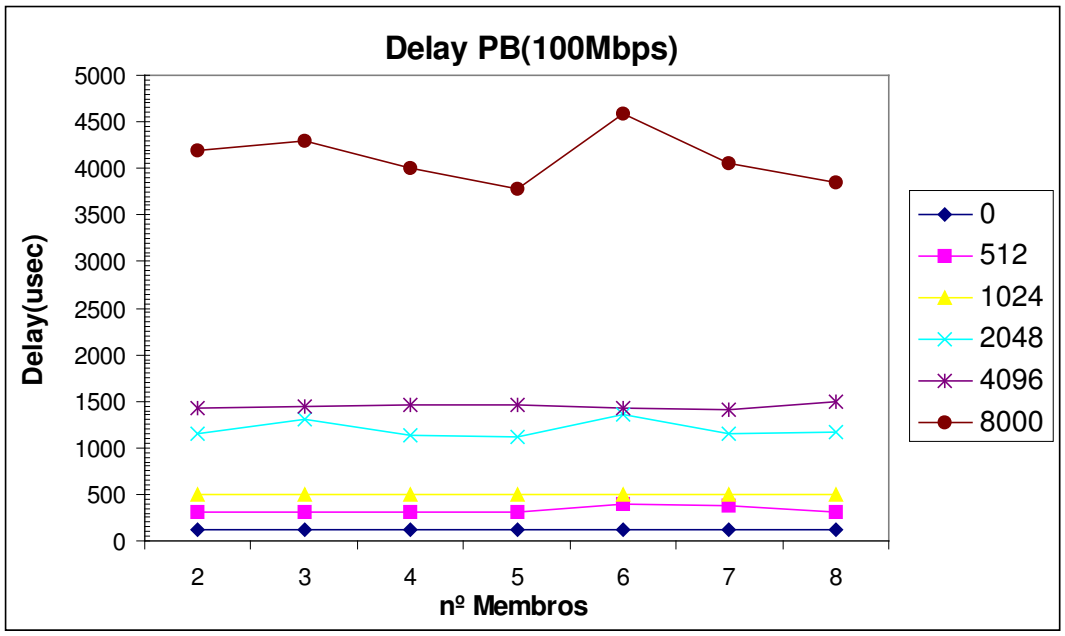

Figura 24 – Delay PB 100 Mbps

Para o método BB, com a rede funcionando a 100 Mbps, os resultados estão apresentados no gráfico presente na Figura 25. Para mensagens nulas a latência ficou em 150 microsegundos, duas vezes melhor que a latência para 10 Mbps. Para mensagens de 8000 bytes a latência ficou em 1,4 milisegundos, uma melhora de 5 vezes.

É interessante notar aqui que o método BB possui uma melhora de magnitude 3 em relação ao método PB para mensagens de 8000 bytes em 100 Mbps. A primeira impressão é que o protocolo BB aproveitou bem melhor o aumento de velocidade da rede do que o método PB. Mas isto não é verdade. O que realmente acontece é que os dados para o método PB estão muito ruins O padrão que se segue em todos os outros casos (velocidade 10 Mpbs e 100 Mbps, e mensagens até 4096 bytes) é que o método BB é duas vezes mais rápido que o método PB para mensagens maiores que 512 bytes. A razão para esta queda de desempenho para mensagens de 8000 bytes no

método PB é que o número de colisões aumenta muito, pois cada mensagem tem que trafegar duas vezes na rede antes de ser entregue a cada membro. Uma vez do emissor até o sequenciador, e outra do sequenciador até todos os outros membros (*broadcast*).

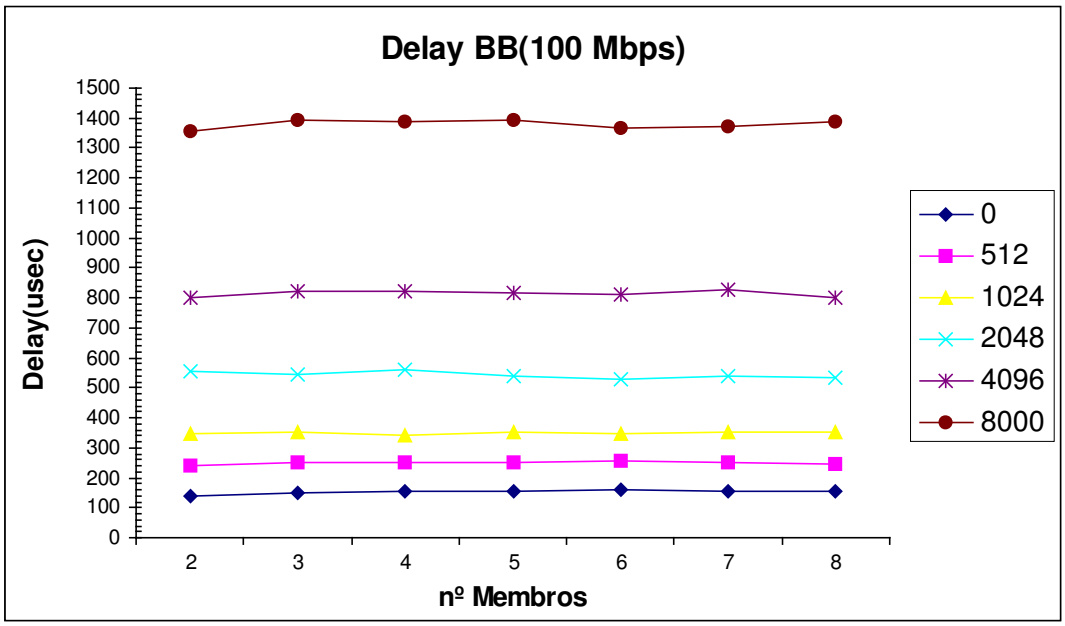

Figura 25 – Delay BB 100 Mbps

Fica claro que os tempos de latência do protocolo BB são expressivamente melhores que os tempos do protocolo PB. Por que então não usar somente o método BB? Por um motivo muito simples. As aplicações são interrompidas duas vezes no método BB, dobrando o tempo gasto em tratamento de interrupções, escalonamento de processos, *overhead* de protocolo, etc. Como se vê, os dois métodos (protocolos) apresentam vantagens e desvantagens, de modo que a escolha entre eles não é tão simples. Face a isto, antes de optar por um dos dois sistemas, o programador deve decidir se a velocidade de transmissão de mensagens grandes é mais importante que a interrupção do processamento do aplicativo.

4.4.2. Taxa de transferência

Os dados de taxa de transferência foram obtidos com todos os membros do grupo enviando mensagens para o grupo inteiro ao mesmo tempo. Cada membro possui duas linhas (threads) de execução. Uma responsável pelo envio das mensagens, e outra responsável pela recepção das mensagens do grupo. Os testes foram executados com cada um dos n membros enviando 10000 mensagens.

Os dados do método PB com a rede operando a 10 Mbps encontram-se na Figura 26. Diferentemente dos dados de latência, o aumento no número de membros resulta numa pequena melhora na taxa de transferência do protocolo. Para mensagens nulas, por exemplo, a taxa de transferência pula de 2200 msgs/s para 3000 msgs/s em grupos de 2 e 8 membros respectivamente. Porém, com o aumento no tamanho das mensagens esta tendência tende a se inverter, pois o número de colisões no meio físico aumenta consideravelmente. Para grupos de 8 membros, com mensagens de 8000 bytes, a taxa de transferência fica em 60 msgs/s.

Já no caso do método BB para a rede operando 10 Mbps, os resultados obtidos encontram-se presentes na Figura 27. Para mensagens pequenas ocorreu o esperado, registrando-se números bastante parecidos com a taxa de transferência do método PB. Para mensagens de tamanho 0, o protocolo apresentou taxa de transferência de 2291 msg/s para grupos de 2 membros e de 3217 msg/s para grupos de 8 membros. Ao aumentar o tamanho das mensagens, o método BB registra uma melhora de 35% na taxa de transferência em relação à performance do método PB. Para mensagens de 8000 bytes, a taxa de transferência fica em torno de 80 msg/s.

Quando a velocidade da rede é alterada para 100Mpbs, percebe-se um aumento considerável na performance do protocolo. A Figura 28 apresenta os resultados para o método PB. A maior taxa de transferência obtida para este método fica em 14170 msg/s para mensagens de tamanho 0 e grupos de 7 membros. Para mensagens de tamanho 8000, o protocolo apresenta uma taxa de transferência em torno de 500 msg/s.

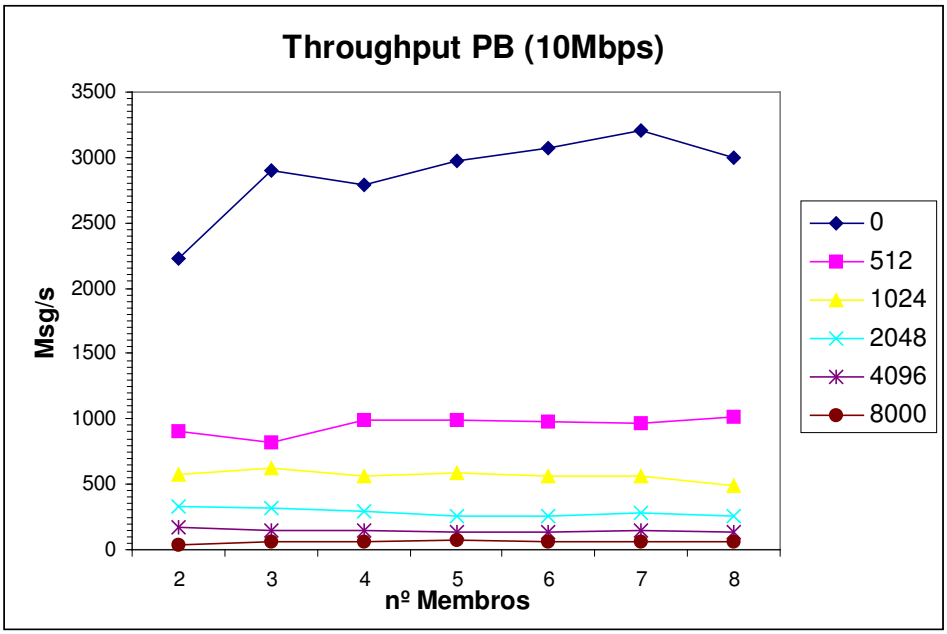

Figura 26 – Throughput PB 10 Mbps

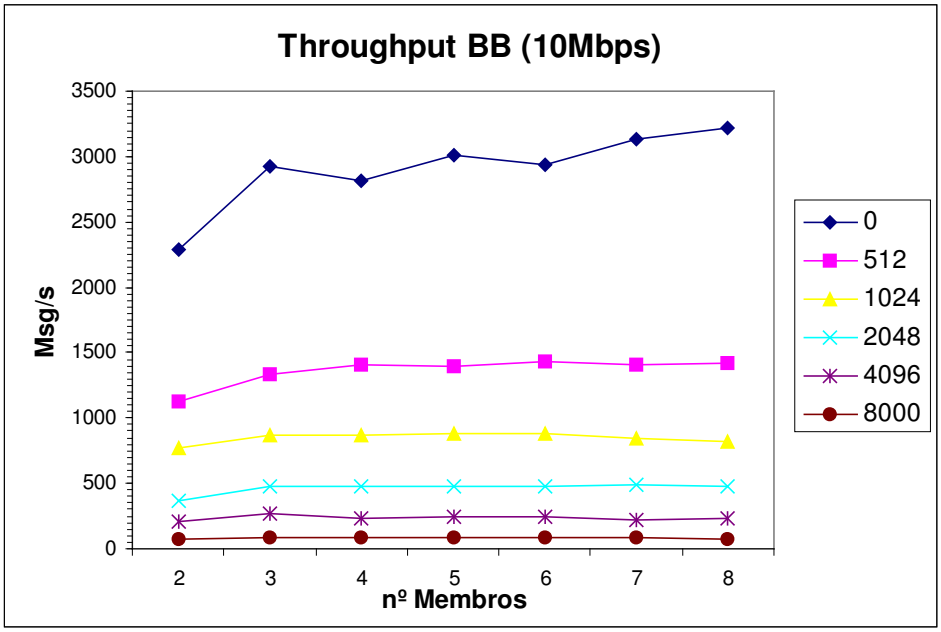

Figura 27 – Throughput BB 10 Mbps

Um fato que se nota nas curvas apresentadas nesta figura em comparação às curvas apresentadas pelo mesmo método funcionando a 10 Mbps, é que a melhora de

performance mediante o aumento do número de membros no grupo fica bem mais evidente com a rede mais rápida. Isto ocorre porque a rede cumpre seu papel de transmitir os dados com maior velocidade, enquanto o tempo de *overhead* do protocolo dentro do *kernel* permanece o mesmo. Assim, o resultado final pode ser traduzido pela seguinte equação: quanto mais mensagens cada *kernel* envia, maior será o percentual de ocupação da rede. Porém esta melhora tem um limite, pois o aumento do número de colisões no meio físico tende a diminuir a performance do protocolo. Um exemplo deste fato está no método PB para mensagens de tamanho 0 e grupos contendo 8 membros.

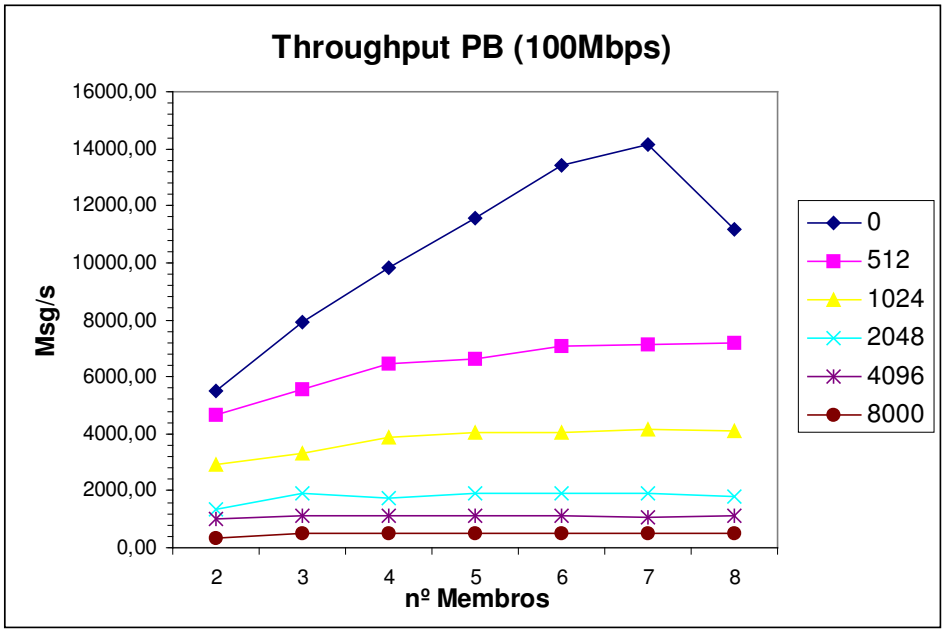

Figura 28 – Throughput PB 100 Mbps

Para o método BB, com a rede a 100 Mbps, os dados obtidos encontram-se na Figura 29. Os números obtidos também são sensivelmente melhores que os obtidos para a rede operando a 10 Mbps. Para mensagens de tamanho 0 e grupos de 8 membros, a taxa de transferência ficou em 14678 msg/s. Para mensagens de tamanho 8000, a taxa de transferência ficou em torno de 700 msg/s. Comparados aos números do método PB para esta mesma velocidade, em média a performance deste método é 35% superior em relação à operação da rede a 10 Mbps.

Comparando as performances do protocolo, na situação de aumento da velocidade da rede, percebe-se uma melhora de 5 vezes para mensagens de tamanho 0 e de praticamente 9 vezes para mensagens de tamanho 8000. Ou seja, o protocolo consegue aproveitar melhor a velocidade superior da rede para mensagens grandes. Porém, mesmo para mensagens pequenas, os números obtidos são muito interessantes e mostram que, ainda que operando com um overhead alto, a velocidade do meio físico influencia bastante na performance final do sistema de comunicação de grupo.

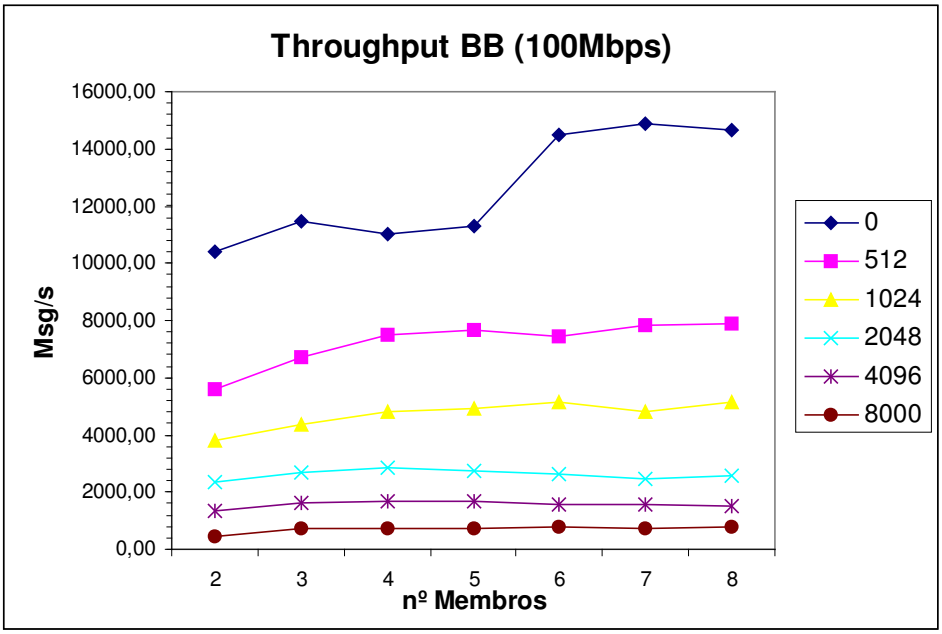

Figura 29 – Throughput BB 100 Mbps

5. APLICAÇÕES

Neste capítulo serão apresentados os desempenhos de algumas aplicações distribuídas para avaliar o comportamento das mesmas quando submetidas a uma alteração na velocidade da rede física. Foram escolhidos três problemas conhecidos cujas soluções baseadas em algoritmos paralelos são utilizadas na análise de desempenho de sistemas baseados em cluster. Os problemas escolhidos são: multiplicação de matrizes, ASP "All-Pairs Shortest Paths" e TSP (Problema do Caixeiro Viajante) (Wikipedia, 2002). A descrição de cada um destes algoritmos juntamente com seus resultados de desempenho segue nas próximas seções.

5.1. Multiplicação de Matrizes

A paralelização do algoritmo seqüencial que efetua a multiplicação de duas matrizes *A* e *B* é conhecida como um dos exemplos mais perfeitos de distribuição de tarefas entre *N* processadores. A operação de multiplicação de duas matrizes é executada através da fórmula abaixo:

$$A \times B = C \mid C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

Nesta fórmula *A* e *B* são duas matrizes, e *n* é o numero de colunas da matriz A. Este número deve ser igual ao número de linhas da matriz *B*. A matriz *C* resultante *n*x*m* é uma matriz com número de linhas *n* igual ao número de linhas da matriz A e número de colunas *m* igual ao número de colunas da matriz B. Na fórmula acima é possível verificar que os cálculos dos elementos da matriz *C* podem ser realizados de maneira independente. Desta maneira até nxm processadores podem ser utilizados para efetuar a computação da multiplicação das matrizes.

Na implementação que foi feita *p* processadores são utilizados. Um dos processadores é utilizado para coordenar a operação. Inicialmente o processador mestre divide a matriz *C* em *p*-1 conjunto de linhas e designa cada um destes

conjuntos para cada um dos outros processadores. Cada um dos conjuntos contém uma seqüência de linhas da matriz *C*. O processador mestre contém também as matrizes de entrada A e B, e no fim da computação deverá conter a matriz resultante *C*.

Quando um dos processadores recebe um conjunto de linhas de C para calcular, realizando uma série de RPCs para o processador mestre com a finalidade de obter as linhas de A necessárias para a operação e todas as linhas da matriz B. As linhas de A necessárias para a operação são exatamente as linhas de C que deverão ser calculadas. Após calcular todos os elementos de C sob sua responsabilidade, o processador devolve estes elementos através de outra seqüência de RPCs para o processador mestre.

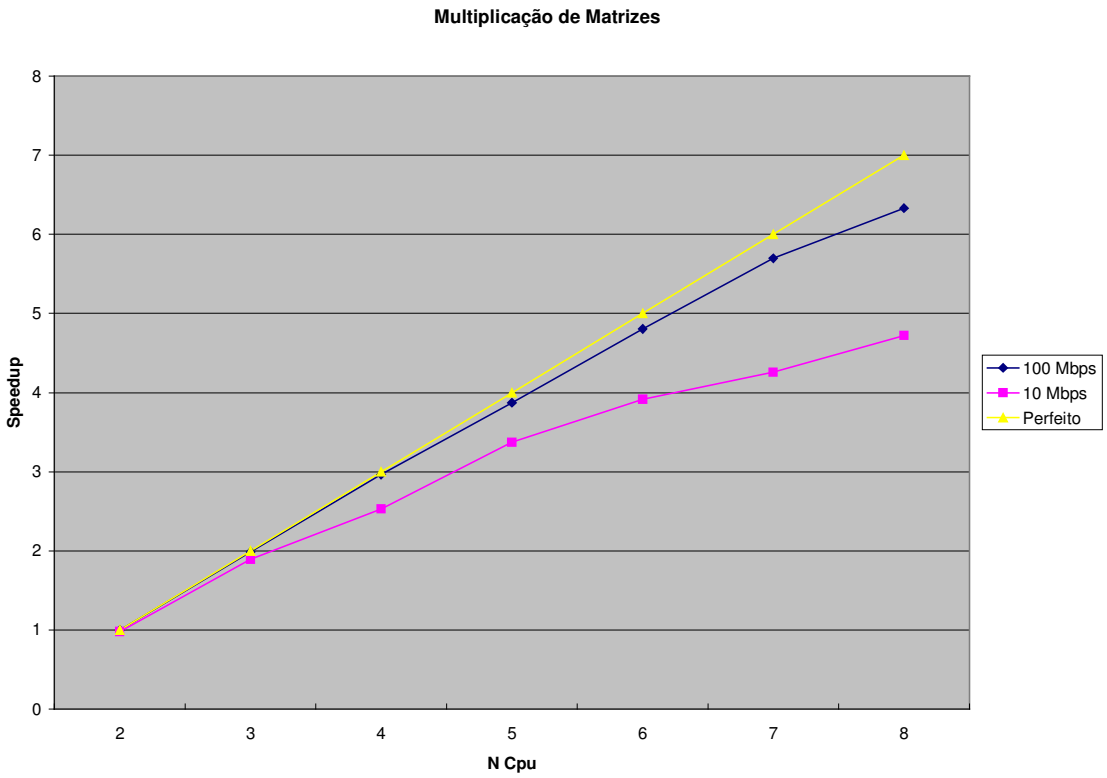

Figura 30 – Speedup multiplicação de matrizes

No gráfico presente na Figura 30 é possível verificar o *speedup* desta operação para duas matrizes quadradas de 1500 elementos nas duas configurações de rede sob análise neste trabalho. É importante notar no gráfico que o conceito de *speedup* sofreu uma pequena distorção nesta análise, pois o processador que está

coordenando a computação não foi considerado para simular o *speedup* perfeito. Nos dados apresentados se nota que a linearidade do *speedup* para a rede operando a 10 Mbps é bem menor que a 100 Mbps. Isto indica que a aplicação consegue tirar maior proveito do aumento do número de processadores com a rede mais rápida, conseguindo manter um crescimento do *speedup* mais constante. O desempenho desta aplicação foi medido utilizando o relógio interno da máquina onde o coordenador executou.

5.2. TSP

O problema do caixeiro viajante é um dos problemas mais antigos da ciência da computação. Este problema é um dos clássicos da teoria da complexidade de algoritmos e já foi provado ser um problema da classe NP-Completo (não pode ser decidido deterministicamente em tempo polinomial).

O problema pode ser colocado da seguinte maneira: dado um conjunto N de cidades e o custo de viajar de uma cidade para outra, qual o menor caminho necessário para passar por todas as cidades? É fácil perceber que uma solução ótima para este problema envolve a procura de praticamente todas as rotas possíveis, e isto pode levar a um algoritmo de complexidade N!. Assim, para um número pequeno de cidades, a quantidade de passos para achar uma solução ótima para este problema pode se tornar muito grande. Desta maneira os algoritmos que são utilizados para resolver este problema baseiam-se em soluções heurísticas.

Neste projeto utilizou-se a técnica de ramificar e limitar o problema (“branch-and-bound”). Esta técnica consiste em gerar um conjunto inicial de rotas com as n primeiras cidades. Para cada um destes ramos é feito um cálculo heurístico de um limite mínimo para o resto da rota, com a influência neste limite de cada cidade que ainda não está na rota. Em seguida cada uma destas rotas passa por um processo recursivo de busca de uma solução ótima, onde cada recursão verifica se a rota calculada até o momento mais o limite mínimo é melhor que a menor solução já encontrada. O cálculo deste limite mínimo é portanto fundamental para o algoritmo, pois através dele é possível reduzir o número de recursões realizadas para cada sub-rota.

Na implementação do algoritmo, um processador mestre gera todas as sub-rotas iniciais, colando-as numa fila de tarefas. Cada processador escravo realiza chamadas RPC para o processador mestre com a finalidade de retirar as tarefas da fila. Sempre que um processador escravo encontra uma rota completa mais curta baseado no algoritmo descrito no parágrafo anterior, ele efetua uma chamada RPC para o processador mestre informando a nova rota. Para os testes de desempenho apresentados Figura 31, foi usado um grafo simétrico de 22 cidades com as subrotas sendo geradas pelo processador mestre com 4 cidades inicialmente.

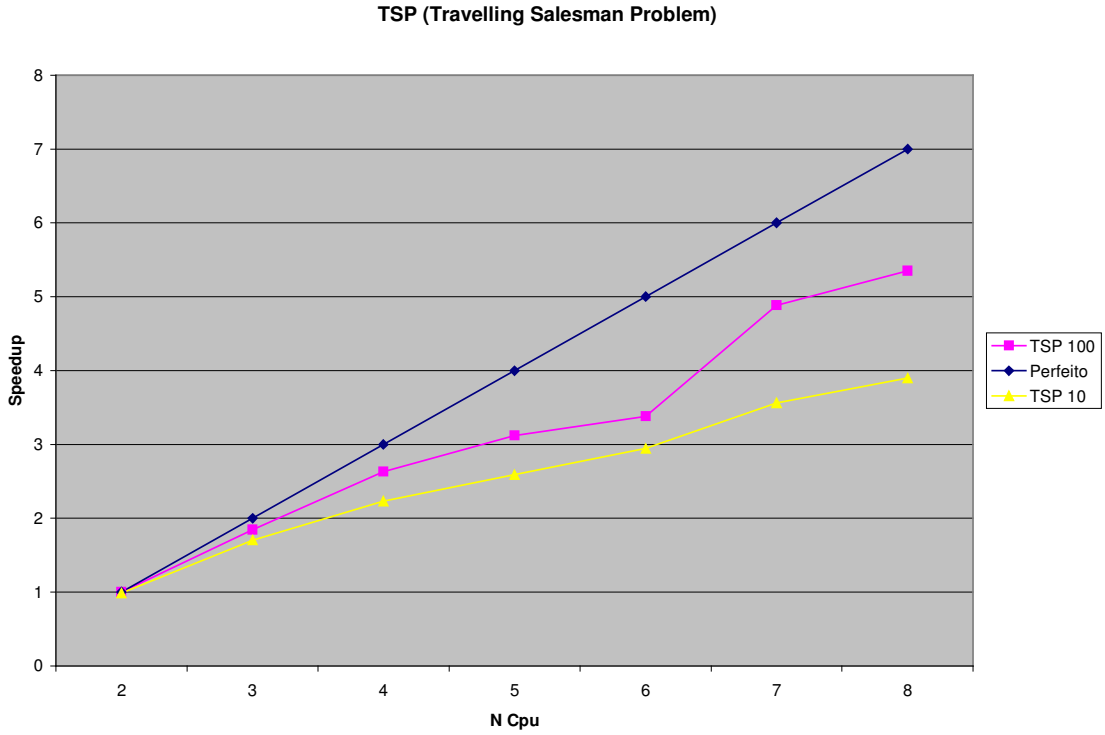

Figura 31 – Speedup TSP

Esta é uma aplicação que trabalha trafegando uma quantidade grande de mensagens pequenas. Como na multiplicação de matrizes, o aumento da velocidade da rede permite um *speedup* real mais próximo do *speedup* perfeito com o aumento do número de processadores. A sensação que se tem ao observar o gráfico é que com 8 processadores, o sistema está muito perto de seu limite com a rede funcionando a

10 Mbps. A rede funcionando a 100 Mbps demonstra um fôlego maior para o acréscimo de novos processadores no sistema.

5.3. ASP

A terceira aplicação distribuída apresentada neste capítulo consiste de um algoritmo paralelo para resolver o problema de encontrar todos os menores caminhos entre dois vértices de um grafo. Neste problema o grafo *G* é mapeado para uma matriz de distâncias, com a distância do vértice *Vi* para o vértice *Vk* armazenada no elemento *Gik* da matriz de distâncias.

O algoritmo utilizado para solucionar este problema é uma versão paralela do algoritmo de Floyd. Neste algoritmo, cada processador computa parte da matriz resultante com a menor distância de cada par de vértices do grafo. A cada iteração do algoritmo todos os processadores devem receber a linha da matriz que foi calculada naquela iteração. Isto resulta em uma quantidade grande de sincronização e de troca de mensagens entre os processadores.

Inicialmente o grafo G é particionado entre todos os processadores, da mesma maneira que no algoritmo de multiplicação de matrizes indicado anteriormente. Um processador é eleito para gerar o grafo G e distribuí-lo para todos os outros processadores. Ao final do processamento este mesmo processador deverá conter a matriz de distâncias resultante.

Numa primeira versão deste algoritmo foi usado o mecanismo de RPC para realizar todas as trocas de mensagem. Quando cada processador inicia, ele realiza uma série de RPCs para o processador principal requisitando todas as linhas do grafo G. Após a distribuição de G, todos os processadores iniciam o algoritmo, e ao inicio de cada iteração do mesmo, o processador responsável pela linha daquela iteração (definida no particionamento do grafo), repassa a mesma para todos os outros processadores, que em seguida utilizam esta linha para atualizar todas as linhas sob sua responsabilidade.

Esta tarefa de distribuição da linha da iteração foi paralelizada através de uma árvore binária. O processador 1 envia a linha para os processadores 2 e 3. O processador 2 por sua vez envia a linha para os processadores 4 e 5, e assim por

diante. Estes envios são feitos através de chamadas RPC. Ao final de todas as iterações, cada processador efetua chamadas RPC para o processador mestre devolvendo a sua parte da matriz de distância calculada.

No gráfico presente na Figura 32 seguem os resultados de performance deste algoritmo para um grafo de 2000 cidades. A quantidade de comunicação deste algoritmo é muito grande (8000 bytes para cada linha transmitida), e a carga de ocupação da rede é muito grande. Isto pode ser notado principalmente para a rede funcionando a 10 Mbps, que após cinco processadores começa a ter uma inversão na curva do *speedup*. Isto significa que a capacidade da rede atingiu seu pico. O *speedup* para 100 Mbps não é perfeito porém o crescimento do *speedup* não parou até a quantidade de 8 processadores.

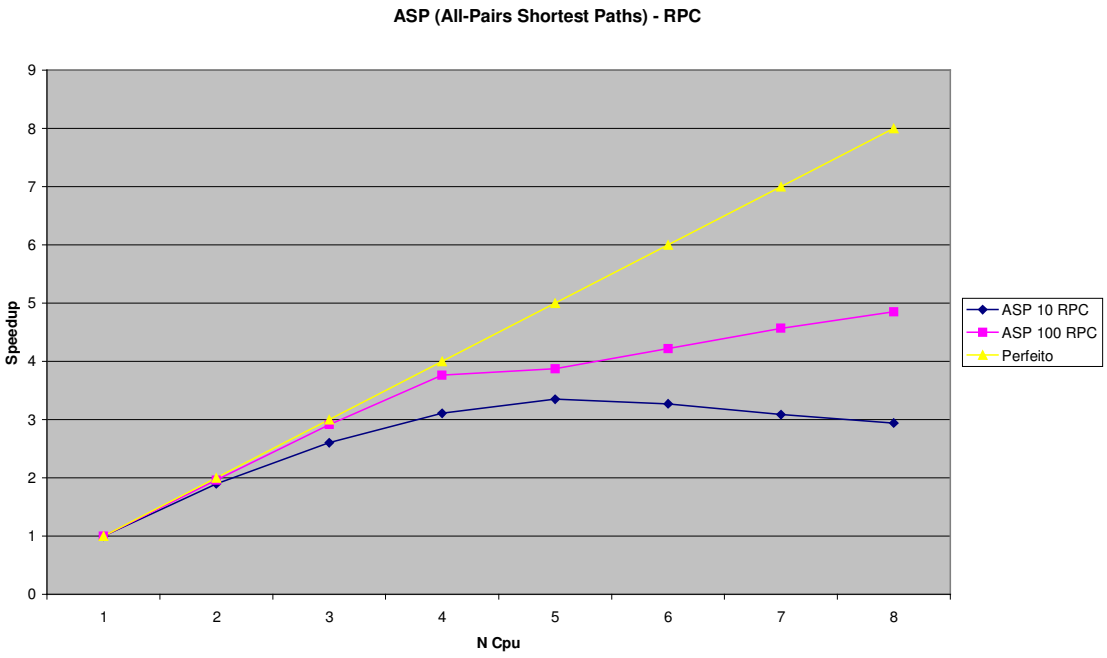

Figura 32 – Speedup ASP RPC

Para efetuar uma segunda análise deste algoritmo na rede do Amoeba, foi feita uma outra implementação do mesmo utilizando as primitivas de comunicação de grupo. Nesta nova versão, um grupo é criado com todos os processadores sendo membros do mesmo. Inicialmente o processador principal envia mensagens para o grupo com a finalidade de distribuir o grafo G.

Cumprida esta etapa todos os processadores (inclusive o principal) iniciam a computação da matriz de distância. Ao inicio de cada iteração, o processador responsável pela linha daquela iteração envia a mesma para o grupo. Desta maneira a sincronização entre os membros fica simplificada porque o protocolo de comunicação de grupo garante a ordem entre mensagens.

Na Figura 33 se encontra o gráfico com a performance do algoritmo nas duas redes em estudo neste trabalho. Nota-se que a peformance fica linear para ambas as velocidades de rede, e as diferenças que aparecem na versão RPC ficam bem mais reduzidas. Isto em parte se deve a problemas com o controle de congestão existente no protocolo FLIP para mensagens de *multicast*. Este sistema de controle de congestão foi feito para uma velocidade de rede de 10 Mbps, e não se comporta adequadamente com a alteração da velocidade da rede para 100 Mbps.

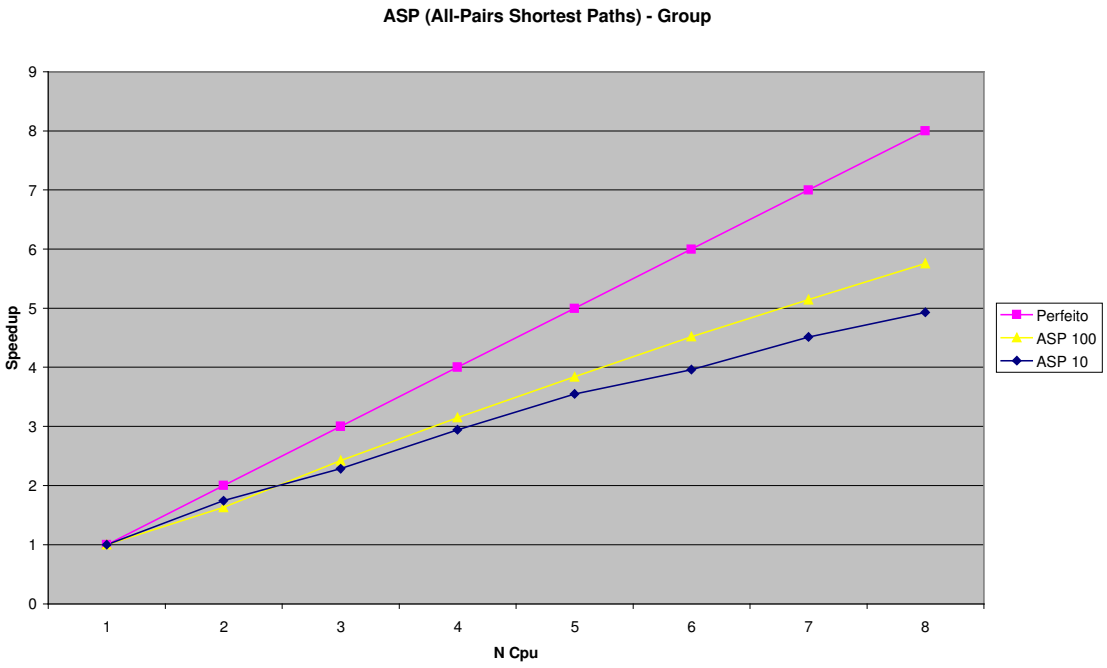

Figura 33 – Speedup ASP group

6. Considerações Finais

Neste trabalho fizemos uma análise do comportamento de um sistema distribuído ao variar a velocidade da rede de interconexão do mesmo. Ao alterar a velocidade da rede de 10 Mbps para 100 Mbps não foi obtido um ganho de performance do protocolo de rede desta mesma ordem. Isto foi comprovado principalmente com as performances obtidas com as aplicações distribuídas estudadas no capítulo 5.

Uma das explicações para este fato é a seguinte. Os mecanismos de controle de fluxo e controle de congestão implementados no protocolo FLIP, protocolo de comunicação de grupo e protocolo RPC estudados neste trabalho são dependentes da velocidade da rede. Estes mecanismos foram calibrados para uma rede ethernet de 10 Mbps e não funcionam satisfatoriamente quando a rede está operando a 100 Mbps. Fica claro, portanto, que somente o aumento da velocidade da rede não acarreta numa melhora do sistema como um todo. Para que este aumento de velocidade seja realmente repassado para as aplicações, é necessário que o protocolo de rede esteja preparado para esta nova velocidade.

Porém, apesar das observações acima, ficou evidente nos estudos efetuados com as aplicações distribuídas que o uso de uma velocidade de rede alta permite manter um ganho de performance ao acrescentar um número maior de processadores. Além disto o FLIP mostrou ser um protocolo eficiente e que em condições normais, juntamente com o protocolo de comunicação de grupo, facilita bastante o desenvolvimento de aplicações distribuídas.

6.1. Trabalhos Futuros

Ao fim deste trabalho algumas tarefas ainda ficaram incompletas. A principal delas é balancear o sistema de controle de fluxo e congestão do protocolo FLIP. O comportamento inadequado destes sistemas, principalmente nos mecanismos de *multicast*, afeta consideravelmente o protocolo de comunicação de grupo do

Amoeba. Além disto poderia ser realizado um trabalho de comparação do FLIP com o protocolo TCP/IP e com outros protocolos de funcionalidade reduzida, como o U-net e o FM (*Fast Messages*).

REFERÊNCIAS BIBLIOGRÁFICAS

AMIR, Y. et.al. Transis: a Communication Subsystem for High Availability. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING (FTCS-22), 22, Boston, Massachusetts, 1992. **Proceedings**: IEEE Computer Society Press, 1992. p.76-84.

ANDERSON, T.; CULLER, D.; PATTERSON, D. A Case for NOW (Networks of Workstations). **IEEE Micro**, fev. 1995.

BAL. H.E.; KAASHOEK, M.F.; TANENBAUM A.S. Orca: A Language for Parallel Programming of Distributed Systems. **IEEE Transactions on Software Engineering**, n.3, v.18, p.190-205, mar. 1992.

BAL, H.E. et al. Orca: a Portable User-Level Shared Object System. **Technical Report IR-408**, Vrije Universiteit, Amsterdam, jun. 1996.

BAL, H.E.; HOFMAN, R.; VERSTOEP, K. A comparison of three high speed networks for parallel cluster computing. In: INTERNATIONAL WORKSHOP ON COMMUNICATION AND ARCHITECTURE SUPPORT FOR NETWORK-BASED PARALLEL COMPUTING, 1, Springer-Verlag, Fev 1997. **Proceedings**. 1997, p.184-197.

BAL, H.E. et al. Performance of a High-Level Parallel Language on a High-Speed Network. **Journal of Parallel and Distributed Computing**, v.40, n.1, p.49-64, fev. 1997.

BHOEDJANG R.A.F.; RÜHL T; BAL H.E. Design issues for user-level network interface protocols on Myrinet. **IEEE Computer**, 31(11), p.53-60, 1998.

BIRMAN, K.P. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. **Operating Systems Review**, v. 28, p. 11-21, jan. 1994.

BIRMAN, K.P.; CLARK, T. Performance of the ISIS Distributed Computing Toolkit. **Cornell University's Computer Science Department Technical Report**, jun. 1994.

CHERITON, D.R.; SKEEN, D. Understanding the Limitations of Causally and Totally Ordered Communication Systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 14, Asheville, Carolina do Norte, dez.1993. **Proceedings**, ACM Press, 1993, p. 44-57.

CULLER, D. E. et al. Parallel computing on the berkeley NOW. In: JOINT SYMPOSIUM ON PARALLEL PROCESSING (JSPP), 9, 1997. **Proceedings**. 1997.

ESTADOS UNIDOS. Digital Equipment Corporation. **DIGITAL semiconductor 21140A PCI fast ethernet LAN controller – Hardware Reference Manual**. Maynard, Massachusetts, 1998. ON: EC-QNTNF-TE.

GIANNINI, L.A.; CHIEN, A.A. A software architecture for global address space communication on clusters: Put/Get on Fast Messages. In: HIGH-PERFORMANCE DISTRIBUTED COMPUTING CONFERENCE, 1998. **Proceedings**. 1998.

GUERRAOUI, R.; SCHIPER, A. Total order multicast to multiple groups. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS. 1997. **Proceedings**. 1997.

JALOTE, P. **Fault tolerance in distributed systems**. Englewood Cliffs, New Jersey: Prentice Hall, 1994.

KAASHOEK, M.F. et.al. An efficient reliable broadcast protocol**. Operating Systems Review**, v.23, n.4, p.5-19, out. 1989.

KAASHOEK, M.F.; TANENBAUM, A.S.; AND VERSTOEP, K. A Comparison of Two Paradigms for Distributed Computing. In: ACM SIGOPS WORKSHOP, 5, Le Mont St. Michel, France, set. 1992. **Proceedings**. 1992.

KAASHOEK, M.F.; TANENBAUM, A.S. Efficient Reliable Group Communication for Distributed System. **Dept. of Mathematics and Computer Science Technical Report,** Vrije Universiteit, Amsterdam, 1992.

KAASHOEK, M.F. et al. FLIP: an internetwork protocol for supporting distributed systems. **ACM Trans. Computer Systems**, v.11, n.1, p.77-106, fev. 1993.

KAASHOEK M.F.; TANENBAUM, A.S.; VERSTOEP, K. Using Group Communication to Implement a Fault-Tolerant Directory Service. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 13, maio 1993. **Proceedings**. 1993. p. 130-139.

KAASHOEK, M.F.; TANENBAUM, A.S. An evaluation of the Amoeba group communication system. In: IEEE INTERNATIONAL CONFERENCE OF DISTRIBUTED COMPUTING SYSTEMS, 16, maio 1996. **Proceedings**. 1996. p.436-447.

KARAMCHETI, V.; CHIEN, A.A. Software overhead in messaging layers: where does the time go? **ACM SIGPLAN Notices**, v.29, n.11, p.51-60, nov. 1994.

LANGENDOEN, K. et al. Integrating Polling, Interrupts, and Thread Management. In: FRONTIERS 96, Annapolis, MD, out. 1996. **Proceedings**. 1996. p.13-22.

LANGENDOEN, K.; HOFMAN, R.; BAL. H. Challenging Applications on Fast Networks. In: HPCA-4 HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 4, Las Vegas, Nevada, fev.1998. **Proceedings**. 1998. p.125-137.

LAURIA, M.; PAKIN, S.; CHIEN, A. Efficient layering for high speed communication: Fast Messages 2.x. In: HIGH PERF. DISTRIBUTED COMPUTING CONF. (HPDC7), 7, Chicago, Illinois, julho 1998. **Proceedings**. 1998.

LAURIA, M.; PAKIN, S.; CHIEN, A. Efficient layering for high speed communication: the MPI over Fast Messages (FM) experience. **Cluster Computing**, v.2, n.2, p.107-116, set. 1999.

MARTIN, R. P. et al. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In: ANNUAL INTERNATION SYMPOSIUM ON COMPUTER ARCHITECTURE, 24, Denver, Colorado, junho 1997. **Proceedings**. 1997. p.85-97.

MOSER, L.E. et.al. Totem: A Fault-Tolerant Multicast Group Communication System. **Communications of the ACM**, v.39, n.4, p.54-63, 1996.

NIEUWPOORT, R.V. et.al. Wide area parallel computing in JAVA. In: JAVAGRANDE'99, 1999. **Proceedings**. 1999.

OEY, M.; LANGENDOEN, K.G.; BAL, H.E. Comparing kernel-space and user-space communication protocols on Amoeba. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTER SYSTEMS, 15, Vancouver, B.C., Canada, Maio 1995. **Proceedings**. 1995.

PAKIN, S.; LAURIA, M.; CHEIN, A. High Performance Messaging on Workstations Illinois Fast Messages (FM) for Myrinet**.** In: SUPERCOMPUTING '95, San Diego, California, 1995. **Proceedings**. 1995.

PAKIN, S.; KARAMCHETI V.; CHIEN, A. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors**. IEEE Concurrency**, v.5, n.2, p.60-73, 1997.

PATTERSON, D.A.; HENESSY, J.L. **Computer architecture: a quantitative approach**. 2.ed. São Francisco, California: Morgan Kaufmann Publishers, 1996.

PLAAT, A.; BAL, H.; HOFMAN, R. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, Orlando, Florida., Jan. 1999. **Proceedings**. 1999. p.244-253.

RENESSE, R.V.; STAVEREN, H.V.; TANENBAUM, A.S. The Performance of the Amoeba Distributed Operating System**. Software - Practice and Experience,** v.19, n.3, p.223-234, mar. 1989.

RENESSE, R.V.; BIRMAN, K.P.; MAFFEIS, S. Horus: A flexible group communication system. **Communications of the ACM**, v. 39, p. 76--83, abr. 1996.

RÜHL, T. et al. Experience with a Portability Layer for Implementing Parallel Programming Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA'96), Sunnyvale, California, 1996. **Proceedings**. 1996. p.1477-1488.

SINGH, J.P.; WEBER, W.D.; GUPTA, A. SPLASH: Stanford Parallel Applications for Shared Memory. **Computer Architecture News**, v.20, n.1, p.5-44, mar. 1992.

TANENBAUM A.S. et al. Experiences with the Amoeba Distributed Operating System. **Communications of the ACM**, v.33, n.12, 1990.

TANENBAUM, A.S. et al. The Amoeba distributed operating system - a status report. **Computer Communications**, v.14, n.6, p.324-335, ago. 1991.

TANENBAUM, A.S., BAL, H. E., KAASHOEK, M.F. Programming a distributed system using shared objects. In: INTERNATIONAL SYMPOSIUM ON HIGH

PERFORMANCE DISTRIBUTED COMPUTING, 2, , Spokane, Washington, Julho 1993. **Proceedings**. 1993. p.5-12.

TANENBAUM, A.S.;WOODHULL, A.S. **Operating systems: design and implementation**. 2.ed. Upper Saddle River, New Jersey: Prentice Hall, 1997.

VITENBERG, R. et.al. Group Communication Specifications: A Comprehensive Study. **Technical report**, Dept. of Computer Science, Technion, Israel, set. 1999.

WELSH, M.; BASU A.; EICKEN, T.V. Low-latency communication over fast ethernet. In: EURO-PAR, 1996. **Proceedings**: vol. I, 1996.

WELSH, M.; BASU A.; EICKEN, T.V. ATM and fast Ethernet network interfaces for user-level communication. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 3, 1997. **Proceedings**. 1997. p.332-342.

WIKIPEDIA – THE FREE ENCYCLOPEDIA. Seção Computer Science. Disponível em: <http://www.wikipedia.com>. Acesso em: 20 de mar. 2002.

APÊNDICE A – Driver DEC21140

```
/*

   Universidade Estadual de Sao Paulo - Escola Politecnica
   PCS - Computer Engineering Departament

   Author: Amilcar Rosa Pereira - Master Science Student
   Date: 10/01/1998
   Last Modification: 05/11/1999
   File: DEC21140.C
   Contents: Amoeba Device Driver for network interfaces that use
             the Digital Semiconductor 21140A PCI Fast Ethernet LAN
             Controler. The Driver was initially designed for the
             Acer Fast Ethernet interface.

*/



#include <bool.h>
#include <amoeba.h>
#include <debug.h>
#include <machdep.h>
#include <type.h>
#include <pci.h>
#include <assert.h>

INIT_ASSERT


#include <global.h>
#include <byteorder.h>
#include <ethif.h>
#include <sys/flip/ethproto.h>
#include <sys/flip/ethpreamble.h>
#include <sys/flip/packet.h>
#include <sys/flip/flip.h>
#include <sys/proto.h>



#include "i386_proto.h"

void dec_intr();
void dec_received();
void dec_transmitted();
void build_media_table();
void select_media();
void dec_timer();

#define CSR0 0x00
```

```
#define CSR1 0x08
#define CSR2 0x10
#define CSR3 0x18
#define CSR4 0x20
#define CSR5 0x28
#define CSR6 0x30
#define CSR7 0x38
#define CSR8 0x40
#define CSR9 0x48
#define CSR10 0x50
#define CSR11 0x58
#define CSR12 0X60
#define CSR13 0x68
#define CSR14 0x70
#define CSR15 0x78

#define PCI_DEVICE_ID_DEC_21140    0X0009



#define RX_RING_SIZE 512
#define TX_RING_SIZE 512

#define RCVPOOLSIZE 2000
#define RCVETHPKTSIZE (PKTBEGHDR + 1514)

#define MAXINTR 4


static char *medianame[] =
                   {"TP (10 Mb/s)","BNC (10 Mb/s)","",
                  "SYM_SCR (100BASE-TX)","TP full-duplex",
                  "SYM_SCR full-duplex (100BASE-TX)",
                  "100BASE-T4","100BASE-FX(fiber)",
                  "100BASE-FXFD (fiber full-duplex)" };

/*
 * dec21140 hardware and driver statistics
 */
typedef struct decstat {
    int         decs_carlost;      /* carrier sense lost */
    int         decs_collisions;   /* packets collided at least once
*/
    int         decs_crc;          /* input CRC errors */
    int         decs_deferred;     /* deferred packets */
    int         decs_fifo;         /* FIFO underrun */
    int         decs_frame;        /* input framing errors */
    int         decs_heartbeat;    /* heart beat failure */
    int         decs_lcol;         /* late collisions */
    int         decs_lost;         /* packets lost */
    int         decs_ovw;          /* over writes */
    int         decs_read;         /* packets read */
    int          decs_written;          /* packets written */
    int          decs_xcollisions; /* aborts due to excessive
collisions*/
    int          decs_frozen;      /* frozen transmitter */
    int          decs_align;       /* frame alignment error */
    int          decs_wraps;       /* wrap around copies */
    int          decs_maxintr;     /* max interrupts handled */
```

```
    int           decs_maxrecv;           /* max packets received at
once */
    int           decs_maxtrans;          /* max packets transmited
at once */
} decstat_t, *decstat_p;


typedef struct pkt_arrived_on_interface {
      pkt_p pkt;
      int ifno;
} pktoi_t;


typedef struct tx_descriptor {
      int32 status;
      uint32 count;
      uint32 buffer1;
      uint32 buffer2;
} tx_desc;

typedef struct rx_descriptor {
      int32 status;
      int32 count;
      uint32 buffer1;
      uint32 buffer2;
} rx_desc;



#define DEFAULT_MEDIA   0x03  /* 100BASETX - Twisted Pair */



typedef struct infoblock {
      char type;
      uint8 media;
      uint8 csr12;
      uint16 command;
} block_t, *block_p;


typedef struct mediatable {
      uint16 default_media;
      uint8 csr12dir;
      uint8 count;
      block_p blocks;
} mediatable_t, *mediatable_p;



typedef struct dev_dec {
      uint32 setup_frame[48];
      tx_desc dec_tx_ring[TX_RING_SIZE];
      rx_desc dec_rx_ring[RX_RING_SIZE];
      pkt_p dec_tx_pkts[TX_RING_SIZE];
      pkt_p dec_rx_pkts[RX_RING_SIZE];
      pool_t dec_rpool;
```

```
      unsigned int dirty_tx, dirty_rx;
      unsigned int cur_tx, cur_rx;
      int media_sense;
      int polarity;
      uint8 bus;
      uint8 device_fn;
      uint8 irq;
      uint32 base_io;
      uint32 csr6;
      int interrupt;
      int tbusy;
      int softifno;
      char eth_addr[6];
      mediatable_t mtable;
      unsigned char eeprom[128];
      decstat_t dec_stat;
} dec_t, *dec_p;




static dec_p *dec_data;


typedef struct dec_linked_list {
      dec_t dev_dec;
      struct dec_linked_list *next;
} decll_t;

static decll_t dec_list_root;

static int *ndec;

static int rcvenq = 0;

static int txenq = 0;

#define EE_READ         0x4801          /*CSR9 SERIAL ROM READ
SELECT*/

#define EE_CS                 0X01
#define EE_CLOCK        0X02
#define EE_DI                 0X04
#define EE_DO                 0X08

#define EE_READ_CMD           0x180

void ee_udelay(int usec)
{
    int i = 0;
    int b,c;
    for (i = 0; i < usec; i++)
       b=b+c;
}

int read_eeprom(int ioaddr, int location)
{
```

```
    int i;
    short value;
    unsigned short retval = 0;
    int ee_addr = ioaddr+CSR9;
    int cmd_addr = EE_READ_CMD | location;

    out_long(ee_addr, EE_READ & ~EE_CS);
    out_long(ee_addr, EE_READ);

    for (i = 10; i >= 0; i--) {
      value = (cmd_addr & (1 << i)) ? EE_DI : 0;
        out_long(ee_addr, EE_READ | value);
        ee_udelay(1000);
        out_long(ee_addr, EE_READ | value | EE_CLOCK);
        ee_udelay(1500);
      out_long(ee_addr, EE_READ | value);
      ee_udelay(2500);
    }

    out_long(ee_addr, EE_READ);

    for( i = 16; i > 0; i--) {
      out_long(ee_addr, EE_READ | EE_CLOCK);
        ee_udelay(1500);
        retval = (retval << 1) | ((in_long(ee_addr) & EE_DO) ? 1 :
0);
        out_long(ee_addr, EE_READ);
        ee_udelay(2500);
    }
    out_long(ee_addr, EE_READ & ~EE_CS);
    return(retval);
}


void dec_init_ring(hardifno)
int hardifno;
{
    int i;
    dec_p decd;
    char *data;
    pkt_p bufs;
    pkt_p pkt;
    decd=dec_data[hardifno];

    data = aalloc((vir_bytes) RCVPOOLSIZE * RCVETHPKTSIZE, 0);
      if (data == (char *) 0) {
            printf("dec21140.c: rcvpool allocation failure\n");
      }
    bufs= (pkt_p) aalloc((vir_bytes) RCVPOOLSIZE * sizeof(pkt_t),
0);
    pkt_init(&decd->dec_rpool, RCVETHPKTSIZE, bufs, RCVPOOLSIZE,
data,
            (void (*)()) 0, 0L);


    decd->dirty_tx = decd->cur_tx = 0;
    decd->dirty_rx = decd->cur_rx = 0;
    for (i = 0; i < RX_RING_SIZE; i++) {
```

```
            PKT_GET(pkt, &decd->dec_rpool);
            pkt->p_admin.pa_size = RCVETHPKTSIZE;
            proto_setup_input(pkt, eh_t);

            decd->dec_rx_pkts[i] = pkt;

        decd->dec_rx_ring[i].status = 0x80000000;
            decd->dec_rx_ring[i].count = RCVETHPKTSIZE;
            decd->dec_rx_ring[i].buffer1 = (uint32) pkt_offset(pkt);
            decd->dec_rx_ring[i].buffer2 = 0;
    }
    decd->dec_rx_ring[i-1].count = RCVETHPKTSIZE | 0x02000000;
    for (i = 0; i < TX_RING_SIZE; i++) {
        decd->dec_tx_ring[i].status = 0;
      decd->dec_tx_ring[i].count = 0;
      decd->dec_tx_pkts[i] = 0;
    }

}


int dec_init(hardifno, softifno, ifaddr, nrcvpkt, nsndpkt)
int hardifno,softifno;
char *ifaddr;
int *nrcvpkt,*nsndpkt;
{
    dec_p decd;
    uint32 base_io;
    uint8 irq;
    unsigned char ee_data[128];
    int aux;
    unsigned char pci_latency, pci_command, pci_cache_size;
    uint32 *setup_frame;



    *nrcvpkt = RX_RING_SIZE;
    *nsndpkt = TX_RING_SIZE;
    decd=dec_data[hardifno];
    decd->softifno = softifno;
    pcibios_read_config_dword(decd->bus, decd->device_fn,
                              PCI_BASE_ADDRESS_0,
                             &base_io);
    base_io &= ~3;
    pcibios_read_config_byte(decd->bus, decd->device_fn,
                             PCI_INTERRUPT_LINE,
                            &irq);
    pcibios_read_config_byte(decd->bus, decd->device_fn,
                             PCI_LATENCY_TIMER,
                            &pci_latency);
    pcibios_read_config_byte(decd->bus, decd->device_fn,
                             PCI_CACHE_LINE_SIZE,
                            &pci_cache_size);
    pcibios_read_config_byte(decd->bus, decd->device_fn,
                             PCI_COMMAND,
                            &pci_command);

    decd->base_io=base_io;
    decd->irq=irq;
```

```
    out_long(base_io + CSR6, 0x00040000);
    /* Resetar o chip mantendo o bit 0 setado por 50 ciclos PCI */
    out_long(base_io+CSR0, 0x00000001);
    pit_delay(10);

    out_long(base_io + CSR0, 0x00204800);

    /* Paralizar o processo de transmissao e recepcao de quadros */

    out_long(base_io + CSR6, in_long(base_io + CSR6) & ~0x2002);
    decd->csr6 = 0;
    setirq(decd->irq, dec_intr);
    pic_enable(decd->irq);
    for (aux = 0; aux < 128/2; aux++) {
        ((short *)ee_data)[aux] = read_eeprom(base_io, aux);
    }

    for (aux = 0; aux < 6; aux++){
        ifaddr[aux] = decd->eth_addr[aux] = ee_data[20+aux];
    }
    build_media_table(hardifno,base_io);

    /* Adjust the General Purpose Port Direction Pins */
    out_long(base_io + CSR12, (uint32) (0x100 | decd-
>mtable.csr12dir));

    dec_init_ring(hardifno);
    select_media(hardifno);
    setup_frame = decd->setup_frame;



    *setup_frame++ = 0xffff;       /* Accept Broadcast Addresses */
    *setup_frame++ = 0xffff;
    *setup_frame++ = 0xffff;
    for (aux = 1; aux < 16; aux++) {
        *setup_frame++ = ((uint16 *)decd->eth_addr)[0];
        *setup_frame++ = ((uint16 *)decd->eth_addr)[1];
        *setup_frame++ = ((uint16 *)decd->eth_addr)[2];
    }
    decd->dec_tx_ring[0].buffer1 = (uint32) decd->setup_frame;
    decd->dec_tx_ring[0].count = 192 | 0x08000000;
    decd->dec_tx_ring[0].status = 0x80000000;
    decd->cur_tx++;


    decd->interrupt = 0;
    decd->tbusy = 0;
    out_long(base_io + CSR3, (uint32) decd->dec_rx_ring);
    out_long(base_io + CSR4, (uint32) decd->dec_tx_ring);
    out_long(base_io + CSR7, (uint32) 0x0001fbff);
    out_long(base_io + CSR6, decd->csr6);
    out_long(base_io + CSR6, decd->csr6 | 0x2000);
    out_long(base_io + CSR6, decd->csr6 | 0x2002);
    out_long(base_io + CSR2, 0);
    out_long(base_io + CSR11, 0x17530);  /* 30000 x 81.92us = 2.5
seg. */
    return(1);
}
```

```
void select_media(ifno)
int ifno;
{
    dec_p decd;
    int i;
    uint32 base_io;
    uint32 newcsr6 = 0;

    decd = dec_data[ifno];
    base_io = decd->base_io;
    for (i = 0; i < decd->mtable.count; i++) {
        if (decd->mtable.blocks[i].media == DEFAULT_MEDIA) {
          newcsr6 = (decd->mtable.blocks[i].command & 0x71) << 18;
          newcsr6 |= 0x02280000;
          decd->media_sense = (decd->mtable.blocks[i].command &
0x0e) >> 1;
          decd->polarity = (decd->mtable.blocks[i].command & 0x80) ?
1 : 0;
          out_long(base_io + CSR12, decd->mtable.blocks[i].csr12);
            }
    }
    decd->csr6 |= newcsr6;

}

void build_media_table(ifno,base_io)
int ifno;
uint32 base_io;
{
    dec_p decd;
    unsigned char ee_data[128];
    int i;
    mediatable_p mtable;
    block_p blocks;
    unsigned char *p;

    decd=dec_data[ifno];
    mtable=&decd->mtable;

    for (i = 0; i < 128/2; i++) {
        ((short *)ee_data)[i] = read_eeprom(base_io, i);
    }
    p = &ee_data[(uint16)ee_data[27]];
    mtable->default_media = *p | *(p+1) << 8;
    p+=2;
    mtable->csr12dir = *p++;
    mtable->count = *p++;
    blocks = (block_p) aalloc((vir_bytes) mtable->count *
sizeof(block_t), 0);
    for(i = 0; i < mtable->count; i++) {
        blocks[i].type = *p & 0x80;
        blocks[i].media = *p++;
        blocks[i].csr12 = *p++;
        blocks[i].command = *p | *(p+1) << 8;
        p+=2;
    }
```

```
    mtable->blocks=blocks;
}



void dec_intr(irq)
int irq;
{
    dec_p decd;
    int csr5, base_io, csr12;
    int i, dirty_tx, entry, int_count = 0;
    pkt_p pkt;
    int nreceived = 0;

    for (i = 0; i < *ndec; i++) {
      decd = dec_data[i];
      if (decd->irq == irq)
          break;
    }
    decd->interrupt = 1;
    base_io = decd->base_io;
    csr12 = in_long(base_io + CSR12);
    do {
          if (int_count > MAXINTR) {
                  printf("int_count: %d\n", int_count);
                  break;
         }
         csr5 = in_long(base_io + CSR5);
         out_long(base_io + CSR5, csr5);

           if (!(csr5 & 0x18000))
               break;

           int_count++;
           if (int_count > decd->dec_stat.decs_maxintr)
               decd->dec_stat.decs_maxintr = int_count;

           if (csr5 & 0x0040) { /* A frame is waiting for reception*/
                   if(!rcvenq)
                   enqueue(dec_received, (long)i);
                   rcvenq++;
           }

           if (csr5 & 0x0007) { /* Frames transmited, clear buffers*/
                   if (!txenq)
                   enqueue(dec_transmitted, (long)i);
                   txenq++;
           }
           if (csr5 & 0x0800) { /* Timer expired */
                   enqueue(dec_timer, (long)i);
           }

    } while(1);
    decd->interrupt = 0;
}


void dec_timer(ifno)
int ifno;
```

```
{
    dec_p decd;
    int base_io;
    uint32 csr6, csr12;

    decd = dec_data[ifno];
    base_io = decd->base_io;
    csr12 = in_long(base_io + CSR12);
    if ( ((csr12 & (1 << decd->media_sense))!=0) == decd->polarity)
{
            printf("Media Inactive, CSR12 %X \n", csr12);
    }
}


void dec_transmitted(ifno)
int ifno;
{
    dec_p decd;
    int entry;
    int dirty_tx;
    uint32 base_io;
    pkt_p pkt;
    decd=dec_data[ifno];

    entry = decd->dirty_tx % TX_RING_SIZE;
    dirty_tx = decd->dirty_tx;
    base_io = decd->base_io;
    if ((decd->cur_tx - decd->dirty_tx) >=
              decd->dec_stat.decs_maxtrans) {
            decd->dec_stat.decs_maxtrans =
                  decd->cur_tx - decd->dirty_tx;
      }
    while (decd->cur_tx > dirty_tx++) {
        if (decd->dec_tx_ring[entry].status < 0) {
            break;
        }
            if ((decd->dec_tx_ring[entry].status & 0x8000) &&
                (!(decd->dec_tx_ring[entry].count & 0x08000000))) {

                decd->dec_tx_ring[entry].status);

                if (decd->dec_tx_ring[entry].status & 0x0800)
                    decd->dec_stat.decs_carlost++;
                if (decd->dec_tx_ring[entry].status & 0x0200)
                  decd->dec_stat.decs_lcol++;
                if (decd->dec_tx_ring[entry].status & 0x0100)
                  decd->dec_stat.decs_xcollisions++;
                if (decd->dec_tx_ring[entry].status & 0x0080)
                  decd->dec_stat.decs_heartbeat++;
                if (decd->dec_tx_ring[entry].status & 0x0002)
                  decd->dec_stat.decs_fifo++;
            }
            else {

                if (decd->dec_tx_ring[entry].status & 0x0001)
                  decd->dec_stat.decs_deferred++;
                if (decd->dec_tx_ring[entry].status & 0x0078)
                  decd->dec_stat.decs_collisions +=
```

```
                  (decd->dec_tx_ring[entry].status & 0x0078) >> 3;
                decd->dec_stat.decs_written++;
                decd->dec_stat.decs_lost +=
                  in_long(base_io + CSR8) & 0xffff;
            }

                                /* Not Setup Frame */
            if (!(decd->dec_tx_ring[entry].count & 0x08000000)) {
                pkt=decd->dec_tx_pkts[entry];
                pkt_discard(pkt);
                decd->dec_tx_pkts[entry]=0;
            }
            decd->dirty_tx=dirty_tx;
            entry = dirty_tx % TX_RING_SIZE;
    }
    txenq=0;
}




void dec_received(ifno)
int ifno;
{
    uint32 entry;
    dec_p decd;
    int32 status;
    int length;
    int aux;
    pkt_p pkt, newpkt;
    pktoi_t *pktoi;
    int rx_count = 0;


    decd = dec_data[ifno];
    entry = decd->cur_rx % RX_RING_SIZE;

    /* Next entry belongs to the driver */
    while (decd->dec_rx_ring[entry].status >= 0) {
        decd->cur_rx++;
            status= decd->dec_rx_ring[entry].status;

            length = (status >> 16) - 4;

            assert(length <= 1514 && length >= 60);
        if (length < 60 || length > 1514)
            PKT_GET(newpkt, &decd->dec_rpool);

            if (newpkt == (pkt_p)0) {
            decd->dec_rx_ring[entry].status = 0x80000000;
            entry = decd->cur_rx % RX_RING_SIZE;
            continue;
            }

            if (status & 0x809C || !(status & 0x320)) {
            decd->dec_rx_ring[entry].status = 0x80000000;
            entry = decd->cur_rx % RX_RING_SIZE;
            continue;
```

```
            }
            newpkt->p_admin.pa_size = RCVETHPKTSIZE;
            proto_setup_input(newpkt, eh_t);
            pkt = decd->dec_rx_pkts[entry];
            decd->dec_rx_pkts[entry] = newpkt;
        decd->dec_rx_ring[entry].buffer1 = (long) pkt_offset(newpkt)
;
            decd->dec_rx_ring[entry].buffer2 = 0;
            decd->dec_rx_ring[entry].status = 0x80000000;
            entry = decd->cur_rx % RX_RING_SIZE;
            pkt->p_contents.pc_totsize =
          pkt->p_contents.pc_dirsize = length;

        eth_arrived(decd->softifno, pkt);

            decd->dec_stat.decs_read++;
            if (decd->dec_stat.decs_maxrecv < ++rx_count)
                decd->dec_stat.decs_maxrecv = rx_count;
    }
    rcvenq=0;
}



void dec_stop(ifno)
int ifno;
{
    dec_p decd;
    uint32 base_io;

    decd = dec_data[ifno];
    base_io = decd->base_io;

    pic_disable(decd->irq);
    /* Disable interrupts by clearing the interrupt mask. */
    out_long(base_io + CSR7, 0x00000000);
    /* Stop the chip's Tx and Rx processes. */
    out_long(base_io + CSR6, in_long(base_io + CSR6) & ~0x2002);

    printf("dec21140.c: dec_stop\n");
}


void dec_alloc(nif)
int nif;
{
    int aux;
    uint16 index;
    uint16 vendor_id,device_id;
    uint8 bus,device_fn;
    int result = 0;

    decll_t *dec_cell;
    extern hei_t heilist[];
    hei_p hp;


    for (hp=heilist;hp->hei_name;hp++) {
      if (!strcmp(hp->hei_name,"dec21140")) {
```

```
          ndec=&(hp->hei_nif);
          *ndec=0;
      }
    }
    dec_cell=&dec_list_root;
    for (index=0; index < 0xFF; index++) {
            if (!pcibios_find_class(PCI_CLASS_NETWORK_ETHERNET<<8,
index,
                                &bus, &device_fn)) {


                  pcibios_read_config_word(bus, device_fn,
PCI_VENDOR_ID,
                                          &vendor_id);
                  pcibios_read_config_word(bus, device_fn,
PCI_DEVICE_ID,
                                                  &device_id);

                if (device_id != PCI_DEVICE_ID_DEC_21140)
                    continue;
              if (vendor_id != PCI_VENDOR_ID_DEC)
                    continue;

                if (*ndec) {
                    dec_cell->next=(decll_t *)aalloc((vir_bytes)
sizeof(decll_t),0);
                    dec_cell=dec_cell->next;
                }
                ++*ndec;
                dec_cell->dev_dec.bus=bus;
                dec_cell->dev_dec.device_fn=device_fn;
            }
    }
    dec_data = (dec_p *) aalloc((vir_bytes) *ndec * sizeof(dec_p),
0);
    dec_cell = &dec_list_root;
    for (aux=0; aux < *ndec; aux++) {
      dec_data[aux]=&(dec_cell->dev_dec);
      dec_cell=dec_cell->next;
    }
    printf("ndec = %d\n", *ndec);
}


void dec_send(ifno, pkt)
int ifno;
register pkt_p pkt;
{
    dec_p decd;
    uint32 base_io;
    int i;
    uint32 csr5;
    int entry,length;
    decd = dec_data[ifno];
    base_io = decd->base_io;
    length = pkt->p_contents.pc_totsize;
    assert(length >= 60 && length <= 1514);
    decd->tbusy = 1;
    if (decd->cur_tx - decd->dirty_tx < TX_RING_SIZE) {
```

```
        entry=decd->cur_tx % TX_RING_SIZE;
        decd->dec_tx_pkts[entry]=pkt;
        decd->dec_tx_ring[entry].buffer1 = (long) pkt_offset(pkt);
        decd->dec_tx_ring[entry].count = pkt->p_contents.pc_dirsize;
        length -= pkt->p_contents.pc_dirsize;
        if (length > 0) {
            decd->dec_tx_ring[entry].buffer2 =
            (long) pkt->p_contents.pc_virtual;
            decd->dec_tx_ring[entry].count |= (length << 11);
        }

        if (decd->cur_tx - decd->dirty_tx > (TX_RING_SIZE/2))
            decd->dec_tx_ring[entry].count |= 0xE0800000;
            else
            decd->dec_tx_ring[entry].count |= 0xE0800000;
        if (entry == TX_RING_SIZE - 1)
            decd->dec_tx_ring[entry].count |= 0x02000000;

        decd->cur_tx++;
        decd->dec_tx_ring[entry].status = 0x80000000;  /* chip
ownership */
        /* Trigger a transmit demmand command */
        out_long(base_io + CSR1, 0);
    }
    else {
        pkt_discard(pkt);
      csr5 = in_long(base_io + CSR5);
      out_long(base_io + CSR6, in_long(base_io + CSR6) & ~0x2000);
      out_long(base_io + CSR6, in_long(base_io + CSR6) | 0x2002);
    }
    decd->tbusy = 0;
}


int dec_setmc(ifno, list)
int ifno;
ethmcast_p list;
{
#ifndef NOMULTICAST
    dec_p decd;
      uint32 *setup_frame;
      uint32 base_io, csr6, flag;
      ethmcast_p em;
      int i, entry, nmcast = 0;

      decd = dec_data[ifno];
      base_io = decd->base_io;
      setup_frame = decd->setup_frame;

      csr6 = in_long(base_io + CSR6);
      for(em = list; em != (ethmcast_p)0; em= em->em_next,
nmcast++);

      if (nmcast < 15) {                  /* Perfect Address
Filtering Mode */
            for(em = list, i = 0; i < nmcast; i++ , em = em-
>em_next) {
                  *setup_frame++ = ((uint16 *)em->em_addr)[0];
```

```
                        *setup_frame++ = ((uint16 *)em->em_addr)[1];
                        *setup_frame++ = ((uint16 *)em->em_addr)[2];
                }
                *setup_frame++ = 0xffff;
                *setup_frame++ = 0xffff;
                *setup_frame++ = 0xffff;
                do {
                        *setup_frame++ = ((uint16 *)decd->eth_addr)[0];
                        *setup_frame++ = ((uint16 *)decd->eth_addr)[1];
                        *setup_frame++ = ((uint16 *)decd->eth_addr)[2];
                } while(++i < 15);
                flag = 0x08000000;
                csr6 &= ~1;
                decd->csr6 &= ~1;
        }
        else {            /* Imperfect Address Filtering Mode */
        }

        disable();

        entry = decd->cur_tx++ % TX_RING_SIZE;
        if (entry != 0) {       /* Not Beginning of tx_ring */
                                /* Adding a null entry */
                decd->dec_tx_ring[entry].status = 0x80000000;
                decd->dec_tx_ring[entry].count = flag;
                entry = decd->cur_tx++ % TX_RING_SIZE;
                if (entry == (TX_RING_SIZE - 1))
                        decd->dec_tx_ring[entry].count |= 0x02000000;
        }
        decd->dec_tx_ring[entry].status = 0x80000000;
        decd->dec_tx_ring[entry].count = flag | 192;
        decd->dec_tx_ring[entry].buffer1 = (uint32) decd->setup_frame;
        if (entry == (TX_RING_SIZE - 1))
                decd->dec_tx_ring[entry].count |= 0x02000000;
        out_long(base_io + CSR6, csr6);
        out_long(base_io + CSR1, 0);
        enable();

#endif
}

#ifdef STATISTICS

/*
 * Dump dp8390 statistic information for every adapter
 */
int
dec_netdump(begin, end)
    char *begin, *end;
{
    register char *p;
    int i;

    p = bprintf(begin, end, "======== Dec21140 statistics
=======\n");
    for (i = 0; i < *ndec; i++) {
      dec_p decd = dec_data[i];
      decstat_p decsp = (decstat_p) &decd->dec_stat;
      int n;
```

```
      p = bprintf(p, end, "dec #%d:\n", i);
      p = bprintf(p, end, "read    %7ld ",   decsp->decs_read);
      p = bprintf(p, end, "written %7ld ",   decsp->decs_written);
      p = bprintf(p, end, "frame   %7ld ",   decsp->decs_frame);
      p = bprintf(p, end, "crc     %7ld ",   decsp->decs_crc);
      p = bprintf(p, end, "lost    %7ld\n",  decsp->decs_lost);
      p = bprintf(p, end, "deff    %7ld ",   decsp->decs_deferred);
      p = bprintf(p, end, "coll    %7ld ",   decsp-
>decs_collisions);
      p = bprintf(p, end, "xcoll   %7ld ",   decsp-
>decs_xcollisions);
      p = bprintf(p, end, "carlost %7ld ",   decsp->decs_carlost);
      p = bprintf(p, end, "fifo    %7ld\n",  decsp->decs_fifo);
      p = bprintf(p, end, "heart   %7ld ",   decsp->decs_heartbeat);
      p = bprintf(p, end, "lcoll   %7ld ",   decsp->decs_lcol);
      p = bprintf(p, end, "ovw     %7ld ",   decsp->decs_ovw);
      p = bprintf(p, end, "frozen  %7ld ",   decsp->decs_frozen);
      p = bprintf(p, end, "align   %7ld\n",  decsp->decs_align);
      n = decd->cur_tx - decd->dirty_tx;
      p = bprintf(p, end, "tqueued %7ld ", n);
      p = bprintf(p, end, "maxintr %7ld ", decsp->decs_maxintr);
      p = bprintf(p, end, "maxtrans%7ld ", decsp->decs_maxtrans);
      p = bprintf(p, end, "maxrecv %7ld\n\n", decsp->decs_maxrecv);

    }
    return p - begin;
}
```