UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Plataforma de comunicação tempo real sobre clusters SCI

por

TALES HEIMFARTH

Dissertação submetida a avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Prof. Dr. Franz J. Rammig
Co-Orientador

Porto Alegre, dezembro de 2002.

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Heimfarth, Tales

Plataforma de comunicação tempo real sobre clusters SCI / por Tales Heimfarth. — Porto Alegre: PPGC da UFRGS, 2002.

150 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2002. Orientador: Wagner, Flávio Rech; Co-orientador: Rammig, Franz J.

1. Comunicação de tempo real. 2. Sistemas distribuídos. 3. Clusters SCI. I. Wagner, Flávio Rech. II. Rammig, Franz J. III. Título.

Agradecimentos

À minha esposa, Carolina A. Heimfarth, por me incentivar e apoiar nos diversos momentos e acreditar nas minhas potencialidades. Aos meus pais, Celso D. Heimfarth e Carmen T. L. Heimfarth, que sempre apostaram em mim e vibraram com cada conquista alcançada.

Meu muito obrigado ao meu orientador Prof. Flávio Rech Wagner, que com sua sabedoria e imprencindível estímulo, desenpenhou papel fundamental na elaboração desse trabalho. Agradeço também por ter possibilitado minha ida para a universidade de Paderborn e, desta forma, auferir experiências enriquecedoras na área acadêmica.

Também agradeço de forma especial ao meu co-orientador, Prof. Franz J. Rammig, que me deu a oportunidade de trabalhar em seu grupo de pesquisa, sempre me apoiando no desenvolvimento desse projeto.

Aos colegas Prasad Chrishua, Marcelo Götz e Carsten Böke, pelas inúmeras sugestões e discussões de assuntos relacionado ao meu trabalho.

Outrossim, agradeço ao grupo de pesquisa do professor Franz Rammig pela oportunidade de trabalhar em um ambiente onde a palavra chave é cooperação.

Sumário

Lista de Abreviaturas

ABR Available Bit Rate

API Application–Program Interface

ARP Address Request Protocol

ATM Asynchronous Transfer Mode

CBR Constant Bit Rate

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSMA/CD Carrier Sense Multiple Access / Collision Detection

DMA Direct Memory Access

DSM Distributed Shared Memory

E/S Entrada/Saída

EVENTS Computer Vision Enables Non-Constrained Transmission of Scenario

EWQ Echo Waiting Queue

FIFO First In First Out

GNU GNU is Not Unix

HIQ Host Input Queue

ID Identificador

INET Internet

IP Internet Protocol

LIQ Link Input Queue

MAC Media Access Control

MIP Moving Images Prototype

MMU Memory Management Unit

MPI Message Passing Interface

NRT Not Real-Time

OSI Open Systems Interconnection

PCI Peripheral Component Interconnect

PC Personal Computer

RTAI Real-Time Application Interface

RTC Real-Time Communication

RT Real-time

SCI/RT Real-Time Scalable Coherent Interface

SCI	Scalable Coherent Interface
SIP	Still Image Prototype
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TTRT	Target Token Rotation Time
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
VBR	Variable Bit Rate
VPM	Virtual Parallel Machine

Lista de Figuras

Lista de Tabelas

Resumo

Devido a sua baixa latência e alta largura de banda, os clusters equipados com o adaptador *SCI* são uma interessante alternativa para sistemas de tempo real distribuídos.

Esse trabalho apresenta o projeto e implementação de uma plataforma de comunicação de tempo real sobre *clusters SCI*. O hardware padrão do *SCI* não se mostra adequado para a transmissão de tráfego de tempo real devido ao problema da contenção de acesso ao meio que causa inversão de prioridade. Por isso uma disciplina de acesso ao meio é implementada como parte da plataforma.

Através da arquitetura implementada é possível o estabelecimento de canais de comunicação com garantia de banda. Assim, aplicações multimídias, por exemplo, podem trocar dados com taxa constante de comunicação.

Cada mensagem enviada é recebida somente uma vez. Assim, mensagens com a semântica de eventos podem ser enviadas. Além disso, a ordem e o tamanho das mensagens são garantidos.

Além do tráfego com largura de banda garantida, a plataforma possibilita a troca de pacotes *IP* entre diferentes máquinas do *cluster*. Esses pacotes são inseridos no campo de dados dos pacotes próprios da plataforma e após são enviados através de um canal especial. Assim, é possível aplicações trocarem mensagens através do uso de pacotes *IP*. Além disso, essa funcionalidade da plataforma permite também a execução de bibliotecas de comunicação baseadas em *TCP/IP* como o *MPI* sobre o *cluster SCI*.

A plataforma de comunicação é implementada como módulos do sistema operacional Linux com a extensão de tempo real RTAI.

A avaliação da plataforma mostrou que mesmo em cenários com muita comunicação entre todos os nodos correndo, a largura de banda reservada para cada canal foi mantida.

Palavras-chave: Comunicação de tempo real, Sistemas distribuídos, Clusters SCI .

TITLE: "A REAL-TIME COMMUNICATION PLATFORM BASED ON AN *SCI* CLUSTER"

Abstract

Due to its low latency and high bandwidth, the SCI-network is an interesting alternative for distributed real-time systems.

The project and the implementation of a real-time communication platform over a *SCI* Cluster is here presented. The original *SCI* hardware is not suitable to deal with real-time communications due to the priority inversion problem. A deterministic MAC is part of the platform to avoid the contention problem that causes this priority inversion.

With this platform is possible the establishment of real-time channels with reserved bandwidth. Then, multimedia applications, for example, can exchange data with a constant rate.

Besides this traffic with guaranteed bandwidth, the platform makes possible the exchange of *IP* packets. These packets are inserted in the data field of the platform's packets and are transported to the destination over a special channel. This functionality allows the use of *TCP/IP* communications over the *SCI* and communication libraries based on *TCP*, such as *MPI*.

The platform was implemented as a set of Linux/RTAI modules.

The evaluation shown that even in scenarios with a high demand of communication, the reserved bandwidth is guaranteed for each channel.

Keywords: Real-time Communication, Distributed Systems, SCI Clusters .

1 Introdução

Com o advento das redes de comunicação extremamente rápidas como o *Fast Ethernet*, *Mirinet* e *SCI* (*Scalable Coherent Interface*) e o barateamento progressivo dos microcomputadores tornou-se muito comum a construção de *clusters* de computadores para resolver problemas de grande complexidade computacional. De modo geral o objetivo de um *cluster* é possibilitar a execução de aplicações distribuídas e paralelas em um hardware de custo relativamente baixo, mas que apresentam altas taxas de processamento.

Dentro do universo de aplicações que podem ser executadas dentro de um *cluster*, um pequeno grupo pode ser evidenciado. São aquelas onde não só o resultado lógico correto é esperado, mas também uma correção temporal. A correção nesse tipo de aplicação não se dá somente pelo resultado lógico de um determinado processamento, mas isso juntamente com o momento que esses resultados são gerados. Um sistema em que as aplicações apresentam restrições temporais é conhecido como sistema de tempo real. Normalmente esses sistemas são utilizados quando existe uma necessidade de interação com ambientes cuja dinâmica difere da dinâmica interna do sistema [CAM 98].

O tempo tem um papel fundamental nos sistemas de tempo real. Em uma aplicação de tempo real distribuída, os limites de tempo de cada tarefa podem depender de outras tarefas remotas e do tempo de comunicação entre elas. Isso porque tempos de transmissão não determinísticos de mensagens entre tarefas de tempo real podem afetar o tempo de processamento das tarefas que participam da comunicação. Assim, os limites de tempo locais dependem também da comunicação. Conseqüentemente, necessita-se de uma comunicação com garantias de tempo que pode ser provida por uma rede de tempo real.

A disponibilidade de comunicação de tempo real em redes de comunicação rápidas torna possível a execução de aplicações de tempo real de alto desempenho de forma distribuída. Aplicações de vídeo multimídia incorporam grandes fluxo de dados contínuos com restrições temporais e são exemplos de aplicações que se beneficiam com esse tipo de tecnologia.

Uma plataforma de software que permita a troca de mensagens com restrição temporal em *clusters* de alto desempenho não está disponível para grande parte das tecnologias disponíveis. Isso dificulta o desenvolvimento das aplicações de tempo real distribuídas de alto desempenho.

Entre as diferentes redes de comunicação utilizadas na interligação de nodos de um cluster encontra-se o *SCI*. O *SCI* baseia-se na norma IEEE 1596-1992 é caracterizado por permitir a conexão de até 64k nodos com uma grande largura de banda. Embora sejam extremamente eficientes quando utilizados em *clusters* rodando aplicações que requerem alto desempenho e comunicação, os adaptadores *SCI* não são apropriados para o transporte de tráfego de tempo real. Existem dois motivos principais para isso. Primeiro, o protocolo de acesso ao meio físico não permite o estabelecimento de limites temporais e é susceptível a contenções. Segundo, não existe uma priorização do tráfego de tempo real em detrimento do tráfego normal.

Para resolver esses problemas existem algumas propostas de extensão do hardware *SCI*. Essas propostas são baseadas na norma IEEE (P)1596.6 (extensão à

norma IEEE 1596-1992) e objetivam acrescentar a priorização de mensagens e assim acabar com as contenções e permitir tráfego de tempo real. Porém, essas soluções requerem uma nova implementação de adaptadores *SCI*.

O objetivo dessa dissertação é apresentar o projeto e a avaliação de uma plataforma de software que permita a comunicação de tempo real sobre o hardware *SCI* já existente e largamente utilizado e avaliado. Essa plataforma utiliza soluções de software para os problemas expostos acima. Porém, vai muito além ao propor uma pilha completa de protocolos que permitem o uso do paradigma de troca de mensagens de tempo real utilizando o meio físico do *SCI*.

Através dessa plataforma de software é possível a transmissão de fluxos contínuos de dados com garantias de transmissão. Dentre as garantias temos o *delay*(atraso) máximo e a reserva de largura de banda para determinada comunicação. Além disso, permite também a transmissão de tráfego normal através de canais de baixa prioridade.

O objetivo principal da plataforma proposta é prover primitivas de comunicação que permitam a execução de aplicações multimídia com tarefas que têm restrições de tempo e de precedência sobre um cluster *SCI*. Normalmente o requisito desse tipo de sistema é de tempo real flexível, onde a perda eventual de um limite de tempo não acarreta um dano irreparável ao sistema. Apesar disso, como será evidenciado no decorrer do trabalho, a plataforma desenvolvida pode também ser utilizada em sistemas de tempo real firme.

Algumas tecnologias de comunicação utilizadas em redes comuns já nasceram com especificação para o tráfego de tempo real. Como exemplo temos as redes ATM que possuem categoria de serviço própria para tráfego com restrições temporais. Outras tecnologias que não possuem um design de hardware apropriado para tempo real receberam camadas de software que implementam protocolos que tornam possível o tráfego de tempo real. Por exemplo, o *RETHER* [VEN 96] é um protocolo *Ethernet* de tempo real implementado em software por um grupo de pesquisa da Universidade Estadual de Nova York. Ele utiliza como protocolo de acesso ao meio determinístico a passagem de um *token* no lugar do *CSMA/CD* não determinístico original do padrão *Ethernet*. Uma proposta alternativa que utiliza o protocolo *TDMA* para disciplinar o acesso ao meio físico em adaptadores de rede *Ethernet* é descrita em [LAN 2002].

A plataforma de software proposta nessa dissertação acrescenta camadas de software ao hardware do *SCI* que implementam protocolos que garantam limites temporais de comunicação. Assim, é possível comunicação RT com a grande largura de banda oferecida pelos adaptadores de rede de alta velocidade aos quais pertence o *SCI*.

1.1 Motivação

A motivação desse trabalho nasceu de um projeto da União Européia chamado *EVENTS*. O objetivo desse projeto é desenvolver novos métodos de visão computacional e de sistemas de tempo real que tornem possível a construção de um sistema inovativo de interpolação de imagens a ser utilizado para transmissão de TV em cenários abertos. Uma grande importância é dada à aplicação do sistema em transmissões de partidas de futebol.

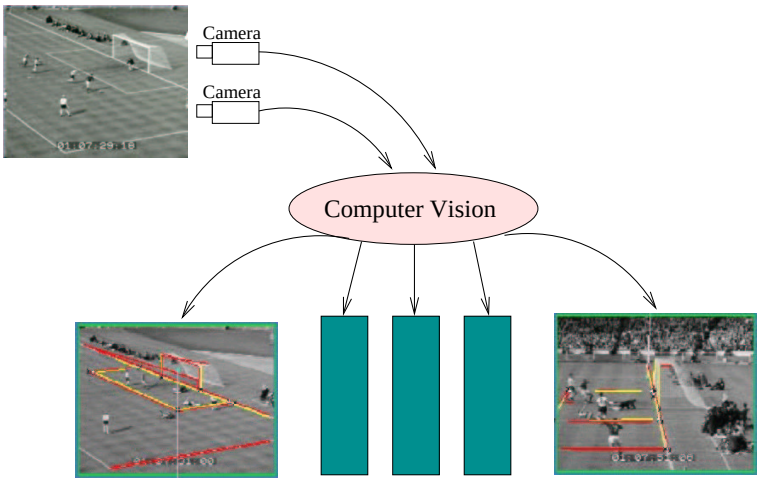

FIGURA 1.1 – Objetivo final do sistema *EVENTS*

O sistema *EVENTS* é baseado em um conjunto de câmeras posicionadas em posições diferentes em torno do cenário com o intuito de dar a maior cobertura possível. As câmeras devem ser sincronizadas no tempo. Usando imagens capturadas simultaneamente de um certo cenário, sob condições de luz similares e existindo alguma continuidade entre elas, o sistema é capaz de gerar visões interpoladas de um ângulo selecionado por um usuário humano (veja figura 1.1).

Além disso, o usuário é capaz de, usando algum dispositivo de apontamento, mudar o ponto de vista de uma maneira gradual e o sistema produz uma navegação virtual pelo ambiente. Dois protótipos do sistema formam propostos:

***Still Image Prototype*:** O protótipo de imagens fixas (SIP) é um sistema de baixo custo utilizado em aplicações como replay de futebol ou vídeo lentos. Não são necessários requisitos de tempo real nessa fase do projeto.

***Moving Images Prototype*:** O protótipo de imagens em movimento (MIP) é uma aplicação multiprocessada para a renderização de cenas interpoladas de seqüências de imagens em movimento. Assim, o usuário tem a ilusão de estar observando ao vivo vídeos com a perspectiva de locais onde não existem câmeras físicas.

O protótipo de imagens em movimento requer uma grande capacidade computacional que não é possível se obter um uma máquina monoprocessada. Assim, o projeto requer a utilização de uma arquitetura paralela ou um *cluster* de computadores para o cálculo da imagem interpolada em tempo real. Foi definido nos requisitos do projeto que um *cluster* de computadores PC utilizando adaptadores *SCI* deveria ser utilizado.

Devido à característica de tempo real necessária para o MIP, uma camada de software que torne possível esse tipo de comunicação sobre uma rede de computadores interconectados pelo *SCI* tornou-se necessária. Além disso, essa camada de software tornaria também possível a execução de uma grande gama de outras aplicações multimídias que têm requisitos de comunicação semelhantes.

Outra motivação para a construção da plataforma de comunicação foi o fato de que o MIP foi projetado para utilizar o paradigma de troca de mensagens enquanto que a plataforma *SCI* originalmente suporta memória distribuída compartilhada.

1.2 Apresentação do trabalho

A dissertação está organizada como segue. O capítulo 2 apresenta uma revisão bibliográfica sobre aspectos de comunicação de tempo real. São apresentadas as características desejáveis de uma comunicação de tempo real, além de alguns fundamentos de redes de computadores. No final do capítulo uma breve introdução da tecnologia *ATM* que provê canais para o tráfego de comunicação de tempo real é mostrada.

No capítulo 3 a infraestrutura de software e hardware utilizada no desenvolvimento da plataforma de comunicação é apresentada. O sistema operacional GNU/Linux, seu núcleo e uma extensão de tempo real são expostos. Por fim, o adaptador *SCI* que é o hardware de comunicação utilizado pelo sistema é mostrado. A proposta de extensão *SCI/RT* é discutida no fim do capítulo.

O capítulo 4 corresponde à parte central do trabalho. Apresenta o desenvolvimento da plataforma de comunicação de tempo real. As diversas camadas que formam a sua arquitetura são exploradas e as decisões de projeto analisadas. Esse capítulo cobre desde a camada de acesso ao meio, responsável por evitar o problema da contenção no *SCI*, até a API desenvolvida que permite a troca de mensagens de tempo real entre tarefas distribuídas no *cluster*.

Uma avaliação da plataforma de comunicação desenvolvida é apresentada no capítulo 5. A metodologia de avaliação utilizada assim como as medições realizadas são mostradas no capítulo.

As considerações finais do trabalho são apresentadas no capítulo 6. Aspectos positivos e negativos da plataforma são discutidos assim como sugestões de melhorias a serem desenvolvidas em trabalhos futuros.

2 Revisão Bibliográfica

2.1 Introdução

Diversos aspectos estão envolvidos em uma comunicação RT em um sistema distribuído. Basicamente, para que essa comunicação possa ocorrer, temos que ter o sistema físico de comunicação e os protocolos que dêem suporte para que uma comunicação de tempo real possa ocorrer. Mas não é só isso. Para que o sistema seja eficiente, outras funcionalidades são necessárias. Dentre dessas estão, por exemplo, os diversos escalonamentos que podem ser feitos. Esses escalonamentos podem ser executados de forma estática, se conhecermos todo o sistema antes da sua inicialização, ou dinamicamente, onde o sistema deve ser flexível para adicionamento de mais tarefas ou mensagens.

A plataforma proposta nesse trabalho utiliza o *SCI* como sistema físico de comunicação e acrescenta uma série de camadas que implementam protocolos que permitem o oferecimento de canais de comunicação com garantias de banda.

Esse capítulo faz uma revisão bibliográfica sobre duas áreas relacionadas com comunicação em tempo real. A primeira trata de características comuns de uma comunicação RT. Já a segunda área trata dos aspectos da rede de comunicação necessários para prover uma comunicação determinística que é a carecterística principal de sistemas de tempo real.

Adicionalmente, no final do capítulo, uma tecnologia de rede que provê canais de comunicação com garantia de banda semelhanta aos canais disponibilizados pela plataforma é apresentada. Essa tecnologia é a ATM (*Asynchronous Transfer Mode*) cujos canais providos podem ser utilizados para o transporte de tráfego de aplicações multimídia. Nesse capítulo é feita uma breve apresentação do ATM e as características que permitem o transporte de dados multimídia como áudio e vídeo.

2.2 Característica da comunicação RT

2.2.1 Escalonamento e temporização das mensagens

Em um sistema distribuído de tempo real, as tarefas que podem ser esporádicas ou periódicas chegam dinamicamente para cada nodo e possivelmente se comunicam com tarefas que executam em outros nodos. Essa atribuição de tarefas a diversas máquinas cria a necessidade de diversos tipos de escalonamentos não existentes para aplicações tempo real de máquina única. Esses tipos de escalonamento estão listados abaixo [NOR 2000]:

1. Escalonamento de tarefas dentro de um nodo (escalonamento local). Esse escalonamento também está presente em nodos únicos, mas em sistemas distribuídos muitas vezes uma coordenação entre os diversos escalonamentos locais é necessária.

2. Migração de tarefas entre nodos (escalonamento global).

3. Escalonamento das mensagens em um link de comunicação.

Esse último escalonamento é utilizado para garantir comunicação entre as tarefas dentro de um nodo e em nodos diferentes. Em sistemas de tempo real distribuídos, a execução das tarefas e a entrega das mensagens dentro do limite de tempo associado é de grande importância. Sendo assim, o tempo entre uma tarefa colocar uma mensagem na fila de transmissão e a tarefa destinatária efetivamente receber a mensagem deve ser limitado. Esse tempo é conhecido por *end to end delay* (atraso fim a fim).

Segundo [TIN 94], normalmente esse tempo é dividido em 4 maiores componentes:

Atraso de geração: Tempo necessário para a aplicação gerar e colocar na fila de transmissão a mensagem;

Atraso de fila: Tempo que a mensagem leva para ganhar acesso ao meio de comunicação após ser enfileirada;

Atraso de transmissão: Tempo que a mensagem leva para ser transmitida no dispositivo de comunicação;

Atraso de entrega: Tempo que é gasto para processar a mensagem no nodo destino antes de finalmente chegar na tarefa destinatária.

O *atraso de geração* é o tempo do pior caso da geração pela aplicação da mensagem e sua colocação na fila de mensagens. Esse tempo representa um tempo gasto pela aplicação para gerar a mensagem e o tempo para essa mensagem ser colocada na fila de mensagens do sistema.

Já o *atraso de fila* representa o tempo que uma mensagem gasta para ser removida da fila pelo dispositivo de comunicação. Num canal de comunicação ponto a ponto, a mensagem precisa esperar por outras mensagens mandadas pelo mesmo nodo; já em um canal compartilhado precisa concorrer com mensagens mandadas por outros processadores também.

O *atraso de transmissão* é o tempo necessário para a mensagem ser mandada após ter sido retirada da fila. O *atraso de entrega* é o montante de tempo necessário para que os dados recebidos sejam processados e a entrega na aplicação destino seja feita. Isso inclui o tempo de decodificar cabeçalhos de pacotes, copiar dados de mensagens entre buffers e notificar que a mensagem foi entregue. Esse tempo pode levar uma parte significativa do tempo total de comunicação.

Para manter a soma desses tempos previsíveis, existem diversos mecanismos. Para uma rede geograficamente distribuída [1], por exemplo, pode-se empregar uma rede de chaveamento de circuitos, que pode prover entrega de mensagens em tempo real de forma fácil. Para isso, basta reservar uma parte da largura de banda da rede de acordo com o pico de comunicação de cada aplicação. Mas como o tráfego de tempo real pode ser em forma de rajadas, a não ser que se consiga usar a largura de banda não utilizada para tráfego não de tempo real, pode existir um grande desperdício de banda [ARA 93]. Note-se que esse sistema não resolve o problema no nível de enlace, só de rede. Além disso, a latência do sistema deve ser levada em conta.

[1] Não é o tipo mais comum de comunicação em tempo real, pois o normal é se ter um cluster onde as aplicações rodam numa rede de alta velocidade

2.2.2 Características desejáveis da comunicação de tempo real

Todos os métodos de comunicação em tempo real buscam prover uma entrega com baixas perdas ou nenhuma perda (tempo real "soft" e "hard"). As seguintes características ainda são desejadas [ARA 93]:

- *Latência* conhecida e mínima para transmissão de mensagens.
- *Jitter* conhecido e mínimo para latência de transmissão.
- Habilidade de integrar serviços normais com os de características de tempo real .
- Adaptabilidade a mudanças nas conexões.
- Alta utilização da banda da rede.
- Baixo *overhead* no cabeçalho das mensagens.
- Baixo *overhead* no processamento dos protocolos de tempo real .

Os dois primeiros itens devem continuar válidos o maior tempo possível mesmo na presença de *overload*.

Conforme [HUM 2001], ao se considerar a rede isoladamente, a latência e o *jitter* de transmissão de mensagens resultam na integração do tempo originado pelo método de acesso ao meio físico com o tempo devido à propagação do meio. Uma infra-estrutura que provê as características mostradas acima é a base necessária para que possa haver garantia temporal nas camadas mais altas, das quais as aplicações se servem.

2.2.3 Semântica das mensagens

Uma mensagem transmitida em um sistema RT pode ter dois tipos de informação:

- Mensagem do tipo *evento*: mensagem que deve ser consumida exatamente uma vez pelo consumidor na ordem temporal exata. Por exemplo, uma mensagem evento pode ser transmitida na mudança de um estado de algum sensor que está sendo monitorado.
- Mensagem do tipo *estado*: pode ser processado um número arbitrário de vezes e sempre sobrescreve a antiga versão. Por exemplo, um sensor de temperatura pode mandar a sua medida de temperatura em tempos pré-determinados (ciclicamente).

2.3 Rede de comunicação

Para existir uma comunicação RT, o aspecto mais básico a ser tratado são as redes de comunicação e seus protocolos. Nessa seção, serão vistos alguns aspectos relevantes de software/hardware de redes que devem ser levados em conta em um projeto de comunicação RT.

TABELA 2.1 – Propriedades de uma rede abstrata

Propriedade	Nome	Descrição
An1	Difusão	Todos os destinatários devem receber o mesmo quadro
An2	Detecção de Erros	Destinatários detectam qualquer quadro corrompido pela rede, em transmissão local
An3	Ordenação de Rede	Quaisquer dois quadros reconhecidos em dois pontos de acesso diferentes são reconhecidos na mesma ordem
An4	Full-Duplex	Quando solicitada a indicação de reconhecimento de quadro por parte do receptor, esta deve ser fornecida ao emissor
An5	Consistência	Destinatários recebem uma mensagem não corrompida em tempos diferentes de no máximo t conhecido
An6	Atraso de transmissão limitado	Todos os quadros da rede são enviados com um atraso de transmissão máximo
An7	Grau de Omissão Limitado	Em um intervalo conhecido, erros de transmissão podem afetar no máximo k transmissões.
An8	Inacessibilidade Limitada	Em um intervalo conhecido, a rede pode estar inacessível no máximo j vezes, com uma duração máxima T_{max}.

Fonte: [BAC 98]

2.3.1 Propriedades de uma rede abstrata

Baseado nas exigências impostas por serviços, Veríssimo [VER 91] propôs um conjunto de propriedades básicas que uma rede de comunicação teórica (abstrata) deveria conter. Esse conjunto de propriedades é mostrado na tabela 2.1.

A propriedade An1 garante o *broadcast* da rede. Já a propriedade An2 diz respeito à criação de um canal livre (na medida do possível) de erros. Essas propriedades são desejáveis em comunicação RT.

A propriedade do ordenamento (An3) é importante para que todos os nodos compartilhem uma mesma visão quanto a ordem de ocorrência dos eventos no sistema.

Já a propriedade An5 diz respeito à diferença de tempo de recepção entre os diversos receptores. Esse tempo deve ser limitado em comunicações RT. A propriedade An6 também é importante para mensagens RT. É imprescindível conhecermos o atraso máximo da rede para que possamos ter um limite máximo no tempo de transmissão de mensagens fim a fim (*end to end delay*). Ver seção "Escalonamento e temporização das mensagens").

A propriedade An7 é bem interessante, pois não permite que a rede fique indeterminadamente em um estado errôneo, no qual todos os limites de tempo de mensagens RT são perdidos. A propriedade An8 também é relevante para que os limites de tempo máximos das mensagens sejam cumpridos.

2.3.2 Tecnologia da rede

Uma questão que tem relevância, quando se trata de uma rede empregada para comunicação de tempo real, é o tipo de tecnologia utilizada na transmissão de dados. As duas tecnologias que podem ser empregadas para a transmissão tanto de tráfego normal quanto de RT são:

Redes de difusão: As redes de difusão têm apenas um canal de comunicação, compartilhado por todas as máquinas. As mensagens, que são chamadas de pacotes, enviadas por uma máquina são recebidas por todas as demais. Um campo de endereço dentro do pacote identifica o destinatário. Quando um pacote é recebido por uma máquina, ela analisa o campo de endereço. Se o pacote é endereçado a essa máquina, ela o processará, senão ele será descartado.

Redes ponto a ponto: As redes ponto a ponto consistem em múltiplas conexões de pares individuais de máquinas. Para ir da origem ao destino, um pacote desse tipo de rede muitas vezes tem que passar por inúmeras máquinas intermediárias, que o direcionam adiante através de um processo chamado de roteamento. Normalmente, redes pequenas utilizam a tecnologia de difusão, enquanto que as maiores o roteamento de pacotes dentro de uma rede ponto a ponto [TAN 97].

Os protocolos (e problemas) enfrentados para se prover uma comunicação RT estão intimamente ligados com a tecnologia utilizada. Além disso, a propriedade An1 terá sua implementação naturalmente facilitada em redes do tipo difusão.

2.3.3 Topologias de redes

Outra questão relevante no domínio das redes de comunicação utilizadas em um cluster RT é a topologia utilizada. Para a comunicação RT não existe nenhuma restrição quanto ao tipo escolhido.

A topologia está ligada diretamente com a tecnologia de rede que está sendo utilizada. Para redes de difusão, as topologias mais empregadas são mostrada na figura 2.1; note-se que, na topologia em anel de uma rede de tecnologia de difusão, cada bit é propagado de modo independente, sem esperar o restante do pacote ao qual ele pertence. Geralmente, cada bit percorre todo o anel no intervalo de tempo em que alguns bits são enviados.

Já para redes ponto a ponto, as topologias mais empregadas são mostradas na figura 2.2. A topologia do tipo anel pode ser também utilizada para redes ponto a ponto, na qual um nodo recebe o pacote inteiro e o retransmite para a próxima máquina.

2.3.4 Camadas do sistema de comunicação

O desejo de se prover uma comunicação em tempo real pode refletir nas várias camadas da camada OSI de uma rede. Devido à ampla aceitação desse modelo como

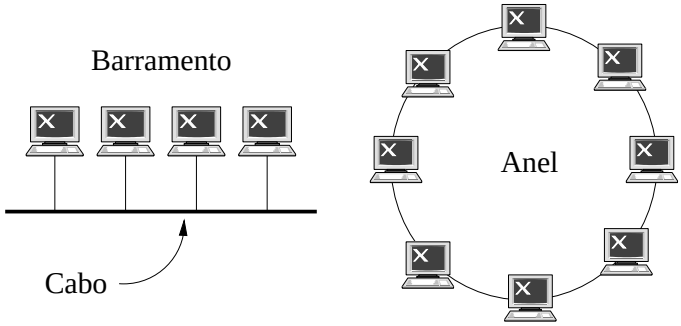

FIGURA 2.1 – Topologias possíveis para uma rede de difusão [TAN 97]

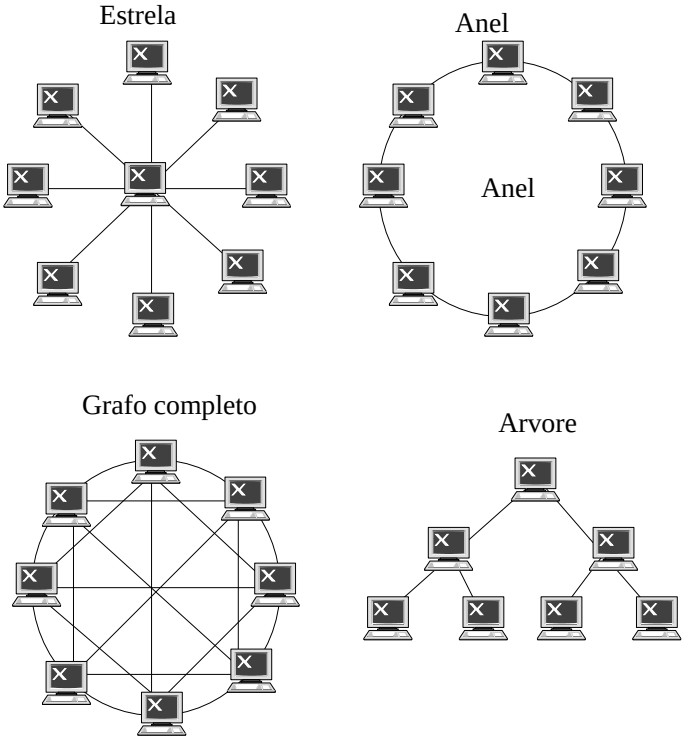

FIGURA 2.2 – Topologias possíveis para uma rede ponto a ponto [TAN 97]

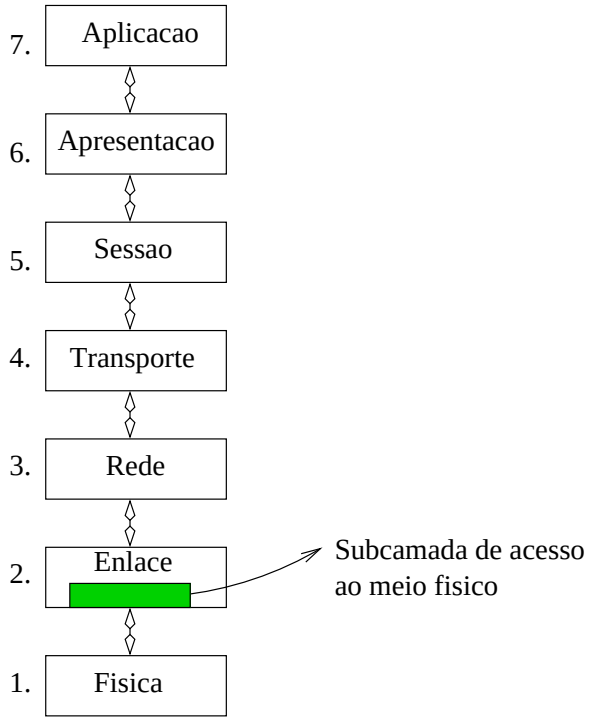

FIGURA 2.3 – Modelo de referência *OSI* [TAN 97].

referência, ao analisar-se a comunicação RT não se pode deixar de utilizar o modelo com certas modificações.

O modelo de referência OSI com a respectiva camada de acesso ao meio pode ser vista na figura 2.3.

Camada Física

A camada física da rede trata da transmissão de bits puros através de um canal de comunicação. O projeto de rede é feito de modo que se um bit 1 seja colocado em uma extremidade da rede, o outro lado receba também um bit 1. Várias questões devem ser analisadas nessa camada, como a tensão utilizada para representar o 0 ou o 1, o tempo que um bit deve durar, etc...

Camada de Enlace

A camada de enlace, segundo [TAN 97], tem como função principal transformar um canal de transmissão bruta, que é a camada física, em uma linha que pareça livre de erros. Para executar essa tarefa, os dados de entrada são divididos em quadros de dados, transmitidos seqüencialmente e, caso recebidos com sucesso, os quadros de reconhecimento são processados pelo receptor. Se um quadro for danificado por ruído, a camada de enlace deve reconhecer o erro e pedir a retransmissão. Na rede abstrata apresentada acima, essa camada diz respeito à propriedade An2.

Para redes de difusão, a camada de enlace deve também controlar o acesso a um canal compartilhado. Isso normalmente é feito pela sub-camada de acesso ao meio. Dentro da camada de enlace, normalmente, a implementação dessa sub-camada tem um grande impacto na transmissão de mensagens RT.

Sub-camada de acesso ao meio

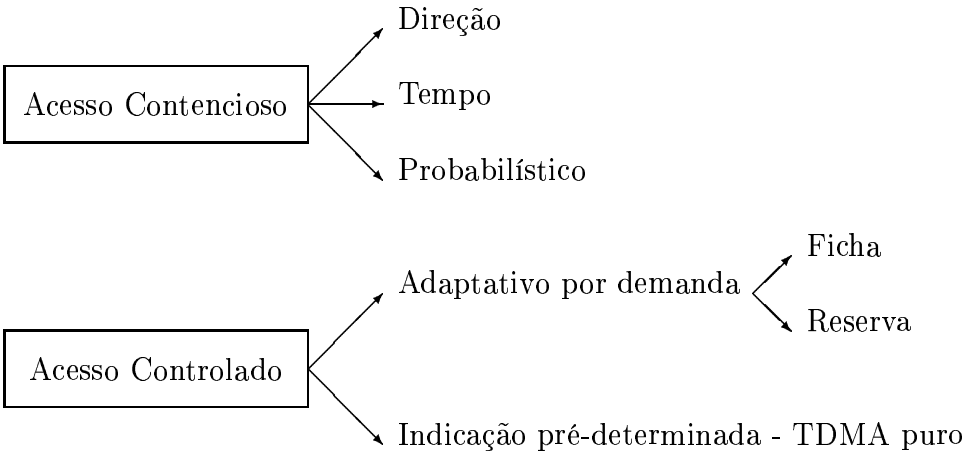

FIGURA 2.4 – Classificação quanto às colisões

Para uma rede do tipo difusão ser empregada, o canal de comunicação deve estar recebendo dados de uma máquina apenas por vez. É preciso utilizar, para isso, um mecanismo de arbítrio para resolver o conflito quando duas ou mais máquinas quiserem fazer uma transmissão simultaneamente. Esse mecanismo pode ser centralizado ou distribuído.

A camada de acesso a um meio físico compartilhado (ou seja, uma rede de difusão) pode ser vista como dois processos: um árbitro de acesso e um controlador de transmissão.

Eles têm a seguinte funcionalidade:

1. Arbitragem de acesso ao meio físico: O árbitro de acesso é responsável por determinar quando um nodo pode mandar uma mensagem pelo canal.

   Para ilustrar a arbitragem ao meio físico, considere-se o protocolo *Token Bus*. Nesse protocolo, nodos na rede são dispostos em um anel lógico. Quando um nodo terminou de utilizar o anel, o árbitro pode dar o canal ao próximo nodo. Isso pode ser implementado pela passagem de um token [MAL 95]. Cada nodo tem direito de transmitir H unidades de tempo. Após isso, o controlador de transmissão passa o token para o próximo nodo no anel.

   É importante ressaltar que existem outras tantas maneiras de executar a arbitragem do acesso do meio físico. Uma classificação quanto às colisões pode ser vista na figura 2.4. A primeira grande divisão é baseada no fato do acesso ao canal estar livre de colisão (acesso controlado) ou não (acesso contencioso). Pode-se ver uma descrição dessa classificação detalhada em [SAN 91].

   Uma outra observação importante é que o processo de arbitragem pode ser implementado tanto de forma centralizada como de forma distribuída.

   Em se tratando de comunicação RT, normalmente, a utilização de protocolos contenciosos como o CSMA (*Carrier Sense Multiple Acess*) é um procedimento

inviável, uma vez que, em tempo real, o atraso máximo das mensagens deve ser conhecido.

2. Controle de transmissão
   O processo controlador de transmissão tem como função a determinação de quanto tempo essa transmissão pode levar.

   Em contraste com o processo de liberação de acesso, o processo de controle de transmissão tem um trabalho menor a desempenhar. Uma técnica comum é deixar cada nodo mandar somente uma mensagem. Outra abordagem é deixar o número de mensagens livre. Em geral, existem duas classes. Em um controle de transmissão estático, a quantidade de tempo que um nodo pode continuar a mandar uma mensagem sobre um canal é fixa em um tempo pré-determinado. Em um controle dinâmico, esse tempo é determinado durante a operação normal da rede.

   A operação do controle de transmissão tem um grande impacto no cumprimento dos limites de tempo da entrega das mensagens. Não podemos ter previsibilidade sem garantia de uma mínima quantidade de tempo na qual um nodo pode transmitir interruptamente. Mas, se deixarmos um nodo monopolizar o meio de acesso compartilhado, podemos ter limites de tempo dos outros nodos sendo quebrados.

A camada de acesso ao meio com seus 2 processos está sendo aqui focada pela sua importância dentro da camada *OSI*, pois a decisão do árbitro de acesso afeta métricas como o *throughtput* e o tempo de mensagem da rede. Para comunicação em tempo real, deve-se observar, ainda, os seguintes pontos [MAL 95]:

Correção temporal: Para satisfazer as constantes de tempo das mensagens de tempo real, o próximo nodo da rede a ganhar o acesso ao canal deve ser escolhido de forma correta. Por exemplo, dar preferência para um nodo com uma mensagem cujos limites de tempo não são críticos pode fazer com que outro nodo perca o seu limite de tempo.

***Overhead*:** Normalmente, em uma rede, os nodos têm poucas informações sobre as mensagens que estão esperando nos outros nodos. Conseqüentemente, o processo de arbitrar o acesso pode decidir baseado em informações incompletas ou não válidas. Para reduzir esse problema o processo de arbitrar o acesso deve tentar coletar informações sobre o estado dos outros nodos. Porém, isso traz um *overhead* ao sistema que torna menor a largura de banda para as mensagens.

Camada de Rede A camada de rede controla a operação da sub-rede. A questão fundamental é como os pacotes são roteados da origem para o destino. O endereçamento dos nodos da rede também é tratado nessa camada. Segundo [TAN 97], nas redes de difusão, o problema do roteamento é simples. Sendo assim, essa camada tende a ser pequena. Para se prover comunicação em RT, essa camada deve normalmente dar prioridade mais alta para o roteamento dos pacotes que têm essa característica, deixando em segundo plano os pacotes (quadros) de comunicação normal.

Para o roteamento das informações, há dois tipos de protocolos:

Circuito Virtual: No início da conexão, é estabelecido uma rota do emissor para o receptor passando por vários nodos intermediários. Cada pacote a ser transmitido irá ter então o número do circuito virtual estabelecido. Não é necessário ter o endereço da máquina destino visto que o próprio circuito contém essa informação;

Datagramas: Cada pacote contém o endereço do emissor e do receptor da mensagem. Durante seu trajeto, cada nodo intermediário analisa o endereço do receptor e manda o pacote pela melhor rota segundo sua tabela de roteamento [COU 2001]. Note-se que dois pacotes da mesma conexão podem seguir por caminhos diferentes. Sendo assim, para uma comunicação RT, a primeira opção é a que traz maior determinismo.

Observe-se que o roteamento não é necessário em redes de difusão (puras).

Camada de transporte A idéia básica dessa camada é receber os dados das camadas superiores, dividi-los em unidades menores e passá-los para camada de rede, além de garantir que todas essas unidades cheguem corretamente ao outro lado. Outra função dessa camada é o ordenamento das mensagens [TAN 97].

Normalmente, essa camada cria conexões para o envio de informações na forma de uma *stream*. O tipo de conexão mais comum é o canal ponto a ponto, livre de erros, que libera os bytes na ordem que foram enviados. No entanto, podem existir tipos de mensagens que não são orientados a conexão nem livre de erros. Como exemplo do primeiro, há o *TCP* e do segundo tipo o *UDP*.

O controle de fluxo também é uma tarefa importante que pode ser realizada nessa camada. Ele serve para que uma máquina rápida não sobrecarregue uma lenta. Esse tipo de controle pode ser realizado também na camada de enlace. Existem duas abordagens para o controle de fluxo:

Controle de fluxo explícito: O receptor confirma explicitamente o recebimento correto da mensagem, enviando uma mensagem de confirmação. Assim, o transmissor sabe que uma nova mensagem pode ser mandada. O problema desse tipo de abordagem é que um *jitter* grande na latência da comunicação é introduzido, pois *timeouts* são utilizados, além de não se saber quantas transmissões serão necessárias para o recebimento correto.

Controle de fluxo implícito: O receptor e o emissor combinam no início da operação do sistema sobre a hora de transmissão de cada mensagem. Isso requer uma base global de tempo. O transmissor se compromete a enviar certas mensagens em tempos pré-determinados. A detecção de erros é feita simplesmente se a mensagem deixou de chegar no horário determinado [HUM 2001].

Resumidamente, a camada de transporte é responsável pela ordenação da rede (característica An3) e também por detecção de erros (característica An2).

Nessa camada, pode-se colocar serviços de comunicação que tenham limites de tempo (*bounding comunication*).

As outras camadas do modelo *OSI* não serão abordadas no presente trabalho, existindo vasta literatura a respeito, e.g., [TAN 97, TAN 2001, COU 2001] .

TABELA 2.2 – Categorias de serviço ATM

Classe	**Descrição**	**Exemplo**
CBR	*Constant bit rate*	Circuito T1
RT-VBR	*Variable bit rate: real time*	Videoconferência (tempo real)
NRT-VBR	*Variable bit rate: non-real time*	Correio eletrônico multimídia
ABR	*Available bit rate*	Navegação Web
UBR	*Unspecified bit rate*	Transferência de arquivos em segundo plano

Fonte: [TAN 97]

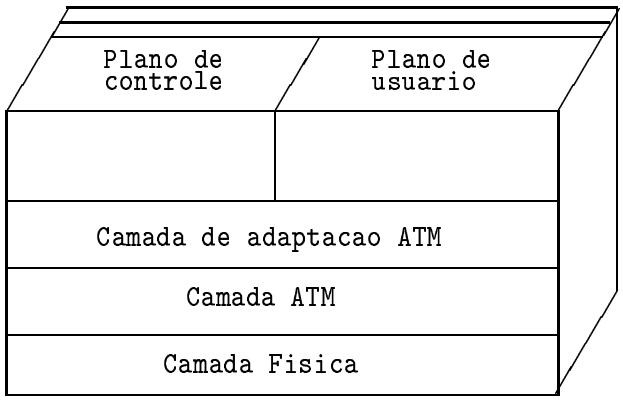

FIGURA 2.5 – Modelo de referência ATM

2.4 Tecnologia de rede ATM

A tecnologia ATM se baseia na comutação de células . Cada célula ATM é composta por um cabeçalho de 5 bytes e dados úteis no tamanho fixo de 48 bytes. O ATM foi proposto tendo em vista questões como disponibilidade de vídeo e áudio sob demanda e substituição do sistema telefônico. Por isso, a especificação ATM propõe diversas categorias de serviços necessárias tanto para tráfego de tempo real (vídeo, áudio) quanto tráfego normal sem reserva de banda. As categorias de serviços disponíveis são mostradas na tabela 2.2.

Devido às redes ATM serem orientadas a conexão, são necessárias mensagens para configurar a rede antes do estabelecimento de conexão. Após, pacotes, que no caso do ATM, por terem tamanhos fixos são, chamados de células, podem ser enviados. Existe garantia na ordem de chagada dentro de uma conexão.

2.4.1 Modelo de referência

Segundo [TAN 97], o ATM possui seu próprio modelo de referência que é diferente do OSI (veja figura 2.5).

O nível físico foi projetado de forma que o ATM é independente do meio de transmissão. Já a **camada ATM** lida com o transporte de células, com estabelecimento e liberação dos circuitos virtuais (que são responsáveis pela conexão). Na

verdade, o elemento básico da camada ATM é o circuito virtual. O fato de uma rede ATM fornecer a possibilidade de criação de circuitos virtuais com garantias de transferência de bits a torna atraente para aplicações de tempo real.

Os circuitos virtuais são unidirecionais e não oferecem confirmação. Um conjunto de circuitos virtuais com mesma origem e destino pode ser agrupados em um caminho virtual. Um caminho virtual é sempre roteado pelos mesmos *hops*, se esse caminho por algum motivo tem que ser modificado, todos os circuitos que ele contém tem o caminho modificado de forma conjunta. Os *switches* (roteadores) ATM no fim de um canal são responsáveis pela multiplexação e demultiplexação do fluxo gerado pelos diversos circuitos virtuais [KOP 97].

Para ser possível o envio de pacotes de tamanho diferente de uma célula, foi criada a camada **ALL** (*ATM Adaption Layer*). Essa camada é responsável pela criação das células ATM a partir de pacotes vindo das camadas superiores.

Além dessas camadas, temos o plano de usuário, que foge dos modelos tradicionais 2D, trata do transporte de dados, controle de fluxo e correção de erro e outras funções de usuário [TAN 97]. Já o plano de controle gerencia as conexões.

2.4.2 Categorias de serviço

As categorias de serviço, mostradas na tabela 2.2, são responsáveis pela característica da rede ATM, possibilitar a transmissão de tráfego com diferentes tipos de requisitos. Segundo [TAN 97] a classe *Constant Bit Rate* tem a finalidade de emular um fio de cobre ou fibra ótica e é responsável pela transmissão de voz nos circuitos T1 de voz PCM de circuitos telefônicos. Também pode ser utilizada para áudio e vídeo em tempo real, já que garante o tráfego a uma taxa constante de transmissão. Já a classe *Variable Bit Rate* pode ser utilizada para tráfego em tempo real ou não. Vídeo e áudio compactado com algoritmos usuais (como MPEG) normalmente tem uma taxa variável de dados em relação ao tempo (pois a taxa de compressão tem correlação com a parte dos dados que está sendo comprimida), sendo assim, essa classe de serviço foi criada. Na classe *RT-VBR* o retardo máximo de um pacote é mais rígido que na classe *NRT-VBR*.

As outras classes não são relevantes para a comunicação de tempo real. A título de curiosidade, a categoria de serviço *Available Bit Rate* é utilizada para aplicações onde não se tem muita informação sobre o tráfego (normalmente em rajadas) e a *Unspecified Bit Rate* não apresenta garantias nenhuma e é a de menor qualidade dentre as disponíveis. Em caso de congestionamento, as células dessa classe são as primeiras a serem descartadas.

Do ponto de vista da performance de tempo real, a tecnologia ATM é adequada para o fornecimento de serviços básicos para sistemas de tempo real distribuídos em áreas geograficamente distantes.

2.4.3 Avaliação

Embora o ATM provenha canais de tempo real que podem ser utilizados na transmissão de mensagens de tempo real, ele difere bastante da plataforma de comunicação proposta nesta dissertação. Isso porque ele foi projetado para ser utilizado em redes multi-hop geograficamente distribuídas e não em clusters de tempo real. A plataforma proposta, assim como o ATM, é baseada em diversas camadas, mas o nível de rede não é necessário pois não há roteamento.

3 Ambiente de execução

3.1 Introdução

A arquitetura que é proposta nesse trabalho baseia-se em uma infra-estrutura de software e hardware bem definida. A base de hardware utilizada no desenvolvimento da plataforma de comunicação de tempo real é um cluster. Um cluster consiste de uma coleção de computadores autônomos que estão fisicamente conectados por uma rede de alto desempenho e trabalham conjuntamente para a resolução de problemas [BAR 2000]. A conexão entre os nodos é feita por redes de comunicação de alto desempenho, tais como o Fast e Gigabit Ethernet [CUN 99], Myrinet [BOD 95] e *SCI*.

No cluster utilizado para desenvolvimento da arquitetura proposta, cada nodo é um computador PC monoprocessado. A tecnologia de rede utilizada é a *SCI* devido ao requisito prévio do projeto *EVENTS*. Esse requisito deve-se à disponibilidade de alta largura de banda e escalabilidade das redes *SCI*.

Já a infra-estrutura de software escolhida baseia-se em plataformas de software livre existentes. A escolha de um sistema operacional livre, com suporte a tempo real e que fornecesse uma grande variedade de serviços que poderiam ser utilizados para o desenvolvimento da plataforma de comunicação foi uma meta do projeto.

Tendo em vista aspectos como disponibilidade de serviços e manutenibilidade, decidiu-se utilizar o sistema operacional Linux junto com alguma extensão de tempo real do núcleo.

A decisão de utilizar o sistema Linux baseia-se no fato de ele ser amplamente difundido além de ser um sistema estável e de código fonte aberto. Outros sistemas, como o FreeBSD, contam também com diversas vantagens aqui expostas mas devido à menor disponibilidade de extensões de tempo real, não se mostraram adequados à aplicação proposta.

Como o sistema Linux padrão é composto por um núcleo monolítico não preemptável, não sendo portanto adequado para execução de tarefas de tempo real, foi necessária a utilização conjunta de uma extensão de tempo real.

Dentre as extensões de tempo real existentes, pode-se citar o *RT-Linux*, o *RedLinux* e o Linux *RTAI*.

Dentre eles escolheu-se o *RTAI* como extensão de tempo real a ser utilizada no projeto devido a sua disponibilidade como software livre e a decisão do projeto *RTAI* de incorporar um API com grande variedade de serviços, ao contrário dos concorrentes. Assim, várias dessas rotinas que o *RTAI* provê são utilizadas pela plataforma. Na figura 3.1 podemos ver a infra-estrutura de hardware/software utilizada juntamente com a plataforma de comunicação proposta nesta dissertação.

A extensão de tempo real *RTAI* juntamente com o sistema operacional Linux são responsáveis pelo oferecimento dos serviços básicos que serão utilizados pela plataforma na sua sincronização, temporização, alocação de memória e intercomunicação com as tarefas de tempo real que se comunicam através da arquitetura proposta. Já a rede *SCI* é responsável fisicamente pela transmissão das mensagens enviadas pelos processos de tempo real de um nodo para processos em nodos remotos através da plataforma de comunicação.

Os tópicos a seguir abordam essa infra-estrutura de hardware e software uti-

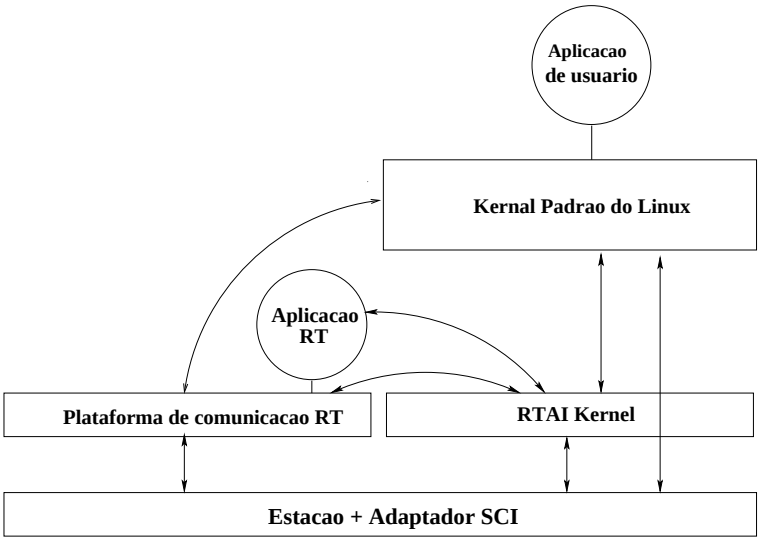

FIGURA 3.1 – Infra-estrutura de hardware/software utilizada juntamente com a plataforma de comunicação proposta

lizada como base para a construção da plataforma de comunicação. Inicialmente alguns aspectos do Linux e do respectivo núcleo serão abordados. A seguir a extensão de tempo real RTAI será apresentada. Por fim, o adaptador de rede *SCI* será analisado.

3.2 GNU/Linux

O GNU/Linux é um sistema operacional que faz parte da família dos Unixs. Originalmente foi desenvolvido para ser utilizado em arquiteturas PC (Intel 80x86), atualmente roda nos mais diversos tipos de plataforma, como Alpha Digital, estações Sparc ou mesmo em Amigas e Machintosh com processador PowerPC.

O núcleo do sistema operacional foi desenvolvido pela iniciativa de um estudante finlandês chamado Linus Torvalds. Normalmente uma distribuição Linux inclui um grande conjunto de softwares e ferramentas. Grande parte dessas ferramentas foram desenvolvidas pelo projeto GNU (Gnu is not Unix) da fundação do software livre [1] [OLI 2001].

Devido a uma intensa interação entre os módulos criados pela arquitetura descrita com o núcleo de tempo real e o núcleo padrão do Linux, além do desenvolvimento de um controlador de dispositivo de rede, é importante uma descrição mais detalhada de alguns aspectos do núcleo do Linux.

3.2.1 Núcleo do sistema operacional

As principais características do núcleo do Linux são mostradas na tabela 3.1.

[1] Software livre não significa simplesmente software grátis, e sim, a garantia de uma série de liberdades, como o direito ao código fonte para estudo ou modificação. Para maiores detalhes acessar *www.gnu.org*

TABELA 3.1 – Algumas características do núcleo do sistema operacional Linux

Característica	Descrição
Multitarefa	O Linux suporta multitarefa verdadeiramente preemptiva. Nenhum processo precisa se preocupar em deixar tempo disponível de CPU para outros processos concorrentes
Multiusuário	Vários usuários podem utilizar o sistema simultaneamente
Multiprocessador	Desde a versão 2.0 o Linux oferece suporte para arquiteturas multiprocessadas [BEC 99]
Carga de executáveis sob demanda	Somente parte do executável necessária no momento para a execução é carregada na memória
Proteção de memória	A área de memória de cada processo é mantida segura contra acessos indevido de outros processos
Paginação	O Linux utiliza a paginação oferecida pela MMU (*Memory Management Unit*) e também implementa o mecanismo de memória virtual

Num ambiente de execução de um sistema Unix como o Linux, vários processos requisitam serviços e recursos como tempo de computação, memória e conexões de rede. O núcleo deve ser capaz de tratar todas essas requisições de forma apropriada. As tarefas que o núcleo deve realizar podem ser divididas conforme a figura 3.2.

As funcionalidades exibidas na figura são:

Gerenciamento de processos: Um processo é definido como um programa em execução. A tarefa de criá-los é delegada ao núcleo do sistema operacional, assim como a comunicação entre os diversos processos. Os processos são também escalonados pelo núcleo e esse age de forma preemptiva: se o processo não liberar voluntariamente a CPU após seu limite de tempo ter expirado, ele será retirado da CPU pelo núcleo do Linux.

Gerenciamento de memória: Em um sistema operacional multitarefa como o Linux, cada processo deve receber uma área de memória e essa área deve ser protegida de acessos errôneos feitos por outros processos. Além disso, a alocação dinâmica de memória também deve ser efetuada.

Sistemas de arquivo: O núcleo do sistema operacional Linux suporta diversos tipos de sistemas de arquivo. Para isso, utiliza-se do conceito de sistema de arquivos virtual que facilita a implementação de diferentes tipos de sistemas de arquivo.

Controle de dispositivos: Para o controle dos mais diversos dispositivos que podem estar conectados à unidade de processamento, existe um código específico para cada tipo de dispositivo. Esse código é conhecido como controlador (*driver*) de dispositivo.

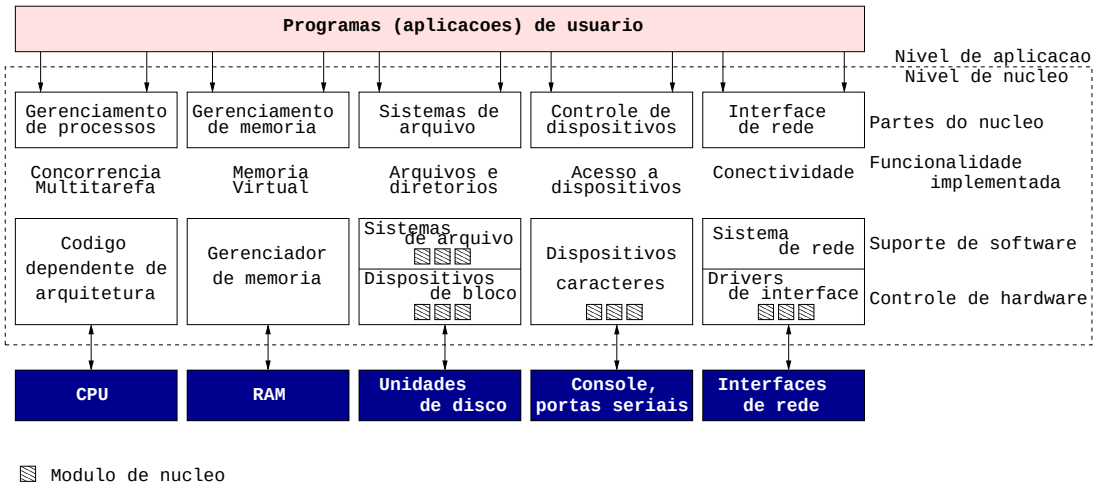

FIGURA 3.2 – Visão das funcionalidades do núcleo do sistema operacional Linux [RUB 98]

Interface de rede: Pacotes de rede chegam de forma assíncrona na interface de rede e devem ser tratados por uma pilha de protocolos existente dentro do núcleo antes de chegarem ao processo receptor. A interface entre essa pilha e o dispositivo de rede propriamente dito é feita por um *driver* de dispositivo de rede.

Nas seções a seguir serão tratados aspectos do núcleo do Linux mais importantes do ponto de vista da plataforma que será apresentada em capítulo posterior.

3.2.2 Módulos

Embora o núcleo do Linux seja monolítico (é composto por um único programa com todas as estruturas de dados, componentes funcionais e rotinas), ele oferece a carga dinâmica de componentes do sistema operacional conforme a necessidade. Esses componentes são chamados de módulos de *kernel* e podem ser ligados ao sistema operacional em qualquer ponto após o Linux ter sido carregado. A maioria dos módulos do Linux implementam controladores de dispositivos, ou, como será visto, programas de tempo real utilizando o *kernel* de tempo real *RTAI*.

Os módulos de núcleo podem tanto ser ligados explicitamente utilizando-se o comando `insmod`, quanto implicitamente através de um *deamon* chamado de *kerneld*. A carga dinâmica de módulos torna necessária apenas a existência de um núcleo mínimo e também flexibiliza-o [RUS 99].

O programa de um módulo normalmente tem uma função de inicialização (`init_module`) e uma de encerramento (`cleanup_module`). Todo o módulo roda no chamado *kernel space* (nível de núcleo do sistema operacional) e tem acesso a toda a memória da máquina e todos os dispositivos, pois o núcleo sempre roda no nível de supervisor do processador.

3.2.3 Controladores de dispositivos

No Linux os dispositivos são divididos em três categorias:

Dispositivos de caracter: Os dispositivos de caracter têm a característica de per-

mitir o acesso a um número arbitrário de bytes em cada operação de E/S [BOV 2001]. Normalmente os bytes são acessados de forma seqüencial. O controlador do dispositivo normalmente implementa as chamadas de sistema *open*, *close*, *read* e *write* [RUB 98].

Dispositivos de bloco: São capazes de transmitir um bloco de tamanho fixo a cada operação de E/S. Os blocos armazenados no dispositivo podem ser lidos em ordem aleatória [BOV 2001]. Dispositivos de bloco podem conter um sistema de arquivo. O Linux permite que a aplicação receba um byte por vez, portanto a diferença entre os dispositivos de bloco e os de caracter é vista somente dentro do *kernel* do Linux. Os dispositivos de bloco, assim como os de caracter, são acessados através de um nodo do sistema de arquivo [RUB 98].

Interface de rede: Alguns dispositivos de entrada e saída não têm arquivo de dispositivo (em outras palavras, não são acessados através de um nodo no sistema de arquivo). A classe de interfaces de rede, que são responsáveis pelas transações de rede (envio e recebimento de pacote), são exemplo de dispositivo sem arquivo.

A interface de rede manipula pacotes de dados sem ter o conhecimento do fluxo de dados que os gerou. Cada interface de rede tem um nome designado, que é único no sistema. Por exemplo, a primeira interface *ethernet* usualmente recebe o nome de `eth0`. O controlador de dispositivo de rede não precisa implementar as chamadas normalmente utilizadas para dispositivos de bloco, como *read* e *write*, e sim chamadas específicas de manipulação de pacotes. No nível de aplicação, no lugar das chamadas de sistema relacionada com o sistema de arquivo, chamadas como `socket`, `bind` e `connect` são utilizadas.

Essa classe de dispositivo é importante no contexto da presente dissertação pois um controlador de dispositivo para a comunicação normal será descrito no capítulo seguinte.

Os controladores de dispositivos podem tanto estar dentro do núcleo monolítico do sistema operacional quanto ser implementados na forma de módulos do sistema.

Sendo um módulo, um controlador de dispositivo normalmente registra suas capacidades na função `init_module`. Uma lista de ponteiros para funções é então guardada pelo núcleo. Essas funções são operações que podem ser realizadas por aquela classe de dispositivo, como por exemplo, *open*, *close*, *read*, *write*, etc. Toda vez que o *kernel* receber uma chamada relativa a um dispositivo, ele irá chamar a função registrada correspondente.

Controladores de interface de rede

Normalmente em uma rede de computadores tem-se vários protocolos trabalhando em camadas para que a comunicação seja estabelecida. Por exemplo, no caso do uso de *sockets* sobre o TCP/IP, tem-se as camadas mostradas na figura 3.3. A tarefa do controlador nesse contexto é a transmissão pela interface de rede dos pacotes IP recebidos pelo núcleo do sistema operacional. O dado a ser transmitido surgiu de uma conexão de alguma aplicação de rede e já foi transformado em pacote TCP e posteriormente IP chegando pronto para a transmissão ao controlado. Se esse

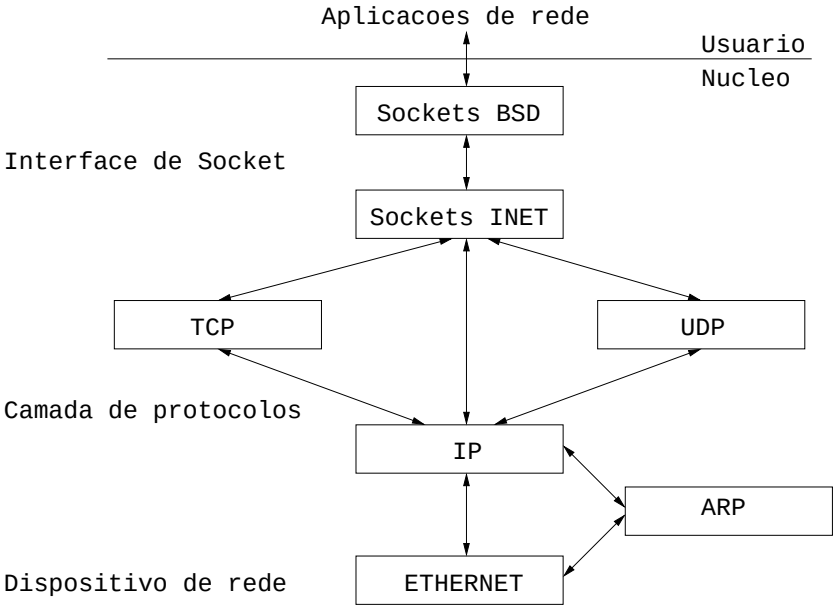

FIGURA 3.3 – Camadas de rede dentro do núcleo do Linux[RUS 99].

controlador for da classe *ethernet* um endereço físico de origem e destino junto com algumas outras informações serão acrescentados ao pacote IP formando um pacote *ethernet* pronto para ir para o meio físico. A tradução de endereço IP para MAC (endereço *ethernet*) é feita utilizando-se um protocolo conhecido como ARP.

Os parágrafos a seguir descrevem os pontos básicos de uma implementação de um controlador (*driver*) de rede para Linux.

A primeira tarefa de um controlador de rede, normalmente realizada na função de inicialização `init_module`, é o registro de quais dispositivos de rede serão gerenciados pelo controlador. Esse registro é feito pela chamada `register_netdevice` que recebe como parâmetro uma estrutura que se chama *net_ device* no *kernel* versão 2.4 e somente *device* no *kernel* 2.2. Nessa estrutura está o nome do dispositivo registrado (no caso de um controlador de placa *ethernet*, por exemplo, o nome pode ser "*eth0*"). Além disso é indicada uma função que irá inicializar o dispositivo, comumente chamada de `driver_init`.

Já a função de inicialização do dispositivo tem a tarefa de preencher uma estrutura do tipo `struct device` que contém informações relativas ao dispositivo de rede, como o mtu (*maximum transfer unit*) da rede e outros parâmetros. Além disso, ela deve associar tratadores aos ponteiros de função que o núcleo irá chamar no momento apropriado. As funções mais importantes que devem ser implementadas pelo controlador e os respectivos ponteiros registrados nessa estrutura são:

`open:` Abre a interface de rede. A interface é aberta toda a vez que o comando *ifconfig* a ativar.

`get_stats:` Chamada toda vez que o núcleo do sistema operacional deseja informações sobre a interface de rede (quando o usuário digita *ifconfig* para obter informações das interfaces, por exemplo).

`stop:` Pára a interface. Normalmente as operações executadas na função `open` devem ser revertidas.

`hard_header:` Função utilizada para fazer o cabeçalho de hardware. Em interfaces da classe *ethernet* existe uma implementação pré-definida que geralmente é utilizada.

`hard_start_xmit:` "*Hardware start transmission*". Método chamado toda a vez que o núcleo do sistema operacional deseja enviar um pacote. A função do controlador recebe o pacote como parâmetro dessa função numa estrutura chamada de `sk_buff` descrita no próximo capítulo.

Após essa estrutura ter sido preenchida, todo o evento associado à interface de rede gerará uma chamada de função dentro do controlador da interface correspondente. Por exemplo, cada pacote que deve ser enviado pela interface *eth0* gerará uma chamada da função `hard_start_xmit` dentro do controlador da interface, tendo como parâmetro o pacote que será enviado.

Uma exceção a esse sistema de *callbacks* é o recebimento de pacotes pela interface de rede que devem ser repassados para o núcleo do sistema operacional (pacotes chegando de outros nodos). A rotina de recebimento não é invocada pelo *kernel*, pois ele não tem o conhecimento da chegada de um pacote, somente a placa de rede o tem. Normalmente ela chama uma interrupção dentro do controlador, e essa interrupção irá, através de rotinas de E/S, ler o pacote e repassá-lo posteriormente para o núcleo. Esse repasse é feito através da chamada `netif_rx`.

Essa é uma visão resumida de como funciona um controlador de interface de rede para Linux. Uma implementação e maiores detalhes podem ser vistos no próximo capítulo.

3.2.4 Tabela de símbolos

Para haver o compartilhamento de variáveis e funções entre os diversos módulos e sua ligação (*binding*) dinâmica, existe no Linux uma tabela de símbolos públicos do núcleo. Todas as variáveis e funções globais de um módulo são automaticamente incorporadas na tabela de símbolos. Se isso não é desejado, primitivas especiais são disponibilizadas para o registro seletivo dos símbolos que devem ir para a tabela. O modificador `static` do `C` também pode ser utilizado para evitar a disponibilização de símbolos na tabela global.

Novos módulos podem utilizar símbolos exportados por módulos carregados anteriormente, isso gera uma pilha de módulos e uma relação de dependência entre eles.

3.2.5 Fila de tarefas

Muitas vezes, uma controlador de dispositivo ou outra função do núcleo deseja postergar a execução de algumas tarefas. O Linux fornece dois mecanismos diferentes para esse propósito: filas de tarefas e os *timers* de *kernel*. Normalmente os *timers* são utilizados para executar uma tarefa em um momento preciso no futuro, enquanto que as filas de tarefas permitem a execução posterior em momentos determinados, que serão vistos a seguir [RUB 98].

Em uma fila de tarefas, cada tarefa é representada por um ponteiro de função e um argumento. A estrutura de um elemento pode ser vista abaixo (retirada de `<linux/tqueue.h>`):

```
struct tq_struct
{
  struct tq_struct *next; /* Próxima na lista */
  int sync;               /* Deve ser  inicializado */
                          /*                com zero */
  void (*routine) (void *); /* Rotina a ser chamada */
  void *data;             /* Parâmetro */
}
```

Após ser criado um representante da tarefa, ela pode ser colocada em alguma das filas pré-existentes no sistema. As seguintes filas estão normalmente disponíveis:

`tq_scheduler:` As tarefas são executadas toda vez que o escalonador roda.

`tq_timmer:` É executada a cada *tick* do timer.

`tq_immediate:` Rodada o mais cedo possível, quando uma chamada de sistema acontece, ou quando o escalonador é chamado.

`tq_disk:` Utilizada internamente para o gerenciamento da memória virtual.

Cada uma dessas filas é uma lista encadeada de ponteiros para tarefas a serem executadas. No momento apropriado para cada fila, o núcleo do sistema operacional irá chamar a função `run_task_queue` que é responsável por rodar todas as tarefas que estão em uma fila. Um importante ponto a destacar é que no momento que o núcleo chama a rotina `run_task_queue` o contexto impõe algumas restrições sobre o que as tarefas na fila podem executar. Isso porque normalmente essas filas são executadas em resposta a alguma interrupção de software. As ações que não podem ser realizadas dentro do chamado tempo de interrupção do núcleo são:

- Acesso ao espaço de usuário. Como não existe contexto de processo, não há caminhos para o espaço de usuário associado com um processo particular
- Acesso ao ponteiro `current` (que aponta para o processo em execução). Isso porque ele está inválido durante o tempo de interrupção
- Invocação do escalonador
- Invocação de primitivas que coloquem a tarefa a dormir (`sleep_on`)
- Primitiva `Wait` de semáforo (pois pode colocar a tarefa a dormir na fila do semáforo)
- Invocação de determinadas funções dentro do núcleo, e.g. kmalloc, por utilizar semáforos no estabelecimento de seções críticas

3.2.6 Organização e alocação da memória

Devido à utilização de memória e mapeamentos de memória em diversos módulos da plataforma proposta, segue-se uma visão geral de como o núcleo do sistema Linux trata questões relativas à memória.

Esse trabalho foi desenvolvido utilizando a versão do sistema Linux para processadores Intel (ou compatíveis) 80x86. O endereçamento de memória feito pelo

Linux está intrinsicamente ligado com aspectos de hardware dos processadores da Intel. Por motivos de arquitetura do processador, existem dentro do núcleo do Linux três tipos de endereço:

Endereços Lógicos: Endereços que utilizam segmentos onde cada endereço consiste em um segmento de memória e um deslocamento dentro do segmento.

Endereços Lineares: Inteiro de 32 bits que pode ser utilizado para endereçar até 4GB de memória. Um página válida de endereços lineares (virtuais) é mapeada em endereços físicos pela MMU (*memory management unit*) do processador.

Endereço Físico: Endereço físico da memória utilizado para endereçar os circuitos integrados. Também é representado por inteiro de 32 bits.

Um endereço lógico é transformado em linear pela unidade de segmentação de memória. Já um endereço linear é traduzido para um físico pela unidade de mapeamento de memória que trabalha com páginas de 4Kb (ou também, nos processadores mais modernos, 4Mb) [BOV 2001].

A segmentação no Linux é utilizada de forma muito limitada. O sistema operacional só a utiliza onde o processador Intel impõe necessidade.

Uma característica interessante do núcleo do sistema operacional Linux é a organização do endereçamento da memória linear. O espaço de endereçamento linear usualmente é muito maior que a memória física real instalada na máquina. Então, o Linux reserva $\frac{3}{4}$ do total de endereçamento (que corresponde a 3GB) como área de endereçamento para ser utilizada pelos diversos processos de usuário. O 1GB restante pode ser acessado somente pelo núcleo. O limite entre essas duas áreas é dado pela constante `PAGE_OFFSET`. Após esse endereço o núcleo mapeia toda a memória física do computador (incluindo o próprio código do núcleo).

Como a memória física inteira é mapeada após `PAGE_OFFSET`, para fazer um acesso direto à memória de vídeo, por exemplo, um *driver* do núcleo precisa simplesmente somar o endereço original da memória de vídeo (entre os 640k e o primeiro 1M nas arquiteturas PC) com o símbolo `PAGE_OFFSET` para ter o endereço linear que deve ser acessado. Após o mapeamento da memória do PC, um espaço livre de 8 Mb é reservado e temos aí endereços livres para serem utilizados pela função `vmalloc` e `ioremap` que serão explicadas a seguir. Esses endereços livres iniciam em `VMALLOC_START`.

Alocação dinâmica de memória

O núcleo do sistema operacional Linux fornece diversas rotinas de alocação dinâmica de memória para a utilização dentro do núcleo e também para retornarem áreas que podem ser utilizadas por processos de usuário. Algumas delas são utilizadas para a alocação de memória fisicamente contígua, que é apropriada para operações de DMA, visto que o controlador só consegue acessar a memória pelo endereço físico (e não linear). Outras alocam blocos contíguos no espaço de endereçamento linear, mas espalhados na memória física do computador. As principais rotinas de alocação de memória dinâmica são:

`kmalloc`**:** Aloca blocos de memória fisicamente contíguos. Note que nem sempre a alocação é imediatamente bem sucedida. Se o número de páginas disponíveis no *pool* que o Linux disponibiliza estiver abaixo da marca `min_free_pages`

o processo que requisitou é posto para dormir aguardando por uma página. É importante ressaltar que o Linux gerencia a memória física da máquina, sendo disponível apenas em tamanhos de página (usualmente 4Kb). Como `kmalloc` somente retorna páginas fisicamente contíguas, o núcleo do sistema operacional gerencia listas de conjuntos de páginas contíguas apropriadas para serem retornadas por `kmalloc` e também para serem utilizadas no DMA. Essas filas são organizadas em tamanhos múltiplos de potência de dois para serem utilizados em um algoritmo de alocação conhecido por *buddy system* que é utilizado para tratar o problema da fragmentação externa. Assim, o maior desperdício possível é de 50%. Para amenizar essa questão, que é conhecida como fragmentação interna, uma outra técnica de alocação conhecida como *Slab* é utilizada.

`get_free_page` **e assemelhadas:** Para alocar pedaços grandes de memória, é melhor a utilização de funções orientadas a páginas. `get_free_page` retorna uma página. Já a chamada `__get_free_pages` pode alocar memória contígua em números de página múltiplos de potências de dois pela utilização do algoritmo *buddy system*. O sucesso da alocação depende do número de páginas no pool de páginas (contíguas) livres.

`vmalloc:` Aloca memória contígua no espaço virtual de endereçamento, enquanto que fisicamente a memória pode estar espalhada por páginas dispersas. Memórias alocadas por essa chamada são visíveis somente dentro do núcleo pois ficam no seu segmento privado de memória (endereço acima do delimitador `PAGE_OFFSET` que normalmente fica no quarto gigabyte do endereçamento linear do PC).

A chamada `vmalloc` é responsável pela alocação dinâmica de memória fisicamente não contígua e arranja essa memória de forma contígua nessa área superior de memória. Já a `ioremap` é utilizada freqüêntemente para o mapeamento de endereços físicos de memória que não estão dentro do intervalo da memória RAM do sistema em endereços virtuais. Isso é necessário pois placas PCI, como os adaptadores *SCI* utilizados na presente arquitetura, utilizam a técnica de E/S mapeados em memória.

3.3 Linux RTAI (*Real-Time Application Interface*)

O *RTAI* é uma extensão do núcleo do sistema operacional Linux para fazê-lo totalmente preemptável. O núcleo padrão do Linux sem essa extensão sofre de deficiências no suporte de serviços de tempo real.

Para atingir um comportamento temporalmente correto, são necessárias algumas modificações no fonte do núcleo, i.e. no gerenciamento de interrupções e nas políticas de escalonamento.

O *RTAI* oferece os mesmos serviços básicos do núcleo adicionando funcionalidades de um sistema de tempo real industrial.

O principal componente do *RTAI* é o gerenciador das interrupções: todas as interrupções de hardware são direcionadas para o *RTHAL (Real-time hardware abstraction layer)* que é o módulo do *RTAI* responsável pelo redirecionamento das interrupções. Quando uma interrupção chega, se ela for de interesse do núcleo

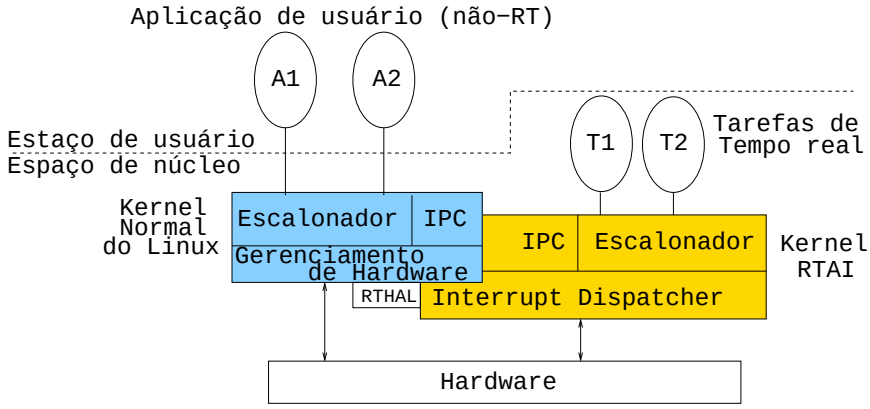

FIGURA 3.4 – Arquitetura do *RTAI* [MOU 2000]

padrão, o *HAL* é responsável pelo redirecionamento dessa interrupção para o núcleo do Linux somente em momentos onde não existir nenhuma tarefa de tempo real para ser executada [MAN 2000].

O *RTAI* considera a totalidade do Linux como uma tarefa de baixa prioridade rodando quando não existe nenhuma atividade de tempo real. Isso faz com que o *RTAI* e as tarefas de tempo real nunca tenham que esperar que o Linux libere algum recurso [MAN 2000a].

Para evitar que o Linux desabilite as interrupções bloqueando uma tarefa de tempo real, todas as ocorrências de `cli`, `sti` e `iret` são substituídas por macros de emulação fornecidas pelo *RTAI*. A camada *RTHAL* só redireciona as interrupções para o Linux quando esse está com as interrupções habilitadas.

O núcleo do *RTAI* foi desenvolvido na forma de módulos do Linux, assim ele roda no espaço de endereçamento do núcleo. Toda vez que os módulos do *RTAI* são removidos, o Linux volta a funcionar do modo convencional.

As tarefas de tempo real também rodam dentro do espaço de núcleo. Assim, elas podem acessar diretamente o hardware e o tempo de troca de contexto entre as tarefas é minimizado. Uma grande desvantagem dessa abordagem é a vulnerabilidade do sistema: bugs dentro de tarefas de tempo real podem levar a uma pane em todo o sistema.

A arquitetura do *RTAI* pode ser vista na figura 3.4.

Na figura pode-se ver as tarefas de tempo real rodando no espaço de núcleo. O *RTAI*, como pode ser visto, é formado por um gerenciador de interrupções, um escalonador com algoritmos baseados em prioridade e primitivas de comunicação interprocesso que são utilizadas para comunicação entre processos RT ou entre processos RT e não-RT, todos localizados em uma máquina local.

A escolha do *RTAI* como suporte à plataforma a ser desenvolvida se deu pelos seguintes motivos:

- A licença que cobre o *RTAI* é GNU, sendo assim software livre. Um dos requisitos do projeto é a utilização de uma infraestrutura baseada em software livre.

- O *RTAI* oferece uma grande quantidade de serviços que podem ser utilizados pela plataforma de comunicação.

- O *RTAI* está em constante melhora, possibilitando que no futuro se utilize novas versões que acrescentam funcionalidades.

- Utilizando o núcleo de tempo real *RTAI* foi possível o desenvolvimento da plataforma de comunicação com algoritmos onde o determinismo temporal é crítico.

3.3.1 Funcionalidades do *RTAI*

O *RTAI* fornece um conjunto de primitivas nas seguintes áreas:

Gerenciamento de tarefas: Permite criar, habilitar, destruir tarefas de tempo real periódicas ou não. Cada tarefa tem uma prioridade dentro do sistema. Na plataforma que será descrita, todas as tarefas foram criadas por primitivas desse grupo.

Funções de tempo: Utilizadas para o controle temporal das tarefas de tempo real. Entre elas, é possível encontrar primitivas que permitem à tarefa dormir por um período, acessar o relógio de tempo real, etc.

Funções de semáforos: Com as chamadas disponibilizadas é possível a utilização de três tipos de semáforos: binários, contadores e de recursos[BIA 2001]. Na plataforma desenvolvida somente semáforos contadores são utilizados.

Gerenciamento de mensagens, *RPC* e *Mailbox*: Utilizados para comunicação entre tarefas.

Funções diversas do *RTAI*: Permitem a manipulação de interrupções e outras funcionalidades associadas ao núcleo do *RTAI*.

Funções de compartilhamento de memória: Utilizadas para a criação de blocos compartilhados de memória entre as tarefas. As rotinas de avaliação da plataforma fazem uso dessas funções para disponibilizar os dados coletados no nível de usuário, como será visto no capítulo de avaliação da plataforma.

Funções do LXRT: Permite a criação de tarefas de tempo real no nível de usuário utilizando um "fantasma" no nível de núcleo que é o real responsável pela chamada das primitivas de tempo real do *RTAI*.

Gerenciamento de FIFO: Permitem o estabelecimento de filas de comunicação entre tarefas de tempo real e tarefas no modo de usuário do Linux.

3.4 SCI (*Scalable Coherent Interface*)

3.4.1 Conexão do barramento de E/S

Os adaptadores de rede *SCI* baseiam-se na tecnologia de conexão de barramento de entrada e saída. Essa conexão de barramentos permite à CPU de um nodo A acessar a memória de outro nodo B utilizando-se de instruções de leitura/escrita em memória. Partes da área de E/S (entrada/saída) do nodo A age como uma janela em parte da memória principal do nodo B. Quando a CPU A executa uma

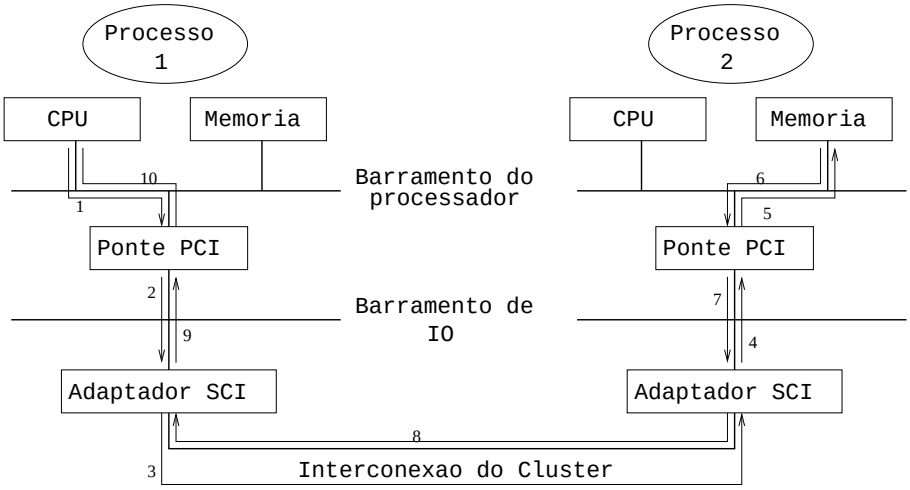

FIGURA 3.5 – Comunicação entre dois processos em dois nodos distintos pela rede *SCI*

instrução de leitura/escrita no barramento de memória do processor dentro da janela de endereçamento mencionada, a requisição é roteada do barramento de memória (caminho 1 na figura 3.5 através da ponte para o barramento de E/S (caminho 2). O adaptador de rede no barramento de E/S do nodo A irá tratar essa requisição repassando-a (através da conexão de entre adaptadores) para o adaptador no nodo B (caminho 3). O adaptador do nodo B então irá repassar essa requisição através da ponte de E/S (caminho 4) para o barramento de memória (caminho 5). A memória então irá responder através da ponte (caminho 6) para o adaptador do nodo B (caminho 7). A resposta então é enviada para o adaptador do nodo A (caminho 8), que a encaminha para a CPU através da ponte de E/S (caminhos 9 e 10). Com respeito a essa conexão, o nodo A é chamado de cliente da memória remota do nodo B.

Essa requisição e sua respectiva resposta acontecem de forma inteiramente transparente para o software. Deve-se ter em vista que devido à inexistência de *cache* nessas áreas de memória, a performance é bastante reduzida para operações remotas. A grande dificuldade técnica de implementação de cache nesse tipo de tecnologia provém do fato de um nodo não poder monitorar o barramento de memória do outro para detectar alterações em posições de memória compartilhada. Do ponto de vista puramente técnico, uma maior eficiência seria obtida ao ligar-se, em vez do barramento de E/S, diretamente o barramento de memória de máquinas distintas. O problema, é que devido à existência de vários tipos de barramento de memória, essa solução se torna impraticável [RYA 97].

Um observação importante é que embora um nodo não possa utilizar cache em partes de memória importada de outro nodo, a sua própria memória, mesmo sendo oferecida a outros nodos, pode ter cache.

3.4.2 Primitivas de sincronização

A memória compartilhada que é provida pelos adaptadores de rede *SCI* oferece um meio para comunicação entre máquinas no cluster. Mecanismos de sincronização podem ser implementados com uma pequena modificação na semântica da operação de leitura/escrita na memória compartilhada:

Leitura destrutiva: Após leitura de uma posição de memória, seu valor é alterado atomicamente

Escrita com interrupção: Modifica uma posição de memória compartilhada e após dispara uma interrupção. Isso permite uma *thread* dormir até que determinada condição é sinalizada pelo nodo remoto

Junto com as operações normais de leitura/escrita temos uma base completa de sincronização e comunicação. A grande desvantagem dessas operações de sincronização é a sua baixa velocidade. Utilizando-se adaptadores típicos *SCI* PCI, temos latência de escrita de memória remota na ordem de $5\mu s$ e leitura $15\mu s$ enquanto que uma interrupção tem latência de $80\mu s$ e a leitura destrutiva e latência consideravelmente maior que a leitura comum.

3.4.3 Transferência de dados

Devido à natureza de um cluster conectado através de adaptadores *SCI*, dois tipos de transferência de dados podem ocorrer:

Leitura/escrita em memória: Utiliza diretamente as instruções de leitura/escrita em memória (método já exemplificado na seção anterior)

***Direct Memory Access*:** Através de um hardware especial um bloco de memória local pode ser copiado para uma janela de memória compartilhada sem a intervenção direta do processador (que é responsável apenas pela inicialização). A programação do DMA normalmente é feita com auxílio de sistema operacional e a transferência normalmente deve ser feita entre áreas de memória física que não podem ser transferidas para o disco pelo gerenciamento de memória virtual.

3.4.4 Hardware

Topologia da rede

Fisicamente, a norma *SCI* diz que os nodos devem ser conectados através de *links* unidirecionais ponto a ponto [HEL 99]. Esses *links* podem ser implementados de forma paralela (para pequenas distâncias) ou serial (grandes distâncias). A maioria das implementações utiliza *links* paralelos em pequenas distâncias. O fabricante dos adaptadores utilizados para a avaliação da plataforma desenvolvida nessa dissertação chama-se *SCALI* e utiliza *links* paralelos.

Teoricamente, até 64 mil nodos podem ser conectados pelo *SCI*. A topologia de rede utilizada pode ser das mais diversas, sendo algumas mostradas na figura 3.6. Como cada adaptador é composto por um par de *links*, configurações como *Torus 2D* são possíveis.

Note que as requisições de leitura/escrita em memória são, de forma transparente, convertidas em pacotes pelo hardware e esses pacotes transmitidos pela rede.

Estrutura da interface de um nodo

Diferentes e complexas tarefas devem ser realizadas por uma interface de rede de um nodo. Por exemplo, deve ser possível inserir pacotes no *link* de saída enquanto

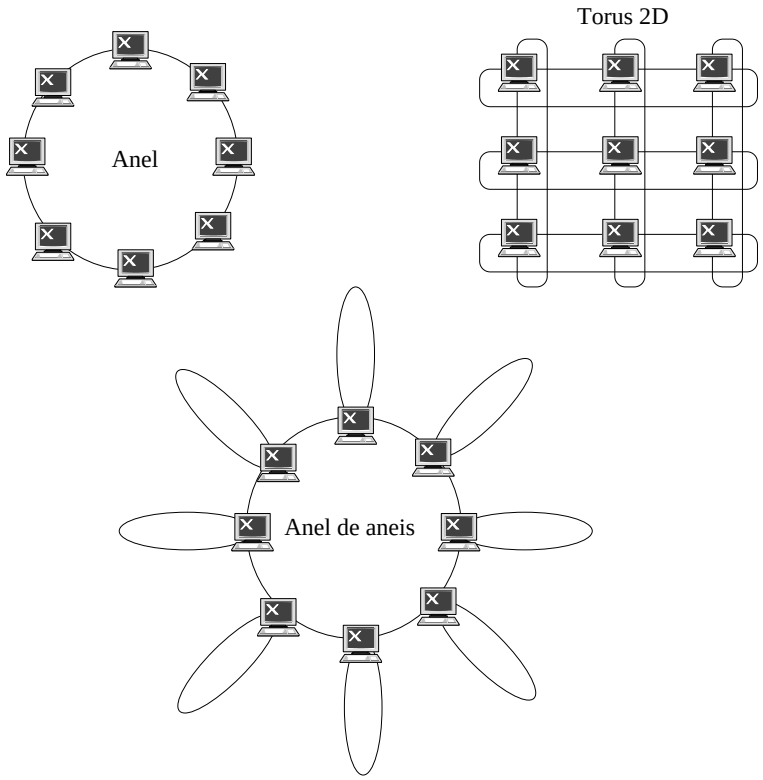

FIGURA 3.6 – Algumas topologias de rede realizáveis com a tecnologia *SCI*

que concorrentemente são retirados pacotes que são endereçados para o nodo local ou simplesmente é armazenado um pacote destinado a outro nodo e re-inserido posteriormente. Além disso, tarefas como a detecção de erro e manutenção da rede devem ser realizadas.

A interface padrão proposta para lidar com essa complexidade pode ser vista na figura 3.7.

A interface de rede contém um *link* de entrada e outro de saída. No de entrada, símbolos chegam de maneira assíncrona com seu próprio sinal de clock. O primeiro estágio da interface, chamado de buffer elástico (*elastic buffer*) é responsável por sincronizar o fluxo de dados que entra com o domínio de clock local. O resto da interface trabalha de modo totalmente síncrono. Os pacotes que chegam pela conexão de entrada são analisados pelo *stripper* e colocados em uma das diversas filas de entrada para serem processados pela lógica de aplicação (*application logic*). Essa fila de entrada é composta por um bloco de requisição (e.g. processador) e por outro de respostas (e.g. memória principal). No caso de um pacote destinado a outros nodos, ele é transferido através do *link* de saída.

Se, durante a chegada de um pacote, o nodo local está inserindo um pacote na conexão de saída, o pacote que chegou deve ser colocado em um *buffer* esperando o redirecionamento. Esse *buffer* é chamado de *bypass FIFO*. Um nodo pode somente colocar um pacote próprio na conexão de saída se ele detectar que o *bypass FIFO* está vazio[HEL 99].

É importante ressaltar que essa é uma das características dos adaptadores *SCI* que impede sua direta aplicação para a transmissão de tráfego em tempo real. Para exemplificar, suponha uma rede com três nodos conforme pode ser visto na figura

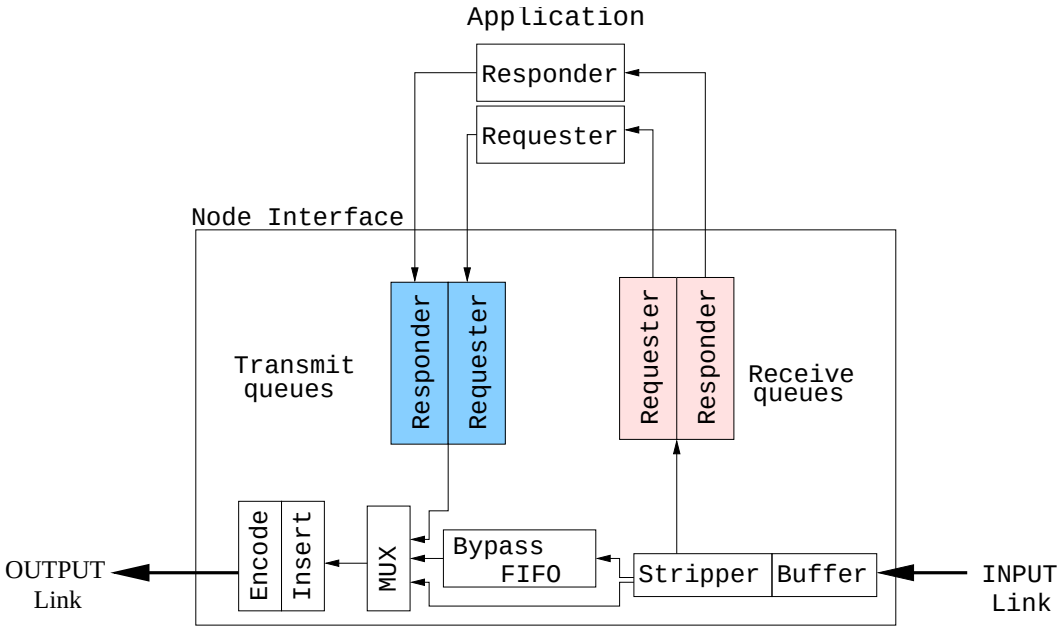

FIGURA 3.7 – Estrutura padrão de uma interface *SCI*

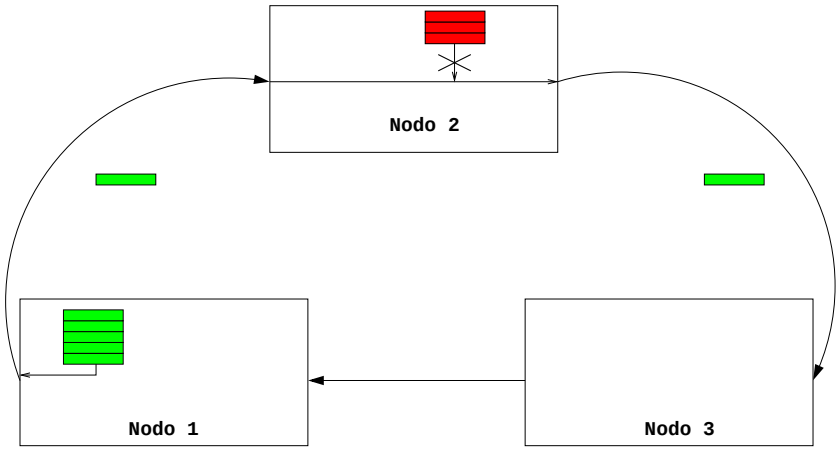

FIGURA 3.8 – Problema da contenção na transmissão de pacotes

3.8. Suponha que o nodo 1 deseja enviar grande quantidade de pacotes para o nodo 3. No meio dessa transmissão, o nodo 2 deseja mandar pacotes para o nodo 3. Como a conexão de entrada do nodo está sempre recebendo pacotes, a fila de *bypass* estará cheia. Assim, os pacotes do nodo 2 terão o acesso ao meio físico bloqueado, havendo portanto um cenário de contenção.

Se a comunicação entre os nodos 1 e 3 for de baixa prioridade, enquanto que os pacotes que o nodo 2 deseja mandar tenham alta prioridade, configura-se um cenário de inversão de prioridade.

A falta de controle sobre a prioridade de comunicação é um empecílho para a utilização direta do *SCI* para o tráfego de tempo real. Em um ambiente de comunicação desse tipo, é necessária uma diferenciação entre as mensagens de tempo real e as comuns. As primeiras devem ter uma prioridade de envio sobre o segundo tipo.

Endereçamento

O *SCI* utiliza endereços de 64 bits divididos em duas partes: 16 bits mais significativos especificam um nodo dentro da rede (daí deriva-se o limite teórico de 64k nodos) enquanto que o restante é utilizado para endereçamento de posições de memória dentro de blocos exportados por um nodo.

Princípio de funcionamento

Nessa seção será abordado o principio de funcionamento de um cluster *SCI*.

O hardware *SCI* pode ser utilizado para conectar barramentos de E/S de nodos separados. Como foi visto, uma rede *SCI* pode ser vista como um adaptador entre as máquinas hospedeiras e um espaço global de endereçamento compartilhado por todas máquinas conectadas ao cluster.

Durante o processamento, um processo deseja acessar um posição de memória dada por um endereço linear virtual. Esse endereço é traduzido pela MMU (*Memory Management Unit*, ou unidade gerenciadora de memória) para um endereço físico. Esse endereço é utilizado para acessar ou a memória local ou o adaptador local (através de entrada/saída mapeada em memória). Uma região de endereços físicos é associada com cada adaptador. Leituras e escritas nessa região são servidas através de sua tradução em pacotes de solicitação que são enviados a um nodo remoto. Essa região age como uma janela na memória do nodo remoto. Endereços de E/S são utilizados pelos adaptadores para acessar a memória local em resposta a requisições de leitura/escrita vinda de adaptadores remotos.

O conceito de mestre/escravo pode ser utilizado para se caracterizar uma comunicação *SCI*.

- O escravo gera requisições de escrita/leitura na sua CPU local que são traduzidas para pacotes de requisições pelo adaptador *SCI* e colocadas no canal de comunicação. Após isso, se for o caso, o nodo espera pela resposta para alimentar a CPU e continuar o processamento

- O mestre recebe requisições e repassa, através do canal de E/S que acaba no barramento de memória. A resposta é mandada de volta para o nodo que originou a requisição.

  A tradução de endereço é feita utilizando-se uma tabela conhecida como *ATT* (*Address Translation Table*, ou tabela de tradução de endereços.

3.4.5 Características da rede *SCI*

Analisando-se uma rede *SCI* sob a ótica das propriedades de uma rede abstrata (ver tabela 2.1 no capítulo de revisão bibliográfica), temos que a propriedade An1 não é conseguida facilmente. O broadcast numa rede *SCI* só pode ser conseguido se um nodo n disponibiliza uma área de memória conectada por todos os outros. Daí os nodos remotos devem fazer a leitura nessa memória, para todos terem acesso ao mesmo dado (característica do *broadcast*). Infelizmente isso no nível físico não corresponde a um verdadeiro *broadcast*, pois diversas mensagens de requerimento da posição e suas respectivas respostas são geradas. Sendo assim, o *SCI* não implementa o conceito físico de *broadcast*.

Já a propriedade An2 é cumprida pela verificação de CRC feita pelos nodos *SCI* com auxílio de limites de tempo. É importante ressaltar que o número de

retransmissões é limitado.

Quanto à ordenação dos pacotes (propriedade An3), o adaptador *SCI* não a garante justamente por causa do controle de erro. Se um determinado pacote contém erros, ele será retransmitido, mas nesse momento outro pacote posterior pode ter sido recebido. Como essa característica é importante para a comunicação RT proposta, uma solução para esse problema será mostrada no capítulo que descreve a arquitetura.

Para termos um atraso de transmissão limitado (característica An6) e determinístico (requisito para transmissões de tempo real), precisa-se de uma uma disciplina de acesso ao meio físico. Essa disciplina é proposta como parte da plataforma do presente trabalho.

3.4.6 Software

O suporte de software para os adaptadores *SCI* oferece serviços tanto para aplicações que rodam no modo de usuário, quanto para módulos rodando dentro do núcleo do sistema operacional. Como os clientes que rodam dentro do núcleo são confiáveis enquanto que as aplicações a nível de usuário não o são, são oferecidas duas interfaces diferentes. Nessa seção somente os serviços oferecidos para módulos de núcleo serão descritos.

O componente de software que será apresentado chama-se *Interconnect Manager* (gerenciador de interconexão), que também é conhecido como *ICM*. Módulos de núcleo podem ser ligados com o módulo *ICM* dinamicamente durante sua carga.

Gerenciamento de memória local

O gerenciador de memória local é responsável por alocar a memória que será compartilhada com os outros computadores através dos adaptadores *SCI*. Um bloco de memória é adequado para o compartilhamento se ele está na memória física todo o tempo, não sendo substituído pelo mecanismo de gerenciamento de memória virtual. Essa restrição é necessária porque as páginas serão acessadas diretamente por máquinas remotas através do barramento de E/S do adaptador local.

Esse bloco de memória pode ser dividido em sub-blocos que são contíguos no espaço de endereçamento de E/S e, por conseqüência, no endereçamento global da rede. Devido à lógica de transformação em pacotes nos escravos, esses sub-blocos devem ter tamanho múltiplo de 64 bytes. Se o hardware dispor de uma MMU de E/S e for capaz de apresentar um bloco de memória em somente um sub-bloco, não existe limite no tamanho do sub-bloco. Senão, ele está limitado ao tamanho de uma página (4096 bytes, no caso da arquitetura Intel).

Uma vez que o bloco de memória que será exportado está alocado, ele deve ser disponibilizado para as outras máquinas da rede. No momento da disponibilização do bloco, é necessário nomeá-lo para a posterior conexão. Um bloco exportado é identificado unicamente por uma tripla de números. O primeiro número é o *module ID* e foi criado para representar o fabricante do programa que está disponibilizando o bloco de memória. Já o *chunk ID* é usado para diferenciar entre os vários blocos alocados pelo mesmo fabricante. Finalmente o *node ID* liga um determinado bloco a um nodo dentro do cluster [HEL 99].

Gerenciamento da memória remota

O gerenciador da memória remota é utilizado para controlar as conexões com

blocos de memória remota. Conectar com um bloco remoto de memória significa estabelecer uma relação cliente/servidor. Depois de estabelecida essa conexão, normalmente ela é mapeada em um endereço físico da memória local que pode finalmente ser mapeado em um endereço linear acessível aos programas. A biblioteca *ICM* somente mapeia um bloco remoto em um endereço físico, o outro mapeamento (para um endereço linear) deve ser feito com alguma chamada do sistema operacional.

Todo o suporte de software descrito nessa seção é utilizado somente no gerenciamento da memória local e remota. Após o estabelecimento dos blocos e sua respectiva conexão, o módulo *ICM* normalmente não é mais necessário, pois deste ponto em diante a comunicação é realizada por leitura/escrita em memória distribuída compartilhada.

3.4.7 Extensão de tempo real (*SCI/RT*)

Como foi visto, o padrão *SCI* provê uma rede com topologia física baseada em anel e operação lógica de barramento. A alta largura de banda junto com latências baixas tornam a plataforma atrativa para aplicações de tempo real distribuídas. O grande problema é que o padrão original do *SCI* não oferece suporte para o tráfego de mensagens com restrições temporais. Isso pode ser resolvido de duas maneiras: através de uma mudança física no hardware do adaptador de rede *SCI* para permitir uma priorização do tráfego de tempo real sobre o normal ou através de uma camada de software que discipline o acesso à rede de forma que a contenção de pacotes de alta prioridade não ocorra.

A primeira solução foi proposta pelo grupo de trabalho IEEE 1596.6 *SCI/RT*. Nela a priorização de mensagens é incluída no hardware do *SCI*. A segunda é proposta nessa dissertação na forma de uma disciplina no acesso ao meio feita pela camada MAC da plataforma.

A plataforma de software proposta soluciona o problema da contenção do hardware original do *SCI* provendo uma camada de acesso ao meio determinística. Mas, ao contrário da proposta IEEE 1596.6, apresenta também uma pilha de protocolos completa, inclusive com uma API que permite o estabelecimento de canais de tempo real para a troca de mensagens sobre o *SCI*. Assim, é possível utilizar o paradigma da troca de mensagens sobre uma rede baseada em memória compartilhada.

A proposta IEEE 1596.6 será apresentada nessa seção por ser uma extensão do *SCI* padrão, embora não seja utilizada na plataforma de comunicação. Essa proposta provê solução de hardware para alguns problemas atacados com camadas de software na arquitetura.

O *SCI/RT* (*Scalable Coherent Interface for Real-Time Applications*) é uma extensão do *SCI* original com o objetivo de mininimizar o problema da inversão de prioridade que foi visto na seção 3.4.4. Para isso foram propostos dois protocolos diferentes. Todos eles utilizam FIFO com prioridades na entrada e na saída de cada interface *SCI*. Com prioridades é possível, por exemplo, diferenciar pacotes com limites temporais do tráfego comum ou mesmo aplicar algum algoritmo de escalonamento baseado em prioridades.

Fila Preemptiva com Prioridades

A primeira solução proposta baseia-se na implementação de uma fila preemptiva com prioridades no lugar da fila de *bypass*. A estrutura do adaptador *SCI* dessa solução é mostrada na figura 3.9.

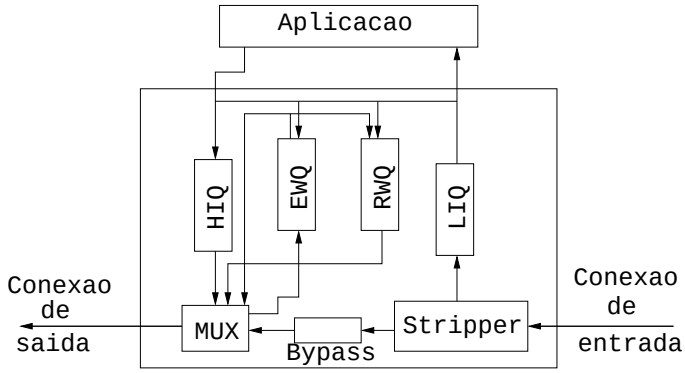

FIGURA 3.9 – Estrutura da interface *SCI/RT* para a solução Fila Preemptiva com Prioridades

A interface é similar ao modelo convencional do *SCI*. As filas *Host Input Queue* (HIQ), *Link Input Queue* (LIQ) e a fila de *bypass* são filas preemptivas com prioridades enquanto que a *Echo Waiting Queue* (EWQ) e a *Response Waiting Queue* são filas de conteúdo endereçável [TOD 2000].

A inversão de prioridade causada pela contenção na colocação de pacotes de alta prioridade do nodo por pacotes de baixa prioridade que estão na fila de *bypass* é evitada pelo fato da fila de *bypass* ser preemptável. Se não existir espaço na fila, o nodo pode tentar obter espaço retirando algum pacote de prioridade mais baixa. Os pacotes preemptados irão se converter em um sinal (eco) de ocupado, indicando que o emissor deve retransmití-los. Além disso, o pacote de maior prioridade é escolhido entre a fila de saída do nodo e a fila *bypass* para ser colocado na conexão de saída.

Uma característica significativa é a de que a fila é ordenada pela prioridade dos pacotes: pacotes com maior prioridade serão enviados antes.

O maior problema dessa solução é a geração de uma grande quantidade de tentativas de envio de pacotes de baixa prioridade.

TRAIN

O protocolo *TRAIN* resolve o problema de inversão de prioridade utilizando um árbitro global de prioridades no lugar de deixar cada nodo gerenciar as prioridades dentro das suas filas internas. A técnica é orientada à topologia de anel. Todos os nodos arbitram conjuntamente o acesso à rede através de uma estrutura de dados que é transmitida através de todo o anel. Somente os nodos com pacotes de mais alta prioridade têm direito de colocar seus dados na rede.

Existem dois modos de operação: o modo de envio imediato e o modo de arbitragem de envio. Uma mensagem especial chamada *LocalMotive* gira no anel coletando a quantidade de dados que os nodos desejam transmitir. Se essa quantidade é menor que a largura de banda da rede, o modo de envio imediato de pacotes é utilizado.

Caso contrário, o modo de arbítragem é utilizado. O pacote especial percorre a rede coletando demanda de tráfego e só as demandas com prioridades mais altas recebem o direito de acesso.

O maior problema dessa técnica é o aumento da latência de transmissão.

Avaliação

O *SCI/RT* baseia-se em uma mudança na especificação do hardware dos adaptadores. Com a plataforma de software proposta nesse trabalho, a base instalada de adaptadores *SCI* comuns pode ser utilizada. Contudo, ela foi projetada para aplicações multimídia onde os requisitos temporais não são tão importantes quanto em tempo real inflexível.

Enquanto que o *SCI/RT* utiliza técnicas baseadas em preempção e arbítragem por prioridade, a plataforma de software utiliza escalonamento do meio físico controlado pela progressão temporal, como será mostrado adiante.

Além disso, a plataforma de software provê muito mais funcionalidade do que simples acesso à rede sem contenção. Ela disponibiliza uma API para o uso do paradigma de troca de mensagem entre as aplicações, além de prover um controlador de dispositivo que permite ao sistema operacional enviar pacotes *IP* de baixa prioridade. Nenhuma dessas funcionalidades está prevista no *SCI/RT*, que é na verdade só uma plataforma de hardware mais adequada para que camadas de software façam a transmissão de tráfego de tempo real.

A própria plataforma apresentada nessa dissertação poderia ser adaptada para utilizar o *SCI/RT* como meio físico para o transporte dos pacotes o que permitiria um escalonamento de mensagens baseado em prioridade em vez do acesso ao meio físico baseado em tempo utilizado, o que aumentaria o aproveitamento da rede em ambientes dinâmicos.

É importante salientar que embora essas técnicas tenham sido desenvolvidas, nenhum fabricante de placas *SCI/RT* existe até o momento [LAN 2000].

4 Plataforma de comunicação RTC

4.1 Introdução

O presente capítulo apresenta o projeto de uma plataforma de software que tem como objetivo prover comunicação de tempo real sobre a infra-estrutura apresentada no capítulo 3. Essa plataforma chama-se **RTC** (*Real Time Communication*). Como foi visto, a simples presença de adaptadores *SCI* comuns com o devido suporte de seu controlador não é suficiente para o estabelecimento de uma comunicação dita de tempo real. Sendo assim, a arquitetura de software proposta tem como objetivo o gerenciamento das mensagens e do hardware de comunicação de forma a disciplinar o envio dos dados e torná-los previsíveis temporalmente, além de prover uma disciplina que priorize mensagens que tenham limites temporais.

Inicialmente será apresentado o modelo de comunicação que foi utilizado na plataforma. Após, uma visão global da plataforma proposta com o inter-relacionamento entre os diversos módulos que a compõem é mostrado. Posteriormente os detalhes de cada módulo da arquitetura são descritos.

4.2 Modelo de comunicação

4.2.1 Introdução

O modelo de comunicação utilizado na plataforma desenvolvida é o de troca de mensagens. Cada nodo do cluster contém um número arbitrário de processos rodando. Estes podem ser processos de usuário se comunicando através de troca de mensagens utilizando *sockets* Unix ou bibliotecas como o *MPI* ou *VPM*. Também existem processos de tempo real sendo executados sobre gerência do *RTAI* com necessidades de comunicação com outros processos do mesmo tipo na própria máquina ou em nodos remotos. Podemos ver esse cenário na figura 4.1.

Na figura pode-se ver os dois tipos de comunicação possíveis. No primeiro tipo, uma tarefa comum escreve em algum *socket* aberto entre os processos A e B. Esse fluxo de caracteres é transformado pelo núcleo do sistema operacional em pacotes IP. Esses pacotes são passados para a plataforma de comunicação que as transmite através do *SCI* para o nodo destino. No nodo destino, o pacote é lido pela plataforma de comunicação e novamente entregue ao núcleo do Linux para posterior entrega no processo destino.

Já a comunicação do segundo tipo, com restrições temporais, acontece de forma semelhante, mas sem a interferência do núcleo do Linux. Um processo C de tempo real deseja enviar uma mensagem para o D. Para isso, ele utiliza diretamente a API de comunicação que é provida pela plataforma desenvolvida. Cada mensagem deve estar associada a um canal de comunicação que será descrito a seguir. A plataforma então trata essa mensagem de maneira apropriada, colocando um cabeçalho com as informações necessárias para a transmissão assim como relativas ao canal ao qual a mensagem pertence. A mensagem com essas informações adicionais é então chamada de pacote. No momento apropriado, o pacote é transmitido para o nodo destino pelo hardware *SCI*. Já no destino, a mensagem é desempacotada e entregue ao processo receptor.

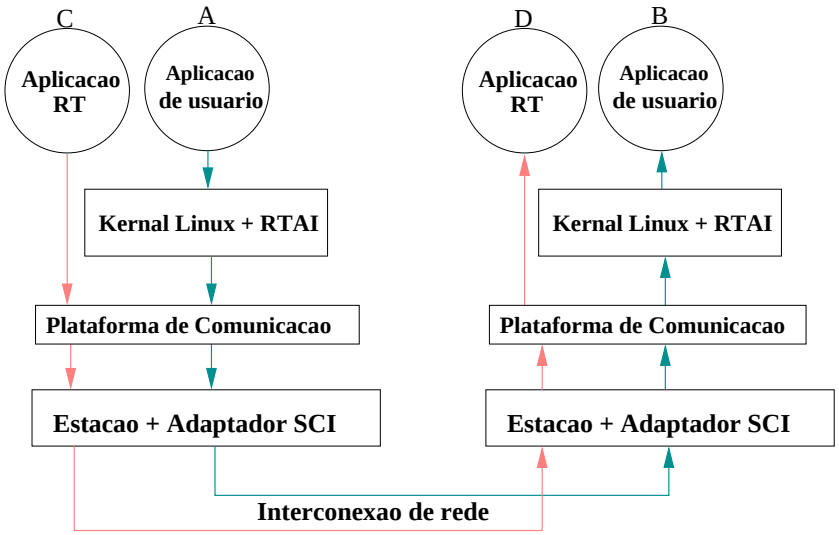

FIGURA 4.1 – Comunicação entre tarefas comuns e de tempo real

4.2.2 Canais de comunicação

A decisão de utilizar-se canais baseia-se nas características comuns da comunicação em tempo real. Como deseja-se garantias temporais nas mensagens enviadas, é mais apropriado a utilização de serviço orientado a conexão. Para poder haver garantias na transmissão de uma mensagem, o sistema necessita de informações sobre as outras fontes de mensagens que podem competir por recursos com a mensagem atual. Então é necessário fornecer um mecanismo de descrição das características da comunicação para que os recursos possam ser reservados. O mecanismo utilizado nesse trabalho é conhecido como canal de tempo real ou simplesmente canal [KAN 93]. O canal pode ser definido como uma conexão unidirecional onde características como largura de banda, atraso máximo e *jitter* máximo podem ser especificados. De acordo com essa especificação o sistema pode fazer a reserva de banda adequada.

Os objetivos da abstração de canais de comunicação são:

- Controle da quantidade de comunicação de tempo real que é permitida
- Gerenciamento da banda total oferecida pelos adaptadores *SCI*
- Reserva de recursos que possibilita a garantia das restrições temporais das comunicações

Podemos comparar a abstração de canal utilizada pela plataforma com a categoria de serviço CBR (*constant bit rate*) apresentada pelas redes ATM. Eles podem ser facilmente utilizados para a classe de tráfego multimídia, pois esse tráfego tem característica de ser um fluxo contínuo de informações com limites temporais de transmissão, pois, por exemplo, se um determinado quadro de uma seqüência de imagens não chegar no momento adequado ter-se-á uma interrupção momentânea.

Os canais são caracterizados por:

Nodo de origem: Os canais são unidirecionais, portanto comportam apenas um nodo de origem

Nodo de destino: Pelo motivo descrito acima, somente um destino é também possível

Identificador do canal: Utilizado para o gerenciamento dos diversos canais

Largura de banda: Quanto da banda disponível será reservada para um determinado canal

Menor e maior mensagens possíveis: Esses dados são utilizados para o cálculo de overhead do sistema que será ilustrado posteriormente

Existem duas fases ligadas ao uso de canais:

1. Estabelecimento do canal com os devidos parâmetros
2. Escalonamento das mensagens dentro dos canais

Durante a fase de estabelecimento do canal uma verificação é feita para verificar se existem recursos suficientes para cada canal que for alocado [MIT 2002].

4.3 Visão geral da plataforma RTC

Para ser possível a comunicação de tempo real baseada em canais, como descrito na seção anterior, uma plataforma de comunicação foi projetada e implementada. Essa plataforma é o sistema central no qual se baseia a comunicação entre os nodos *SCI*. Como o projeto de uma plataforma de comunicação de tempo real é uma tarefa complexa, as diversas funcionalidades que devem ser implementadas foram separadas em módulos ou camadas formando uma pilha de protocolos. Um exemplo conhecido desse tipo de pilha é o modelo de referência OSI mostrado no capítulo 2.

No presente sistema, cada funcionalidade da plataforma é implementada em um módulo separado, tendo uma interface bem definida. Implementou-se um conjunto de módulos separados com funcionalidades específicas para aumentar-se a coesão de cada módulo da plataforma. Esses módulos são integrados com o núcleo do sistema operacional através do comando *insmod* (veja seção 3.2.2 do capítulo 3). A comunicação entre os diferentes módulos é feita através da tabela de símbolos global do núcleo do Linux.

Uma visão geral da plataforma apresentando os diversos módulos que a compõem além da interação (comunicação) entre eles pode ser vista na figura 4.2.

Na figura podemos ver diversos blocos que correspondem a módulos pré-existentes (infra-estrutura) mais a plataforma desenvolvida. As camadas de software que fazem parte do escopo desse trabalho são apresentadas de forma hachurada na figura. No nível inferior da figura está o hardware de um nodo do cluster *SCI*. Cada nodo do cluster é formado por um microcomputador completo mais, no mínimo, um adaptador de rede *SCI* localizado no barramento PCI.

O próximo nível é formado pelo núcleo de tempo real *RTAI* que, conforme foi visto no capítulo anterior, é responsável pelo gerenciamento das interrupções do hardware e também oferece uma série de serviços que são utilizados por parte da arquitetura proposta.

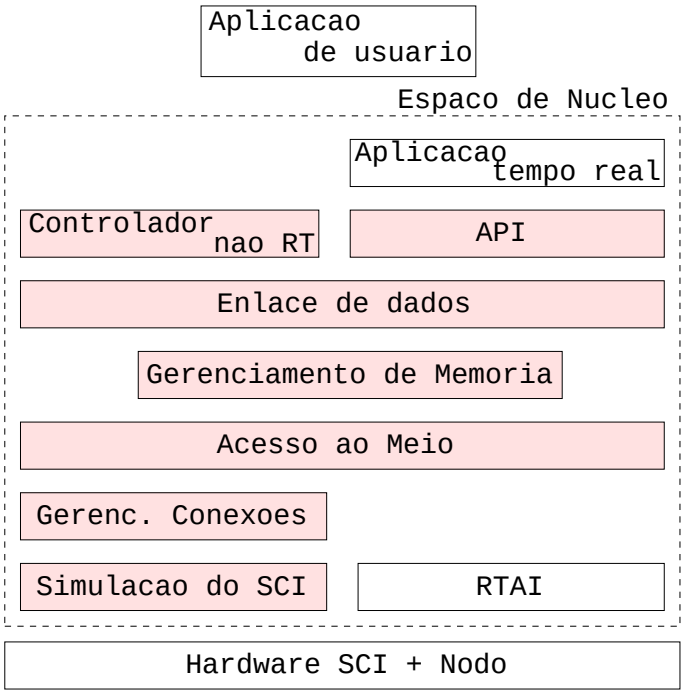

FIGURA 4.2 – Visão geral da plataforma proposta

O módulo imediatamente acima (Simulação Local do SCI) foi desenvolvido com o objetivo de tornar possível a utilização da plataforma de software em um único nodo (máquina local) tanto na fase de desenvolvimento quanto na de operação sem a necessidade de haver adaptadores *SCI*. Ele é responsável pela simulação tanto do hardware SCI quanto da biblioteca de inicialização *Icm* apresentada no capítulo anterior. O fato de poder-se executar software desenvolvido para clusters *SCI* sem a presença do hardware *SCI* traz vantagens de desenvolvimento, demonstração e depuração, visto o elevado custo dos adaptadores de rede.

O próximo módulo na arquitetura é a camada de gerenciamento de conexões. Ela é responsável pelo estabelecimento de todas as conexões que serão utilizadas pelo sistema. Inicialmente nenhum nodo do cluster pode enviar ou receber dados através da rede de comunicação. Isso porque não existe nenhuma área de memória sendo disponibilizada (exportada) por nenhum nodo, e por conseqüência nenhuma conexão com memória remota disponível. No momento que esse módulo é inserido no núcleo do sistema operacional, ele verifica o status do adaptador de rede *SCI* e, se o adaptador está operacional, exporta os blocos de memória necessários para a comunicação. Após todos os blocos estarem disponíveis, o módulo de cada estação importa (conecta) as áreas remotas necessárias para a intercomunicação. Essas áreas agora estão visíveis para as camadas superiores da plataforma. Isso é feito disponibilizando o endereço dessas áreas de memória na tabela de símbolos global do núcleo do sistema operacional.

É importante observar que os dois módulos de software descritos acima não necessitam do núcleo de tempo real (RTAI) situando-se na área de núcleo ("kernel space"). Para maiores detalhes, verifique seção Linux do capítulo 3. Esses dois módulos (simulação do *SCI* e gerenciamento de conexões) rodam dentro do núcleo do sistema operacional Linux padrão.

A próxima camada da arquitetura de comunicação é responsável pelo controle

de acesso ao meio físico (*SCI*). É um dos módulos principais no estabelecimento de uma disciplina para o acesso à rede. Além disso, a inicialização de todos os serviços de tempo real é realizada nessa camada. Uma *thread* dentro desse módulo é responsável pela sincronização da arquitetura de comunicação nos diversos nodos do cluster. O método de acesso ao meio físico utilizado será mostrado em seção posterior.

A camada de gerenciamento de memória oferece primitivas para que a camada de enlace e o módulo controlador não-RT possam alocar blocos de memória que são utilizados para o armazenamento de pacotes de comunicação. Além disso, chamadas que permitam a criação e o gerenciamento de listas de pacotes internas da plataforma são também fornecidas por essa camada.

O bloco de enlace de dados é responsável pela colocação dos diversos pacotes gerados pela camada API e pelo controlador *Ethernet* no meio físico. Essa colocação é guiada pela disciplina de acesso gerada na camada de acesso ao meio. A correta manipulação dos blocos de memória oferecidos pela camada de gerenciamento de conexões é tarefa desse módulo. Além disso, o controle dos limites de comunicação para cada canal de comunicação também é realizado por esse módulo.

A API é responsável pela geração de mensagens associadas a um determinado canal de comunicação. Essas mensagens são geradas a partir de requisições das tarefas de tempo real que rodam dentro do núcleo do sistema operacional. As diversas rotinas de comunicação providas pela API são registradas na tabela de símbolos do sistema operacional e portanto estão diretamente acessíveis para os processos de tempo real que queiram utilizá-las para enviar mensagens. A vantagem desse método é que unifica o procedimento de chamadas do núcleo de tempo real RTAI com a API de comunicação.

É importante observar que esse conjunto de módulos (acesso ao meio físico, enlace de dados e API) é escalonado pelo núcleo do sistema operacional de tempo real sendo que eles se encontram dentro da área conhecida por "real-time space". Eles se utilizam de todo o suporte de rotinas dada por esse núcleo.

Por fim, o módulo controlador não-RT é responsável por gerenciar o tráfego que não tem restrições temporais. Conforme apresentado na introdução, um dos objetivos da presente arquitetura é lidar tanto com tráfego de tempo real quanto com tráfego comum. Decidiu-se construir-se um controlador de dispositivo que disponibiliza para o núcleo do Linux um novo dispositivo de rede da classe *Ethernet*. Na verdade esse módulo emula um dispositivo *Ethernet* recebendo pacotes IP do núcleo do Linux e os repassa para a biblioteca de comunicação através de um canal especial para serem mandados em baixa prioridade para o destino. Uma vez no destino, a biblioteca RT entrega novamente esses pacotes para o módulo controlador não-RT e esse repassa para o núcleo do Linux como se tivesse sido recebido por uma placa de rede *Ethernet* (veja figura 4.3). Isso possibilita que, no cluster, além de aplicações utilizando comunicação de tempo real, pode-se ter outras aplicações comunicando-se por TCP/IP ou mesmo MPI sem nenhuma modificação.

Observando-se o modelo OSI apresentado no capítulo de revisão bibliográfica, pode-se ver que várias camadas da arquitetura proposta correspondem diretamente a camadas propostas pelo modelo OSI. Já o módulo de API pode ser comparado à camada de transporte por controlar o transporte correto dos pacotes utilizando para isso códigos de verificação de erro, como o CRC. Além disso, se houver mudança da tecnologia de hardware, essa camada isola a aplicação oferecendo uma interface

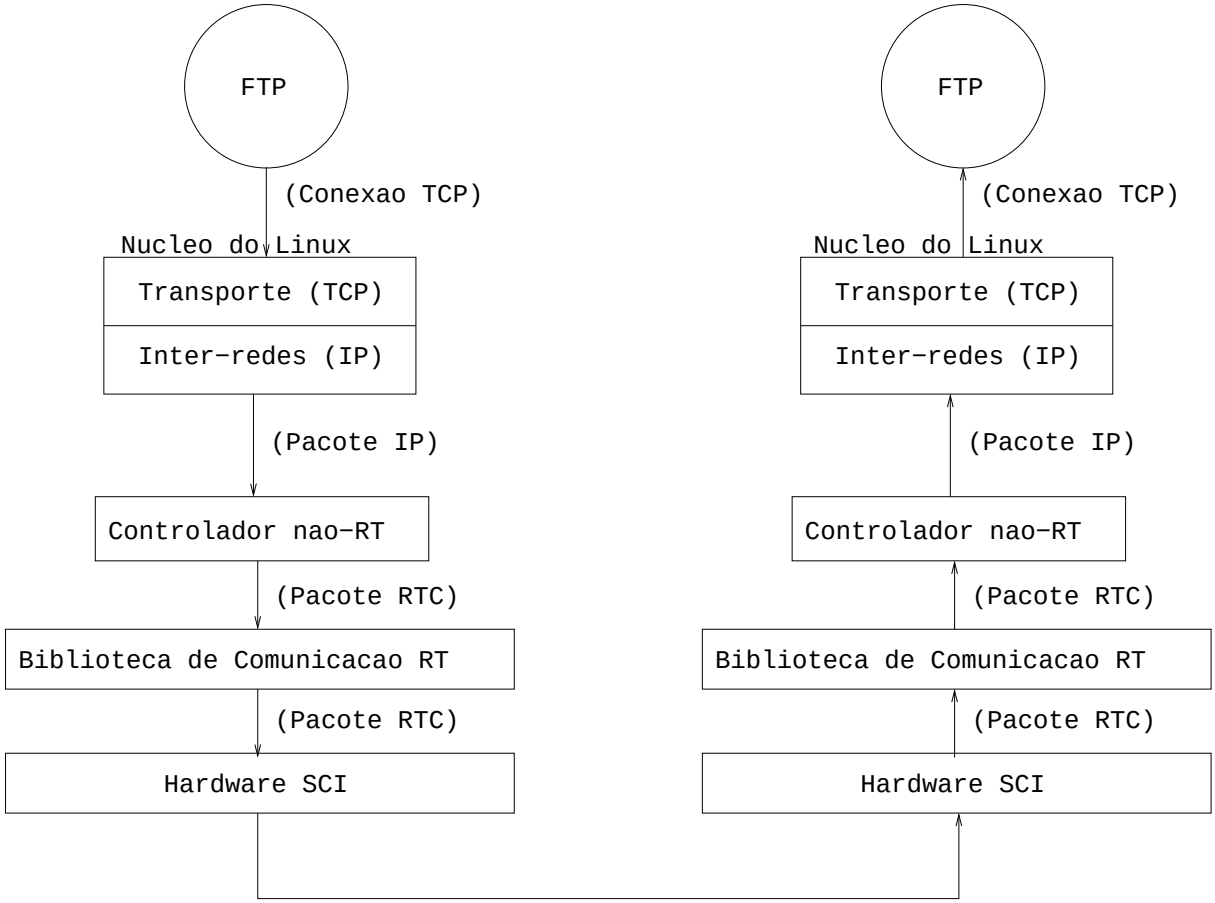

FIGURA 4.3 – Exemplo de comunicação TCP/IP através do *SCI* utilizando a plataforma proposta

única.

Como se pode observar, a totalidade da presente arquitetura implementa serviços de comunicação baseados em mensagens sobre um hardware que proporciona o paradigma de memória compartilhada distribuída. Esses serviços podem tratar mensagens com limites temporais de entrega dentro de canais com características garantidas e também mensagens normais (tráfego IP). Para garantir a comunicação de tempo real, o método de reserva de banda por canal é utilizado. Sendo a plataforma implementada por módulos com interface bem definida, facilmente novos tipos de protocolos/serviços podem ser integrados.

Nas próximas seções, os diversos blocos que compõem a plataforma serão apresentados em detalhes.

4.4 Simulação Local do *SCI*

Módulo de núcleo: `scilocal.o`

Devido ao alto custo de placas *SCI* e a impossibilidade do desenvolvimento da arquitetura sem a presença do cluster, optou-se por simular as funcionalidades do adaptador *SCI* e o seu controlador (*driver*).

4.4.1 Objetivos

Os objetivos que nortearam o projeto de um módulo que emulasse a plataforma de rede são os seguintes:

- permitir a execução de programas distribuídos inicialmente projetados para o cluster *SCI* em uma máquina local
- exigir a mínima modificação possível no código fonte do respectivo programa
- facilitar a descoberta de erros por, ao contrário do adaptador SCI, ser possível a verificação do estado interno do adaptador e também a modificação do módulo para gerar relatórios de estado personalizados

Embora o módulo de software tenha sido construído inicialmente para permitir a execução da plataforma de comunicação descrita nessa tese, ele pode também ser utilizado por qualquer outro programa que execute sobre o *SCI*.

Em uma futura extensão, esse módulo pode ser utilizado para a injeção de falhas com o objetivo de testar o comportamento da plataforma construída sobre o hardware *SCI*

4.4.2 Funcionamento

Conforme visto no capítulo de infra-estrutura, os adaptadores de rede *SCI* permitem a utilização da abstração *DSM* (*distributed shared memory*) para a comunicação de dados, embora no nível físico o adaptador traduza a escrita/leitura em memória remota em envio de mensagens (tradução feita em hardware pelo adaptador *SCI*).

É importante introduzir nesse momento algumas diferenças entre esse paradigma de comunicação e a usual troca de mensagens utilizada por outros adaptadores de

rede. O modelo de programação utilizado nas camadas superiores terá que utilizar os dois paradigmas, pois o meio físico é baseado em memória compartilhada enquanto que o modelo de mensagens deve ser fornecido aos processos de tempo real (aplicação), conforme os requisitos citados no capítulo 1. Em [COU 2001] encontra-se material vasto sobre *DSM* e também uma explicitação das diferenças entre os dois modelos de programação:

> No modelo de troca de mensagens, as variáveis devem ser organizadas e empacotadas em um processo, transmitidas e desempacotadas em outras variáveis no processo receptor. Em contraste, com variáveis (memória) compartilhada, os processos envolvidos podem compartilhar variáveis diretamente, o que torna o empacotamento desnecessário — mesmo ponteiros podem ser compartilhados[1] — portanto não existe necessidade de comunicação separada. (...)
>
> Em favor do paradigma da troca de mensagens, por outro lado, está o fato que ele permite os processos se comunicarem enquanto protegidos dos outros por áreas locais de memória, tendo endereços privados de memória, enquanto que utilizando DSM pode-se provocar a falha de outros processos escrevendo dados errôneos na memória compartilhada. Além disso, o paradigma de troca de mensagem pode suportar máquinas heterogêneas com diferentes normas de representação.

O *SCI* permite a existência do paradigma de memória compartilhada entre os nodos do cluster pelo mapeamento da memória remota em áreas locais. Da mesma forma, o presente módulo permite a utilização do mesmo paradigma aproveitando-se do fato de que todos os processos estão numa mesma máquina, o que facilita bastante a tarefa.

As tarefas que o módulo deve realizar para atingir os objetivos são:

- Simular a fase de inicialização que é composta por criação de blocos de memória a serem compartilhados e o posterior compartilhamento com a conexão de outros processos em máquinas remotas a esses blocos
- Após isso, uma área de memória compartilhada da memória principal do próprio nodo local será utilizada para a troca de dados entre os processos de tempo real.

É importante observar que apenas um subconjunto de toda funcionalidade do *SCI* é emulado pelo módulo aqui descrito. As características que são implementadas são:

- O processo de criação de um bloco de memória que é após exportado
- A conexão e o mapeamento de um bloco remoto para a área de endereçamento local
- O modo de leitura/escrita padrão dessas áreas de memória

[1] No caso de DSM, o compartilhamento de ponteiros deve ser evitado pois dois nodos diferentes têm áreas de endereçamento diferentes

Já as seguintes funcionalidades não são implementadas:

- O modo de leitura destrutivo e o modo de escrita com interrupção
- Interrupções remotas
- Transferências via DMA

O primeiro ítem não é implementado porque o modelo local de memória não contempla esse tipo de operação. Já a interrupção remota não foi desenvolvida por ser uma operação lenta no *SCI* não sendo utilizada pela arquitetura de comunicação. Por não serem permitidas chamadas *Icm* dentro de módulos de tempo real e por ser utilizada memória fisicamente dispersa para a simulação, a simulação do DMA não é necessária e nem possível.

Simulação da biblioteca *Icm*

No capítulo 3 a biblioteca Icm foi apresentada. A tarefa principal do presente módulo é a simulação dessa biblioteca e a disponibilização de blocos de memória locais compartilhados entre os processos de tempo real, que o utilizarão como se estivessem acessando blocos compartilhados pelo adaptador *SCI*.

Para poder-se simular a biblioteca *Icm*, as diversas chamadas devem ser implementadas de maneira consistente (ou seja, do ponto de vista da aplicação que as chama, as funções devem manter a sua semântica original). Como as diversas rotinas utilizam diversos identificadores formados por tipos de dados abstratos, estes tipos também devem ser criados a fim de que não sejam necessárias modificações no código fonte das aplicações.

A biblioteca *Icm* é formada por dois conjuntos de rotinas: gerenciamento da memória local e remota.

Gerenciamento da memória local

O gerenciamento de memória local, como foi visto, permite a criação de blocos de memória que poderão ser oferecidos às outras máquinas da rede. Os principais tipos de dados utilizados nessa operação são o *lchunk*, que representa um bloco de memória que pode ser oferecido para outros nodos da rede, e o *chunkoffer*, que representa a oferta do bloco propriamente dita.

A principal função encarregada da criação de blocos de memórias que serão exportados é a `IcmCreateLocalChunk` que recebe os parâmetros ponteiro de um identificador de bloco, tamanho do bloco, tipo de memória e tipo de mapeamento a ser utilizado (para detalhes verifique seção "biblioteca *Icm*" do capítulo 3).

A assinatura da função é:

```
void IcmCreateLocalChunk(PLCHUNK *pchunk,unsigned32 uSize,
        MEMTYPE memtype, LMAPTYPE maptype, BOOL *fOK)
```

Para gerenciar os diversos blocos de memória que podem ser criados pela aplicação que utiliza esse módulo, criou-se um vetor cujas entradas são estruturas de dados mostradas na figura 4.4. Todos os tipos de de dados abstratos utilizados como parâmetros dessa função e de todas as outras da biblioteca Icm foram recriados e são mostrados na figura 4.5

No momento da chamada da função `IcmCreateLocalChunk`, se existir uma posição ainda não ocupada no array `ChunkTable`, o módulo de simulação utiliza

```
1 typedef struct
2 {
3   MEMTYPE memtype;
4   LMAPTYPE maptype;
5   unsigned32 uSize;
6   void *pointer;
7 } TChunkTable;
8
9 static TChunkTable ChunkTable[MAXNUMCHUNK];
```

FIGURA 4.4 – Estrutura utilizada para o suporte à criação de blocos de memória

essa posição e armazena os parâmetros de entrada da função no *array*. Após isso, o módulo aloca a memória que será utilizada de forma compartilhada entre os processos e guarda seu endereço no campo *pointer* da estrutura mostrada na função 4.4. Tudo isso é protegido por semáforos do núcleo pois pode-se ter dois processos chamando de forma concorrente a função, o que caracteriza condição de corrida.

Para fazer a alocação dinâmica de memória utilizou-se uma chamada do sistema operacional. O núcleo do sistema operacional Linux oferece diversas chamadas como `kmalloc`, `get_free_page` e `vmalloc` para alocação dinâmica de memória. Após o estudo das diversas chamadas disponíveis (que podem ser vistas no capítulo de infra-estrutura), decidiu-se por utilizar a função `vmalloc` devido aos seguintes motivos:

- Não é necessária a alocação de memória contígua visto que só utiliza-se memória local que não será transferida por DMA para nenhum dispositivo
- O uso de outras funções limitaria o tamanho de memória que poderia ser alocado, além de retirar memória contígua do *pool*, competindo com outras tarefas do núcleo e com processos de usuário por um recurso limitado

Como desvantagem da utilização da chamada `vmalloc` pode-se citar:

- Ela impõe uma pequena diferença entre o código que é executado sobre a rede *SCI* do que utiliza a biblioteca de simulação. No ambiente real, quando um nodo mapeia uma área de memória remota no seu espaço de endereçamento, ele deve utilizar a chamada de núcleo `ioremap`. Isso porque o endereço retornado pela função da biblioteca *Icm* é um endereço físico, e um processo normalmente utiliza somente endereços lineares. A chamada *ioremap* preenche a tabela de páginas do Linux mapeando o endereço físico em um endereço linear acima de `VMALLOC_START` (para detalhes, ver capítulo "Ambiente de execução"). Já o módulo de simulação sempre retorna endereços lineares, que não necessitam ser traduzidos. Sendo assim, utilizando-se o *SCI* a função `ioremap` deve ser chamada; já na simulação ela não pode ser utilizada.
- O uso da memória cache do hardware é mais eficiente quando blocos de memória contíguos são acessados pois o principio da localidade norteia o hardware de cache; por conseqüência temos tempos médios de acesso à memória menores quando a memória é alocada de forma contígua.

```
1  typedef int  PLCHUNK;
2  typedef unsigned unsigned32;
3  typedef unsigned short int unsigned16;
4  typedef int ICM_STATUS;
5  typedef int OCONNECTOR;
6  typedef enum
7  {
8    MEMTYPE_ORDINARY=0,
9    MEMTYPE_COND_INT
10 } MEMTYPE;
11 typedef enum
12 {
13   LMAP_CONSISTENT=0,
14   LMAP_STREAMING
15
16 } LMAPTYPE;
17 typedef int BOOL;
18 typedef int PADAPTER;
19 typedef int PCHUNKOFFER ;
20 typedef enum
21 {
22   LCBR_CONNECTED,
23   LCBR_DISCONNECTED,
24 } LCBREASON;
25 typedef enum
26 {
27   RCBR_KILLED=0,
28   RCBR_OTHER,
29 } RCBREASON;
30 typedef int PCONNECTOR;
31 typedef int PCHUNKCLIENT;
32 typedef void (*OFFERCB) (LCBREASON lcbr,PCHUNKCLIENT pclt);
33 typedef void (*CONNECTORCB) (RCBREASON rcbr,PCONNECTOR pclt);
34 typedef enum
35 {
36   CR_CONNECTED = 0,
37   CR_REFUSED,
38   CR_NO_RESPONSE,
39   CR_FAILURE
40 } CONNECT_RESULT;
41 typedef int PPHYSMAP;
42 typedef enum
43 {
44   RMAP_GATHERING=0,
45   RMAP_IO,
46   RMAP_CSR,
47   RMAP_COND_INT,
48   RMAP_INCREMENTER
49 } RMAPTYPE;
```

FIGURA 4.5 – Estrutura utilizada para o suporte à criação de blocos de memória

```
1  typedef struct
2  {
3    PLCHUNK pchunk;
4    PADAPTER padapt;
5    unsigned32 uModuleID;
6    unsigned32 uChunkID;
7    OFFERCB callback;
8    LMAPTYPE maptype;
9    unsigned16 uOwnerNodeID;
10   BOOL introduced;
11 } TOfferTable;
12
13 static TOfferTable OfferTable[MAXNUMOFFER];
```

FIGURA 4.6 – Estrutura utilizada para o suporte à criação de ofertas de blocos de memória

Após a alocação de memória e o preenchimento dos campos da tabela, a posição no vetor é retornada no identificador `PLCHUNK`. Já a função complementar `IcmDestroyLocalChunk` libera a memória alocada e a posição na tabela de *chunks*.

A função `IcmCreateChunkOffer`, cuja assinatura é:

```
void IcmCreateChunkOffer (PLCHUNK pchunk,
                          PADAPTER padapt,
                          unsigned32 uModuleID,
                          unsigned32 uChunkID,
                          OFFERCB callback,
                          PCHUNKOFFER *pofr,
                          LMAPTYPE maptype,
                          BOOL *fok)
```

é responsável por criar uma oferta de bloco de memória criado a um determinado adaptador de rede com um certo nome (para detalhes veja a seção "biblioteca Icm" em infra-estrutura). Resumidamente os atributos de uma oferta são: o bloco que será ofertado, o adaptador de rede em que será feita a oferta, o nome da oferta que é dividido em dois identificadores de 32 bits, uma rotina de *callback* que é chamada quanto um evento importante como conexão acontece, atributos que mudam a forma pela qual o acesso remoto afeta a consistência da memória alocada. Todos esses parâmetros são armazenados no vetor que contém os campos mostrados na figura 4.6

Quando a função `IcmCreateChunkOffer` é chamada, os parâmetros de entrada são salvos na próxima posição livre do vetor `OfferTable` e a posição do registro no vetor é retornada no parâmetro "`pofer`". Complementarmente, normalmente no estágio de finalização, a rotina `IcmDestroyChunkOffer` é chamada e é responsável por eliminar a devida entrada da tabela de ofertas.

Após a criação de uma oferta de bloco de memória, o caminho natural de uma aplicação é fazer a chamada da função `IcmIntroduceOffer` que recebe o manipulador de oferta "`pofer`" e introduz essa oferta na rede de forma que outros computadores remotos possam utilizá-la.

```
1 typedef struct
2 {
3   PCHUNKOFFER pofr;
4   CONNECTORCB callback;
5 } TConnectTable;
6
7 static TConnectTable ConnectTable[MAXNUMCONNECT];
```

FIGURA 4.7 – Tabela de conexões ativas

Na biblioteca de simulação a única ação a ser tomada é a mudança da variável *booleana* `introduced` para verdadeiro. No momento que um nodo remoto deseja conectar com um bloco de memória disponibilizado na rede, esse bloco necessariamente tem que ter sido introduzido, caso contrário a conexão é impossível.

Gerenciamento da memória remota

Para uma comunicação no cluster *SCI* ocorrer, é necessário que a máquina remota (cliente) se conecte com o bloco de memória ofertado. Após essa etapa, esse bloco de memória poderá ser mapeado em endereços locais da memória física através do adaptador de rede. Os principais tipos de dados abstratos envolvidos nessa fase da operação são o *connector* que representa uma conexão com um bloco de memória remoto e o *physmap* que representa um mapeamento físico de um bloco de memória que foi conectado.

A primeira função de gerenciamento remoto da memória é a `IcmConnect` cuja assinatura é:

```
void IcmConnect (PADAPTER padapt,
                 unsigned16 uOwnerNodeID,
                 unsigned32 uModuleID,
                 unsigned32 uChunkID,
                 CONNECTORCB callback,
                 CONNECT_RESULT *cr,
                 PCONNECTOR *pcon)
```

Ela tenta estabelecer uma conexão com o bloco de memória estabelecido pelo nome dado pela combinação de `uOwnerNodeID`, `uModuleID` e `uChunkID`. A função bloqueia até a conexão ser estabelecida, recusada ou um limite de tempo ser excedido.

No módulo de simulação, a primeira tarefa a ser realizada na chamada da função é a procura pela combinação dos três identificadores dentro da tabela de ofertas disponíveis. Se não achado, a função retorna uma código de erro pelo parâmetro `CONNECT_RESULT`. Caso contrário, a tabela de conexões (que pode ser vista na figura 4.7) é preenchida (em uma posição disponível).

Além disso, a rotina de *callback* registrada por ocasião do oferecimento do bloco de memória é chamada para indicar a conexão de um processo à porção de memória oferecida.

Já a desconexão retira essa entrada da tabela de conexões e chama novamente o *callback* para informar o processo que ofereceu o bloco de memória que o nodo remoto está desconectando.

```
1 typedef struct
2 {
3   PCONNECTOR pcon;
4   RMAPTYPE maptype;
5   void *pointer;
6 } TMapTable;
7
8 static TMapTable MapTable[MAXNUMMAP];
```

FIGURA 4.8 – Tabela de mapeamentos

Após a conexão, o mapeamento de um bloco remoto de memória no espaço de endereçamento local é necessário. Como foi visto, o tipo `physmap` representa esse mapeamento. Todas as requisições direcionadas a essa área de memória serão repassadas para o adaptador do nodo remoto que mandará a resposta adequada.

A chamada da biblioteca *Icm* utilizada para criar um mapeamento a partir de uma conexão é:

```
void IcmMap (PPHYSMAP *pmap,
             PCONNECTOR pcon,
             unsigned32 uOffset,
             unsigned32 uSize,
             RMAPTYPE maptype,
             BOOL *fOK)
```

Será mapeado `uSize` a partir do `uOffset` dentro do bloco de memória remoto. A ação tomada pelo módulo de simulação é o preenchimento de uma posição na tabela de mapeamento `MapTable` (veja figura 4.8). O endereço ("*pointer*") é retornado da tabela de "*chunks*" e será o endereço retornado posteriormente para uso compartilhado. O *handler* de mapeamento retornado contém a posição da entrada na tabela.

Um determinado bloco de memória exportado pode ser composto de vários sub-blocos quando mapeado e esse mapeamento terá diferentes deslocamentos no nodo remoto (cliente). Assim, a função `IcmGetNumberOfViews` retorna o número de mapeamentos necessários para cobrir completamente o bloco exportado.

Devido a uma limitação de hardware, máquinas onde não existe uma MMU de E/S o tamanho máximo de um sub-bloco é limitado em uma página de memória. Sendo assim, blocos maiores que, no caso da arquitetura Intel, 4096 bytes terão necessariamente mais de um sub-bloco e por conseqüência mais de uma visão de mapeamento. O caso da biblioteca de simulação é diferente por se tratar somente de memória local alocada continuamente no endereçamento linear (embora fisicamente pode ser não contígua). Por isso, no módulo de simulação `IcmGetNumberOfViews` retorna sempre um sub-bloco.

Por fim, temos a função `IcmGetPhysicalAddressOfView` que é responsável pelo retorno do endereço físico de uma visão (sub-bloco) do mapeamento. Na biblioteca de simulação, a função retorna o endereço armazenado na tabela de mapeamento (`MapTable`).

As figuras 4.9 e 4.10 sintetizam a funcionalidade do módulo de simulação.

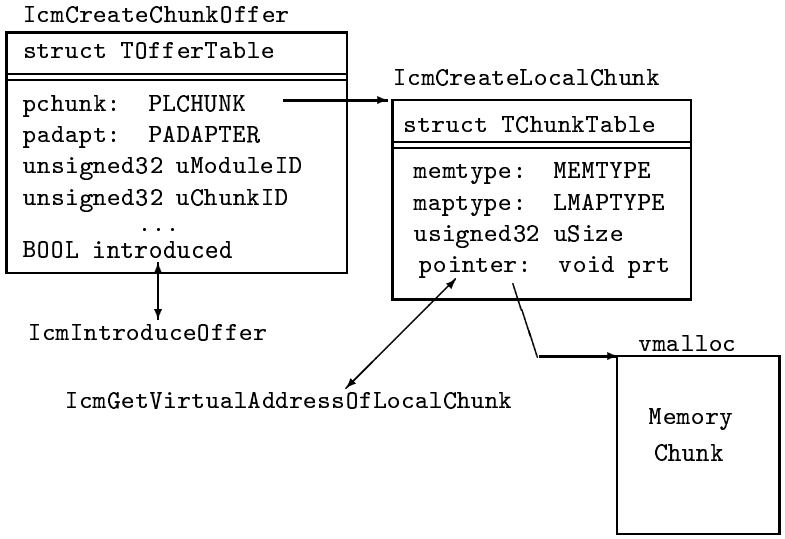

FIGURA 4.9 – Visão geral do funcionamento da simulação do *SCI* – gerenciamento da memória local

A primeira figura apresenta as funções que gerenciam a criação de blocos locais de memória e sua respectiva exportação (`IcmCreateLocalChunk` e `IcmCreateChunkOffer`). As estruturas de dados que armazenam as informações relevantes para o bloco de memória e para a exportação (oferecimento) são mostradas abaixo das respectivas chamadas. Já as outras duas funções, `IcmIntroduceOffer` e `IcmGetVirtualAddressOfLocalChunk` agem também sobre as estruturas de dados já apresentadas. `IcmIntroduceOffer` coloca o valor booleano verdadeiro no campo *Introduced* e `IcmGetVirtualAddressOfLocalChunk` retorna o ponteiro do endereço linear (virtual) do bloco exportado.

Já a segunda figura mostra uma visão geral do gerenciamento da memória remota. Funções que armazenam estado dentro de tabelas no módulo, como já foi visto, são a `IcmConnect` e a `IcmMap`. A `IcmGetNumberOfViews` e a `IcmGetPhysicalAddressOfView` retornam dados vindo da tabela de mapeamento.

Uso da memória compartilhada

Na figura 3.5 do capítulo anterior, podemos ver dois processos se comunicando utilizando o *SCI* e na figura 4.11 os mesmos dois processos usando memória local compartilhada fornecida pela biblioteca de simulação. Na primeira figura, o processo P_1 realiza uma instrução de leitura na memória que ele enxerga como local, mas que na verdade é uma área de memória importada de outro nodo, onde está o processo P_2, dono dessa área que foi exportada. A placa controladora do *SCI* recebe essa requisição (pois esse endereço de memória pertence na verdade ao barramento de E/S *PCI* sendo então remapeado na memória principal com a função *ioremap* do núcleo do Linux, ver capítulo 3 para maiores detalhes) e percebe que esse endereço se refere ao outro nodo da rede. Então o adaptador de rede manda uma requisição de leitura que é recebida pela placa *SCI* do nodo portador do bloco de memória requisitado. Após ser lido da memória principal, o dado então é retornado pela rede e para o processo requisitante através do barramento PCI e finalmente o processo pode continuar no seu fluxo de instruções. Uma importante diferença entre o processamento com o adaptador *SCI* e o com a simulação é que a última

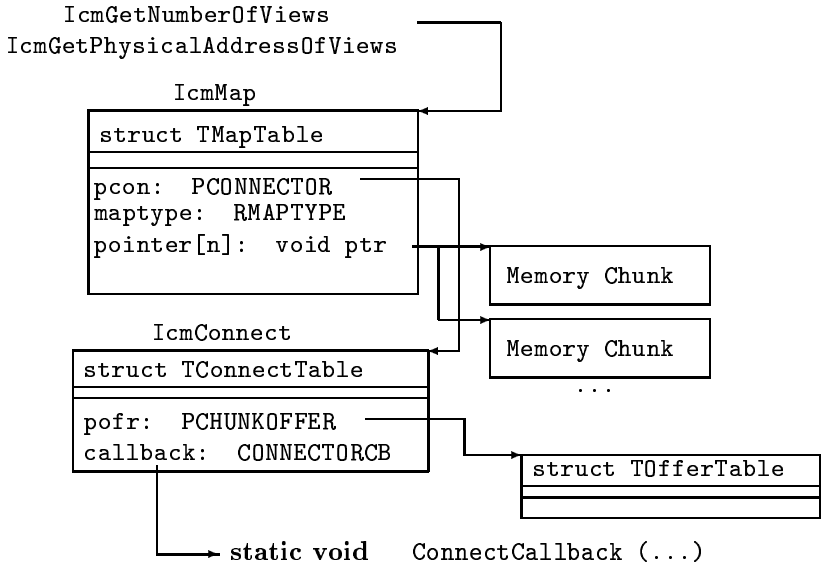

FIGURA 4.10 – Visão geral do funcionamento da simulação do *SCI* – gerenciamento da memória remota

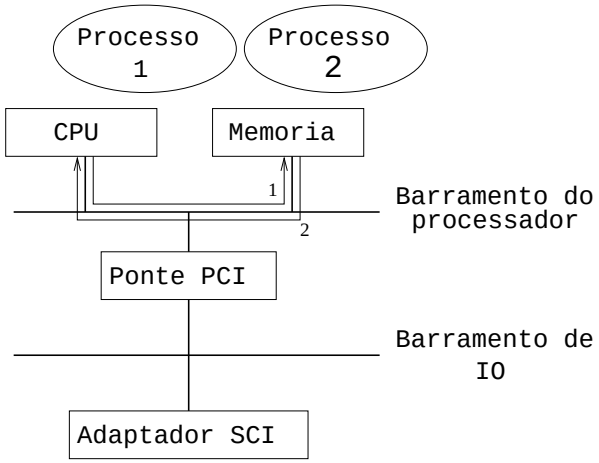

FIGURA 4.11 – Comunicação entre dois processos em máquina local utilizando a biblioteca de simulação do *SCI*

só oferece escrita/leitura ordinária (não é possível utilizar a leitura destrutiva e a escrita combinada com interrupção[2]).

Já na segunda figura, o caminho é mais curto. Como uma área de memória compartilhada está fazendo o papel do adaptador *SCI*, a requisição de leitura pode ser atendida simplesmente por uma leitura na memória local do processador.

Conforme visto nessa seção, é possível uma simulação da rede *SCI* pela simulação da sua plataforma de software (biblioteca *ICM*) e a utilização de memória principal ordinária como área compartilhada. Essa abordagem mostrou-se funcional até certo ponto, pois existem funcionalidades do adaptador *SCI* que não são simuladas de maneira trivial. Podemos citar a leitura destrutiva de memória e a escrita com interrupção. Essas funções não são simuladas pela corrente implementação do módulo de simulação. Além disso, acessos por DMA utilizado a biblioteca *ICM* tam-

[2]Para maiores detalhes consulte o capítulo "Infra-estrutura".

bém não são implementados. Porém, para a plataforma de comunicação proposta, nenhuma das funcionalidades não implementadas é necessária, tornando o módulo de simulação compatível com todos os requisitos da arquitetura de comunicação.

4.5 Camada de conexão

Módulo de núcleo: `rtc_sciconnect.o`

Para que a comunicação entre nodos de uma rede *SCI* seja possível, é necessário o estabelecimento de áreas de memória que serão compartilhadas por processos em máquinas distintas. A criação dessas áreas é o objetivo principal da camada de conexão. A organização das áreas compartilhadas de memória deve ser feita tendo em vista o tipo de comunicação que será realizado. Na plataforma desenvolvida nesse trabalho, não existe restrição entre quais pares de nodos podem trocar mensagens. Sendo assim, todo o nodo deve disponibilizar uma área para ser escrita por nodos remotos. A devida alocação de blocos de memória compartilhados com a correta estruturação de conexões é uma tarefa dessa camada. Além disso, o módulo aqui apresentado deve disponibilizar os endereços dos blocos compartilhados para as camadas superiores.

4.5.1 Objetivos

Os objetivos principais do módulo de conexão são:

- Gerenciamento dos blocos locais e conexões
- Estruturação da interconectividade entre nodos de forma apropriada para o tipo de comunicação necessária
- Disponibilização dos endereços de blocos de memória compartilhados para as camadas superiores

4.5.2 Design da memória compartilhada

Devido à diferença de latência nas diversas modalidades de comunicação que o *SCI* oferece, que são mostradas na tabela 4.1, decidiu-se utilizar somente a operação de menor latência que é a escrita remota. Portanto, toda a comunicação entre nodos é realizada somente com escrita remota. Isso possibilitará menor tempo de latência nas comunicações, como será visto em seções futuras. Para o nodo A enviar uma mensagem para o nodo B, ele deverá escrever um uma área compartilhada previamente disponibilizada pelo nodo B. Essa decisão de utilizar-se somente escrita remota portanto influencia na organização dos blocos de memória compartilhados.

Existem dois modelos de disponibilização e conexão da memória compartilhada que foram adotados por esse módulo. Suponha um *cluster* com n máquinas. No primeiro modelo, cada nodo exporta um bloco de memória que será conectado e utilizado para escrita por todos os outros $n-1$ nodos do *cluster*. Já no segundo modelo cada máquina oferece para conexão $n-1$ blocos de memória. Todos os outros nodos da rede conectam com um único bloco dentre os exportados. O primeiro modelo de conexão para um cluster de três nodos pode ser visto na figura 4.12 enquanto que o segundo é mostrado na figura 4.13.

TABELA 4.1 – Latência de comunicação típica de um *cluster SCI*

Modalidade de comunicação	Latência
Escrita remota	$5\mu s$
Leitura remota	$15\mu s$
Leitura destrutiva	$20\mu s$
Interrupção	$80\mu s$

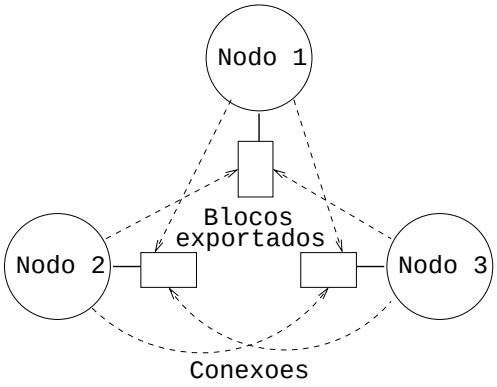

FIGURA 4.12 – Padrão de exportação de blocos de memória onde cada nodo exporta um bloco que é conectado por todos os outros

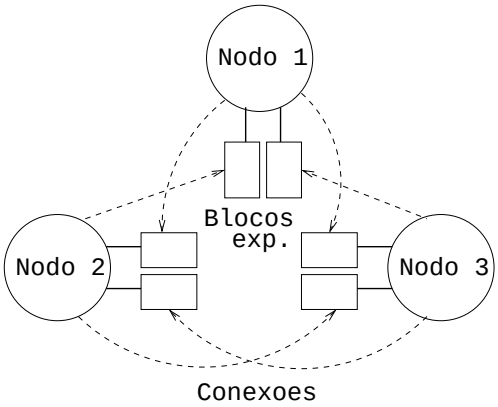

FIGURA 4.13 – Padrão de exportação de blocos de memória onde cada nodo exporta um bloco para cada máquina remota no *cluster*

A grande vantagem do segundo modelo de exportação de blocos compartilhados de memória está na possibilidade de um maior paralelismo de comunicação em relação ao primeiro. Suponha que o nodo 1 escreva uma mensagem em um bloco de memória exportado pelo nodo 2. Em um momento posterior, o nodo 3 deseja também se comunicar com o nodo 2 através de escrita remota. Se o padrão de conexão utilizado for o primeiro proposto (figura 4.12), essa segunda escrita remota só poderá ser efetuada após o nodo 2 ter retirado a informação recebida do nodo 1. Senão essa informação será sobrescrita. Já no segundo modelo, cada bloco de memória compartilhado só precisa ser lido antes do mesmo nodo escrever novamente no bloco.

É importante ressaltar que devido à decisão de não utilizar-se leitura remota, não existe a possibilidade de um nodo ler inicialmente o bloco de memória que ele deseja escrever para verificar onde ele pode começar a escrever. Se isso fosse possível, poder-se-ia utilizar uma posição especial para indicar quais posições do bloco contêm dados que não podem ser reescritos por não terem ainda sido lidos. Poder-se-ia nesse caso utilizar um buffer circular para facilitar esse controle.

O controle da sincronização das escritas remotas/leituras dentro de um determinado bloco de memória será visto em seções posteriores. Pode-se adiantar que a existência de sincronização de relógio permitirá que cada nodo saiba o momento adequado de ler um determinado bloco de memória oferecido e também o momento de escrever um bloco de memória remoto sem o perigo de ele ter sido reescrito ou existir condição de corrida em escritas/leituras.

4.5.3 Funcionamento

O módulo de conexão e exportação de memória utiliza-se da função de inicialização para disponibilizar o(s) bloco(s) de memória de acordo com o padrão de exportação selecionado no aplicativo de configuração da arquitetura que será descrito posteriormente.

Na função de inicialização do módulo (`init_module`), todos os blocos de memória necessários conforme o modelo de conexão adotado são exportados. Para isso, utiliza-se as seguintes chamadas de gerenciamento da memória local para cada bloco necessário:

`IcmCreateLocalChunk:` Cria um bloco de memória local propício para a exportação e utilização como memória compartilhada

`IcmCreateChunkOffer:` Cria uma oferta do bloco que pode ser introduzida na rede através do adaptador *SCI*. O bloco de memória é caracterizado pelo identificador do adaptador e dois identificadores do bloco, `chunkID` e `moduleID`

`IcmIntroduceOffer:` Introduz a oferta na rede sinalizando que o bloco está disponível para a conexão

`IcmGetVirtualAddressOfLocalChunk:` Retorna o endereço linear virtual do bloco criado para ele ser acessível para leitura/escrita da máquina local

Os identificadores do bloco de memória são escolhidos da seguinte forma:

- `moduleID` é lido do arquivo de configuração e é fixo

- `chunkID` depende do modelo de conexão adotado. No primeiro modelo, recebe um identificador fixo. Já no segundo, o identificador é igual ao número do nodo que irá conectá-lo

Após a exportação, cada nodo necessita conectar os blocos remotos das outras máquinas. Mas essa conexão não pode acontecer em qualquer momento. Ela só pode ocorrer quando todos os nodos já tiverem exportado os respectivos blocos locais, sob pena, do contrário, falharem as conexões. Decidiu-se deixar o usuário mandar um sinal para os módulos avisando que todos os nodos já estão com o módulo de conexão instalado, o que significa que todos os blocos já foram devidamente exportados.

Para existir iteração entre um processo de nível de usuário e um módulo de núcleo resolveu-se implementar o método `ioctl`. O `ioctl` é uma chamada implementada por muitos controladores de dispositivo para alterar sua configuração interna. Diferente de muitos outros métodos implementáveis por um controlador de dispositivo, como `read` ou `write` que têm normalmente a mesma funcionalidade para os diversos periféricos, a chamada `ioctl` é específica de cada controlador. Isso porque permite uma configuração específica para cada dispositivo ou a entrada em modos especiais de operação. No caso atual, não existe nenhum periférico a ser controlado pelo módulo aqui descrito; ele simplesmente será registrado como um controlador para poder implementar a chamada que irá realizar as conexões necessárias em cada módulo.

Normalmente um controlador de dispositivo pode implementar uma série de chamadas que serão utilizadas pelo programa de usuário para a interação com um dispositivo. Se o dispositivo for do tipo caracter, por exemplo, o núcleo do sistema operacional utiliza a estrutura `file_operations` que contém uma série de ponteiros para funções implementadas pelo controlador. Na inicialização do módulo do controlador, as diversas funções são registradas nessa tabela de ponteiros.

Um outro método possível para a comunicação entre um programa de usuário e um módulo de núcleo é a implementação de uma nova chamada de sistema (*system call*). O problema é que um módulo não consegue implementar novas chamadas de sistema, tarefa que só é possível modificando-se o fonte do núcleo do Linux (aplicando-se um *patch* para realizar as alterações necessárias no núcleo). Como uma das diretivas que guiou o projeto da plataforma foi a de não modificar-se o código do núcleo, a não ser em último caso, decidiu-se implementar um método de controlador de dispositivo no módulo. No nível de usuário, um dispositivo relativo ao módulo foi criado no diretório de dispositivos (*/dev/*), tornando possível um processo de usuário interagir com o módulo através de chamadas aplicadas a esse arquivo de dispositivos.

Abaixo tem-se o segmento do código responsável pelo registro do módulo como um controlador de dispositivo:

```
1 static struct file_operations fops;
2 driver_name="sci_driver0";
3 driver_name[10]+=THIS_NODE_NUM;
4 fops.ioctl = program_ioctl;
5 #ifdef SCI_LOCAL_SIMULATION
6 register_chrdev(THIS_NODE_NUM+DEVICE_NUM_BASE,driver_name,&fops);
7 #else
8 register_chrdev(DEVICE_NUM_BASE,driver_name,&fops);
```

9 #endif

Primeiramente uma variável que contém o nome do dispositivo que será registrado é inicializada (linha 1). Quando o módulo é executado no *SCI*, somente uma instância estará rodando em cada um dos nodos. Sendo assim, o nome escolhido para o dispositivo é "sci_driver0". Como na simulação tem-se o mesmo módulo carregado (com nome diferente) mais de uma vez, pois se simulará vários nodos em uma só máquina, o nome escolhido varia conforme o nodo que está sendo simulado. Para o nodo 0, temos "sci_driver0", para o 1, o nome escolhido é "sci_driver1" e assim por diante (linha 3).

Após isso, o ponteiro `program_ioctl` que aponta para a implementação da rotina `ioctl` no módulo é devidamente registrado na estrutura `fops` que contém a tabela de funções que podem ser implementadas por um controlador (linha 4). Só essa função é implementada por esse módulo, todas as outras que a estrutura `fops` contém apontam para um endereço nulo. Por fim, utilizando-se a chamada `register_chrdev` (linha 6 ou 8) o módulo é registrado como um controlador de dispositivo. Essa chamada recebe como parâmetro o número *major* que identifica o dispositivo, o nome e a tabela de *callbacks*.

Um parênteses sobre os números *major* e *minor* é aqui necessário. Os dispositivos caracteres (ou de bloco) são acessados no Linux como nodos no sistema de arquivos. Os arquivos de dispositivo são especiais e contêm três identificadores: o tipo de dispositivo que o arquivo representa, o número *major* e o número *minor*. O tipo de dispositivo pode ser "b" para dispositivos de bloco e "c" para dispositivos tipo caracter. Já o *major* identifica qual é o controlador associado a determinado arquivo de dispositivo. O *minor* é utilizado somente pelo controlador de dispositivo para diferenciar entre diferentes arquivos que apontam para o mesmo dispositivo. Por exemplo, existe um controlador no sistema para os discos *IDE*. No diretório de dispositivos temos `/dev/hda0`, `/dev/hda1`, ..., que têm o mesmo *major* (3), mas diferente *minor* para que o controlador consiga identificar à qual partição cada um deles se refere.

Assim, no caso do módulo de conexão, o *major* utilizado é lido do arquivo de configuração. Se deseja-se executar a plataforma em uma máquina local, esse número é acrescido do número do nodo que está sendo simulado no momento, pois cada instância do módulo deve registrar um dispositivo com *major* diferente.

Pode-se verificar no arquivo */proc/devices* se o dispositivo foi registrado corretamente. Por exemplo, seja o *major* 100. Executando-se "`cat /proc/devices`" dispara-se o tratador "read" do arquivo "/proc/devices" que, por convenção, retorna a listagem de todos os dispositivos registrados. Como resultado, tem-se "`Character devices:  100 sci_driver0`".

As chamadas que realizam as conexões com os nodos remotos são colocadas dentro da função `program_ioctl` que agora pode ser invocada de um programa no nível de usuário. Para conectar-se com o bloco de memória remota de todos os outros nodos da rede, utiliza-se a seguinte seqüência de chamadas de gerenciamento de memória remoto:

`IcmConnect:` Conecta com um bloco de memória remoto oferecido identificado por `SCINodeID`, `ModuleID` e `ChunkID`.

`IcmMap:` Cria um mapeamento na área de endereçamento físico local do bloco remoto que foi conectado.

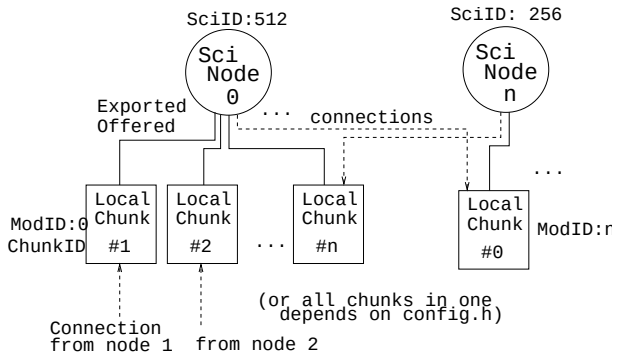

FIGURA 4.14 – Blocos de memória locais e respectivas conexões entre nodos

`IcmGetNumberOfViews:` Retorna o número de sub-blocos contíguos de memória necessários para mapear o bloco.

`IcmGetPhysicalAddressOfView:` Retorna o endereço físico de memória de um dos sub-blocos para torná-lo acessível para leituras/escritas. As leituras/escritas nesse endereço serão automaticamente transformadas em requisições à memória remota tratada pelo *SCI*.

O endereço retornado pela rotina `IcmGetPhysicalAddressOfView` no *SCI* é um endereço físico de memória, enquanto que no módulo de simulação é um endereço lógico. Por esse motivo, quando o *SCI* está sendo utilizado, existe a necessidade de mapear esse endereço para a área de endereçamento virtual, o que não é necessário quando se utiliza o modo simulado. Para fazer-se esse mapeamento, utiliza-se a chamada `ioremap` do núcleo do sistema operacional. Como já visto, essa rotina mapeia um endereço físico em uma área de endereçamento linear virtual acima de `VMALLOC_START`.

Com todos os blocos de memória exportados e conectados por todos os nodos do sistema, o módulo disponibiliza os endereços dos blocos locais e remotos para as camadas superiores da arquitetura na tabela de símbolos globais do núcleo. Uma visão geral dos blocos de memória compartilhados pode ser vista na figura 4.14.

Todas as tarefas realizada pelo módulo descrito acima podem ser resumidas na seguinte seqüência de operação: ao ser instalado, o módulo disponibiliza blocos de memória local para serem conectados por máquinas remotas. Após isso, o usuário, no momento que todos os nodos já estão executado o módulo de conexão, chama uma rotina de usuário em cada nodo que sinaliza o momento de conectar com os blocos de memória disponibilizados pelos outros nodos. Após isso, os endereços são registrados na tabela de símbolos globais do núcleo e o módulo fica inoperante até o usuário resolver retirá-lo do sistema, o que causa uma desconexão dos blocos remotos e o seu término.

O programa de usuário executa as seguintes instruções para disparar a conexão com blocos remotos dentro do módulo:

```
1 aa = open("/dev/sci_driver0",O_RDWR);
2 ioctl(aa,0,0);
```

Esse procedimento pode ser visto na figura 4.15. Após a inicialização do módulo e a execução do programa de usuário no momento adequado, todos os blocos

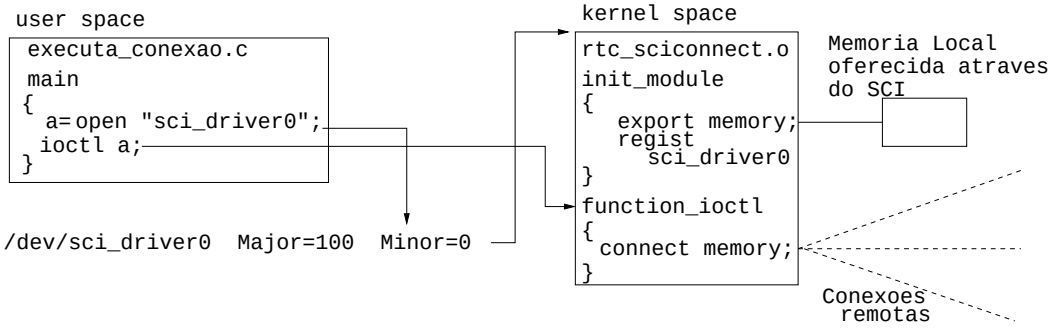

FIGURA 4.15 – Visão esquemática do funcionamento do módulo de conexão

de memória compartilhados estão criados e as interconexões realizadas. Então, os módulos superiores na plataforma podem utilizar essa memória compartilhada para realizar a comunicação necessária.

Por fim, é importante ressaltar que as chamadas da biblioteca *ICM* não podem ser feitas de dentro de uma tarefa de tempo real. Em uma primeira versão esse módulo já estava dentro de tarefas escalonadas pelo *RTAI* o que causou vários problemas. Isso aconteceu porque a biblioteca *ICM* está fortemente ligada com o núcleo do Linux padrão, e dentro de tarefas de tempo real não podemos utilizar todas as chamadas do núcleo.

4.6 Camada de acesso ao meio físico

Módulo de núcleo: `rtc_mac.o`

Fisicamente uma rede baseada em adaptadores *SCI* é do tipo ponto a ponto e não de difusão. Apesar disso, o fenômeno da contenção na transmissão de dados pode acontecer. Isso se deve à arquitetura do adaptador, onde pacotes locais devem esperar por outros que necessitam atravessar o adaptador para atingir o nodo de destino. Devido a isso, uma camada que controle o acesso ao meio físico é necessária.

4.6.1 Objetivos

Os objetivos do módulo responsável por controlar o acesso ao meio físico são:

- Inicializar os serviços de tempo real do núcleo do *RTAI*
- Controlar o acesso ao meio físico da rede *SCI*
- Sincronizar os relógios das máquinas do cluster

Esse módulo é o primeiro a utilizar serviços do *RTAI*, inclusive gerando tarefas de tempo real. Nas próximas seções serão apresentadas as soluções adotadas para atingir os objetivos.

4.6.2 Inicialização do *RTAI*

A primeira tarefa realizada pelo módulo de acesso ao meio físico é a inicialização dos serviços de tempo real do *RTAI*. Isso porque, até nesse momento, o núcleo

de tempo real não foi necessário e permaneceu inativo redirecionando todas as interrupções diretamente para o núcleo padrão do Linux.

A tarefa de inicialização do *RTAI* é executada dentro da função `init_module`. Inicialmente se inicializa as duas tarefas de tempo real que serão executadas:

```
static RT_TASK Synchronization_Task;
static RT_TASK SendMessages_Task;
rt_task_init(&Round_Generation_Task,
     Round_Generation_Thread, 0, 2000, 1, 0, 0);
rt_task_init(&Slot_Generation_Task,
     Slot_Generation_Thread, 0, 2000, 2, 0, 0 );
```

O objetivo dessas *threads* será apresentado posteriormente.

A chamada `rt_task_init` preenche a estrutura do tipo `RT_TASK` com os dados (como endereço do código da *thread*, prioridade, etc) referentes a uma *thread* de tempo real, e também a cria, deixando-a suspensa.

Após a criação das duas *threads* de tempo real que serão executadas, faz-se necessária a inicialização do temporizador que será utilizado pelo *RTAI*. Isso é feito com a chamada `start_rt_timer`.

Finalmente, as duas *threads* podem começar a execução do seu código. Isso é feito utilizando-se a chamada `rt_task_resume` para tarefas não periódicas ou, caso contrário, `rt_task_make_periodic`. Como o período das tarefas não é fixo pelo relógio local da máquina e sim por um sinal de sincronismo que será mostrado nas seções posteriores, a primeira função é utilizada:

```
rt_task_resume(&Round_Generation_Task);
rt_task_resume(&Slot_Generation_Task);
```

4.6.3 Sincronização de relógios

Embora exista um relógio físico em cada nodo do *cluster*, a sincronização entre eles não é um objetivo simples de ser alcançado. Devido ao *drift* intrínseco entre os diversos relógios, mesmo que em um momento t_0 todos os relógios dos nodos na rede sejam sincronizados, em um momento posterior t_1 eles não estarão mais medindo o mesmo tempo. O *drift* acontece porque os diferentes relógios contam o tempo em uma velocidade diferente. Portanto, para mantermos uma certa precisão no tempo, os relógios devem ser atualizados em intervalos regulares de tempo.

Os métodos de sincronização dos relógios podem ser divididos em dois tipos:

Sincronização externa: Para um limite de sincronização $D > 0$, para uma fonte de tempo externa S e para o relógio do nodo j, definido como C_j, temos que: $|S(t) - C_i(t)| < D$ para $i = 1, 2, ..., N$ e todos os tempos t contidos no intervalo de utilização do sistema [COU 2001]. Isso significa que todos os relógios contêm sempre uma hora aproximada de uma *fonte externa de sincronização* S e o limite máximo de divergência é D.

Sincronização interna: Para um limite de sincronização $D > 0$, $|C_i(t) - C_j(t)| < D$ para $i, j = 1, 2, ..., N$ e para todo o intervalo de utilização do sistema. Isso significa que todos os relógios internos da rede têm sempre uma hora aproximada de todos os outros, cujo limite máximo de divergência entre quaisquer dois nodos é D.

Para a implementação do protocolo de acesso ao meio que será discutido a seguir, o modelo de sincronização interna é suficiente.

Para que seja feita a sincronização dos relógios, duas noções de corretude de relógios devem ser levadas em conta:

Monotonicidade: Um relógio sempre deve avançar no tempo. Sendo assim, não se pode alterar o relógio para um tempo anterior.

Limite de *drift*: Os relógios devem ter uma velocidade (*clock rate*) máxima e mínima dentro de um intervalo definido. Isso significa que não se pode dar "pulos" no tempo medido de um relógio. Na sincronização, é necessário alterar-se a velocidade dos diversos relógios dos nodos dentro do intervalo permitido até eles obterem a correta sincronização, sem utilizar atualizações diretas para a hora correta.

Diversos algoritmos de sincronização de relógios dentro de uma rede são apresentados por [COU 2001]. Devido à necessidade de uma quantidade considerável de mensagens e a necessidade de interrupção para sinalizar a chegada de uma nova hora, optou-se por utilizar uma sincronização de relógios combinada com o acesso ao meio físico que será explicada na próxima seção.

4.6.4 Acesso ao meio físico

Conforme já dito no capítulo de introdução, um dos problemas enfrentados pelo *SCI* quando se deseja a transmissão de tráfego de tempo real é a susceptibilidade do protocolo de acesso ao meio físico do hardware a contenções. Esse problema pode ser resolvido pela adequada escolha de protocolos que realizem:

1. Arbitragem do meio físico: a capacidade de transmissão da rede deve ser compartilhada entre os nodos. A estratégia de escolha de qual nodo pode iniciar a colocação de dados no meio físico, em um determinado momento, é tarefa da arbitragem do meio físico.

2. Controle de transmissão: Determina a quantidade de dados que um nodo pode colocar no meio físico após a posse do mesmo (veja capítulo de revisão bibliográfica para maiores detalhes).

O presente módulo implementa a arbitragem do meio físico e o controle de transmissão básico por nodo, já a camada de enlace é responsável pela divisão do tempo de cada nodo pelos processos.

No capítulo de revisão bibliográfica foi apresentada uma classificação dos protocolos de acesso ao meio físico quanto às colisões. Conforme visto, existem dois grandes grupos que são os protocolos de acesso contencioso e os de acesso controlado. Acessos contenciosos não são adequados para comunicações de tempo real devido ao problema da contenção não determinística. Já os de acesso controlado evitam colisão e por isso são apropriados para tráfego de tempo real.

Os protocolos de acesso controlado contam com uma popularidade devido à dificuldade de se fazer uma arbitragem mais precisa do meio físico através da utilização de um algoritmo de escalonamento de tempo real (e.g. *rate monotonic*) para escalonar as mensagens. Isso acontece porque, devido à natureza distribuída do

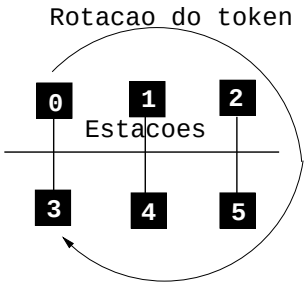

FIGURA 4.16 – Token Bus

acesso ao meio físico, informações globais que permitiriam tais algoritmos serem implementados requerem um grande *overhead* [MUK 99]. Sendo assim, consideraremos nesse trabalho técnicas baseadas na multiplexação em tempo do canal.

Os protocolos considerados na escolha de um árbitro de meio físico juntamente com o controle de transmissão[3] foram o *timed-token* e o *TDMA* (*Time Division Multiple Access*).

Timed-Token O *Timed-Token* é uma técnica na qual as estações do barramento formam um anel lógico. Sendo assim, uma ordem lógica é designada para as estações onde o último membro dessa ordenação tem como próximo elemento o primeiro (veja figura 4.16). Uma lista determina a ordem lógica das estações.

Um pacote de controle conhecido como *token* regula o direito de acesso à rede: quando uma estação recebe o *token*, a ela é garantido o controle do meio físico por um tempo especificado, durante o qual ela pode transmitir um ou mais pacotes. Quando a estação terminou de enviar o tráfego necessário ou quando o tempo expirou, ela passa o *token* para a próxima na seqüência lógica [STA 84].

O protocolo de acesso ao meio físico *timed-token* divide todas as mensagens da rede em duas classes: as mensagens de tempo real e aquelas que não têm limite temporal associado. Cada estação possui uma parte da banda da rede reservada para as mensagens de tempo real. As outras são transmitidas se existe uma disponibilidade de tempo na rede.

Existem ainda no protocolo métodos de recuperação de *token* perdido e de manutenção do anel lógico.

Numa rede do tipo *Timed-Token*, existem dois parâmetros que a configuram, que são o SA_k que determina o tempo máximo que a estação k pode transmitir mensagens de tempo real cada vez que recebe o *token* e o TTRT (*Target Token Rotation Time*) que denota o tempo esperado de rotação do *token*. Sendo assim, a fração da largura de banda total alocada para mensagens de tempo real pela estação k é dada por $\frac{SA_k}{TTRT}$ [LIU 2000].

Uma estação decide se pode transmitir mensagens que não têm restrições temporais baseada no tempo em que o *token* chegou. Este pode chegar cedo ou tarde. Para definir-se o que significa cedo e tarde, seja t_{-1} o tempo de chegada na rodada anterior e t_0 o tempo na rodada atual. Se $t_0 - t_{-1} >= TTRT$, o *token* chegou tarde e não é possível transmitir mensagens que não sejam de tempo real, caso contrário, mensagens sem restrições temporais podem ser transmitidas durante $TTRT - t_0 + t_{-1}$ unidades de tempo. Essa permissão de mensagens sem tempo real faz com que no

[3]Normalmente os protocolos existentes versam conjuntamente sobre as duas tarefas.

pior caso o tempo de rotação do *token* seja $2 \times TTRT$.

Esse protocolo é adaptável para o *SCI* por utilizar um anel lógico. Existem implementações de protocolos utilizando *tokens* em redes fisicamente em forma de anel, como o padrão *Token Ring* que necessita de um anel físico. Embora o *SCI* seja formado por um anel físico, não é possível manipulá-lo no nível de software, sendo que nesse nível só temos acesso à memória compartilhada distribuída.

TDMA (*Time Division Multiple Access*)

O protocolo *TDMA* é um protocolo estático de distribuição do meio de acesso onde o direito de transmissão é controlado pela progressão do tempo. Isso requer uma base de tempo global disponível em todos os nodos. Em um sistema TDMA a capacidade global do meio físico de comunicação é dividida estaticamente em um número de *slots*. Cada *slot* é designado a um determinado nodo da rede. Uma seqüência de *slots* que cobre todos os nodos da rede é conhecida como rodada *TDMA*. Se não houver nada para transmitir, um *frame* vazio é transmitido. Os *rounds* TDMA são configurados por uma tabela e podem ser diferentes. Uma seqüência de todos os possíveis diferentes *rounds* é chamada de ciclo [KOP 97]. Por ser guiado pela progressão do tempo, o *TDMA* é considerado um método *timed-triggered*.

Colisão de mensagens não pode ocorrer no *TDMA* porque cada nodo conhece o momento adequado de colocação das mensagens na rede. Além disso, cada nodo sabe quando existe uma mensagem disponível que necessita ser lida [BUR 2001].

Para a adaptação em uma rede que utiliza adaptadores *SCI* ser possível, é necessária a sincronização dos relógios dos diferentes nodos.

Comparação e escolha

Analisando-se as características dos dois protocolos de acesso ao meio físico apresentados e levando-se em conta a facilidade de adaptação para o *SCI* resolveu-se utilizar o *TDMA*. Isso porque é possível a sua implementação no *SCI* com pequenas modificações e sem a utilização de interrupções. Já no *timed-token*, cada vez que a mensagem especial *token* é passada para a próxima estação no anel lógico, uma interrupção é necessária para avisar o nodo de destino dessa chegada. Sem interrupção, cada máquina teria que ficar verificando a todo instante a chegada do *token* (*pooling*) o que consumiria uma parcela considerável do tempo de CPU, fato indesejável. Outra vantagem do *TDMA* é que devido a sua alocação estática da banda da rede, o *jitter* de transmissão é menor.

Como desvantagem da escolha, pode-se citar que existe a possibilidade de um menor aproveitamento da banda física se nodos que alocaram parte da largura de banda não a utilizarem. O algoritmo *timed-token* pode ser considerado do tipo guloso, pois quando um determinado nodo não utiliza toda a largura de banda destinada a ele em um determinado ciclo, o próximo pode utilizar esse espaço para mandar mensagens não-RT. Já no *TDMA* isso não ocorre, configurando um menor aproveitamento da rede nesses casos. Além disso, o *TDMA* carece de uma maior flexibilidade, visto que o tamanho e a designação de cada *slot* são pré-determinadas (estáticas).

No módulo de acesso ao meio físico, decidiu-se oferecer a divisão do tempo em rodadas e *slots TDMA*. A alocação de um dos *slots* para determinado nodo ou mesmo processo comunicante foi deixada como tarefa para a camada superior.

Implementação

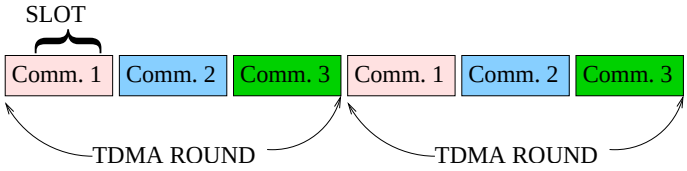

FIGURA 4.17 – Protocolo *TDMA* implementado

No protocolo implementado, as diversas rodadas do protocolo *TDMA* são sempre iguais, sendo então que o tamanho da rodada é igual ao tamanho do ciclo. Na figura 4.17 pode-se ver duas rodadas *TDMA* com algumas comunicações designadas para cada *slot*. Note que em cada *slot* do *TDMA* uma estação terá direito de escrever e outras (que não estiverem sendo escritas pela estação que tem o direito no *slot* atual) poderão ler mensagens deixadas em *slot* anteriores. Como foi dito, essa alocação de estação/*slot* é tarefa da camada de enlace que será explorada posteriormente.

Devido ao fato do *TDMA* ser baseado na progressão do tempo físico, uma base de tempo global é necessária. Para obter essa base decidiu-se sincronizar os relógios no fim de cada rodada *TDMA*. Essa base de tempo sincronizada é mantida somente dentro desse módulo. O resto do sistema utiliza o relógio normal do hardware, onde não se faz nenhuma alteração. Sendo assim, para as aplicações ou mesmo outros módulos do sistema, a sincronização de relógios não é observável.

A combinação desses dois fatores (sincronização no fim da rodada e o confinamento da sincronização) permitiu um relaxamento no parâmetro de corretude anteriormente apresentado chamado de *limite de drift*. Como o relógio é sincronizado no fim da rodada, saltos são permitidos, pois os novos limites de tempo da próxima rodada são calculados já se levando em conta o novo tempo. Com o confinamento evita-se que os saltos, que são altamente indesejáveis e podem causar incorretude temporal de outras aplicações, sejam espalhados para outras áreas do sistema.

É importante observar-se que uma sincronização geral de tempo de todo o sistema pode ser implementada em uma versão futura do sistema com a chamada de sistema `adjtimex`, que recebe como entrada o tempo correto atual e modifica o relógio variando levemente sua velocidade. Isso é feito ajustando levemente o número de microsegundos adicionado à variável `xtime.tv_usec` a cada *tick* do sistema. Essa variável mantém o valor atual do relógio do sistema. Esse método permite que aplicações Linux sejam executadas em um ambiente distribuído com relógio sincronizado. Já os *timers* relativo ao *RTAI* não são alterados nesse caso.

A sincronização do relógio dentro do módulo de acesso ao meio físico é feita através de uma mensagem especial enviada por um nodo do sistema previamente escolhido, que será chamado de *mestre*, para todos os outros nodos, chamados nesse contexto de *escravos*. Essa mensagem indica o início de uma nova rodada *TDMA*. A figura 4.18 apresenta uma rodada em um sistema com 4 nodos e 5 *slots* de tempo com a devida sincronização no final da rodada.

Como pode ser visto na figura, o início de cada rodada é indicado pelo mestre através de uma mensagem especial de sincronismo. Após o recebimento dessa mensagem, os diversos nodos começam a contar o tempo referente a cada um dos *slots*. No momento que um novo *slot* começa, esse fato é indicado à camada superior para ela tomar as ações adequadas para o *slot*, como por exemplo mandar pacotes ou receber, dependendo da ação atribuída a cada nodo nesse *slot*. Ao chegar o

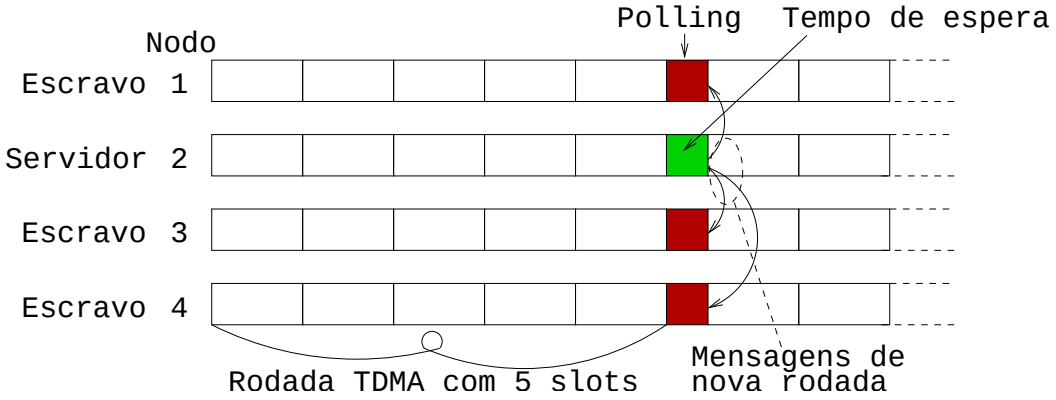

FIGURA 4.18 – Exemplo de ciclo *TDMA* com sincronização

fim do último *slot* de tempo, todos os escravos entram em um ciclo de verificação para notarem a chegada do sinal (mensagem) de sincronismo. Já o mestre aguarda um período de tempo para assegurar que todos os escravos, independentemente da pequena diferença entre os tempos dos temporizadores que porventura possa ter se configurado, já estejam no ciclo de espera do sinal de sincronismo. Após isso, a mensagem especial de sincronismo é enviada. Assim, uma nova rodada é iniciada com os diversos temporizadores de *slots* divergindo em um tempo máximo determinado pelo atraso e o *jitter* da mensagem de sincronismo.

A metodologia de sincronismo apresentada é similar ao sincronismo por barreira entre diversos processos. Nesse tipo de sincronismo existe um determinado ponto no programa que os diversos processos devem esperar até que todos os outros cheguem ao mesmo [AND 91]. No caso da sincronização efetuada pela camada MAC aqui descrita, todos os processos devem esperar em um certo ponto pela chegada do mestre, que devido a uma espera acrescida no fim de sua computação sempre será o último a chegar. Isso ocorre se todos os relógios contiverem um *drift* máximo limitado. Como pode-se ver, os relógios reais da máquina não são realmente sincronizados, e sim o início da contagem de tempo dos diversos temporizadores dos módulos *MAC* nos diversos nodos. Além disso, o sincronismo é combinado dentro do próprio método de acesso ao meio físico *TDMA*.

Uma fórmula aproximada do tempo de espera mínimo do mestre é:

$$Wt = d_{max} + max_{j=1,\ldots,n} drift_m(j) + r + s \tag{4.1}$$

Onde:

Wt: Tempo de espera mínimo

d_{max}: Tempo máximo de entrega da mensagem de sincronismo

$drift_m(j)$: *Drift* do relógio do nodo j em relação ao mestre

r: Tempo máximo de percepção do recebimento do sinal pelo escravo

s: Tempo máximo de escrita do mestre na memória compartilhada

A equação 4.1 diz que o tempo mínimo de espera será o tempo (atraso) máximo de entrega das mensagens somada ao maior *drift* possível entre o relógio de um nodo

e o do mestre. A isso ainda é adicionado o tempo de escrita da mensagem pelo mestre e o tempo de percepção da mensagem pelo escravo.

A implementação do módulo de acesso ao meio utiliza duas *threads* de tempo real. A primeira é responsável pela implementação da barreira de sincronismo, já a segunda implementa a divisão temporal de uma rodada em várias fatias (ou *slots*). As duas *threads* são nomeadas como *thread* de geração da rodada e *thread* de geração dos *slots*

O código da *thread* de geração das rodadas contém diferenças entre o mestre e escravo. Para verificar se a *thread* está rodando em um nodo que é mestre ou escravo, utiliza-se duas constantes fornecidas pelo arquivo de configuração:

`THIS_NODE_NUM`**:** Constante com o identificador do nodo local

`MASTER_NODE_NUM`**:** Constante com o identificador do mestre

Através da comparação dessas constantes pode-se diferenciar o código dos escravos e do mestre. Na figura 4.19 é apresentado um segmento do código do mestre responsável pelo envio da mensagem de sincronismo. Como pode-se ver, esse código é executado em laço infinito.

A primeira tarefa do mestre é enviar o sinal de sincronismo para iniciar uma nova rodada. Mas antes de enviar esse sinal, o mestre deve esperar que os escravos estejam prontos para recebê-lo. Isso é feito utilizando-se a instrução `rt_sleep` do *RTAI* (linha 3). Assim, garante-se que os escravos já estejam verificando o recebimento do sincronismo no momento em que o mestre o envia. Esse tempo de espera foi calculado considerando-se:

$d_{max} = 5\mu s$**:** O manual do *SCI* informa uma latência máxima de transmissão de $5\mu s$. Como o sinal enviado é de somente 1 byte, considerou-se aproximadamente igual ao tempo de entrega dessa byte.

$max_{j=1,\ldots,n} drift_m(j) = 15\mu s$**:** Para chegar-se a esse valor, considerou-se um *drift* máximo de meio segundo dentro do período de 1h. Além disso, o maior ciclo possível para o *TDMA* foi definido em $0,1s$. Sendo assim, $max_{j=1,\ldots,n} drift_m(j) = \frac{0,5s \times 0,1s}{3600s} = 14\mu s \approx 15\mu s$.

$r = 1\mu s$**:** Um acesso de memória ($60ns$) para a leitura do sinal mais o *delay* entre acessos de $400ns$ (que será explicado posteriormente). Como esse tempo ficou na ordem das centenas de nanosegundos, considerou-se $1\mu s$.

$s = 6\mu s$**:** Para um número máximo de 100 nodos, temos um laço de 100 execuções onde um valor é escrito em uma posição de memória (envio do sinal para cada escravo). Isso significa $60ns \times 100$ o que resulta em $6\mu s$.

Como resultado, chegou-se no valor de $27\mu s$ onde se acrescentou uma margem de segurança chegando-se a $35\mu s$ que foi utilizado na chamada do `rt_sleep`. Após aguardar por esse período, o mestre envia um dado sinalizador de início de ciclo para cada escravo. No paradigma do *SCI* isso significa escrever em um bloco de memória compartilhado previamente com cada máquina pelo módulo de conexão. Apresentou-se na seção 4.5.2 a configuração de blocos compartilhados de memória utilizados na plataforma de comunicação. Essa configuração não está completa. Além do apresentado, cada nodo exporta também uma região de memória utilizada

```
1 while(1)
2 {
3 rt_sleep(nano2count(35000)); /* Espera 35us */
4 for (c=0;c<NUM_NODES;c++)
5 if (c!=MASTER_NODE_NUM)
6 { /* Escreve na posição apropriada de um */
7 /* escravo o sinal */
8 *((char *)RemoteBaseBarrier[c])=S;
9 }
10 (...)
11 /* Libera a thread de geração de slots */
12 rt_sem_signal (&new_round_semaph);
13
14 /* Aguarda pelo próximo periodo */
15 rt_sleep(nano2count(CICLE_PERIOD));
16
17 /* Atualiza o contador de ciclos */
18 CicleCounter++;
19 }
```

FIGURA 4.19 – Código simplificado da *thread* de geração de rodadas do mestre

exclusivamente na sincronização. Os endereços dos blocos disponibilizados pelos escravos ficam armazenados no vetor `RemoteBaseBarrier`. Utilizando esses endereços o mestre escreve na memória dos escravos indicando o início de um novo ciclo (linha 8). Após o recebimento desse sinal, todos os escravos iniciam sincronizadamente um novo ciclo.

Ao iniciar um novo ciclo, a *thread* de geração da rodada sinaliza semáforo `new_round_semaph` que acorda a *thread* de geração dos *slots*. Feito isso, a *thread* de geração de rodadas dorme em um temporizador do *RTAI* até o início de um novo ciclo, onde tudo recomeça.

O código do escravo é semelhante ao do mestre. A diferença é que, no lugar de mandar o sincronismo, ele fica verificando o seu recebimento. Um segmento do código pode ser visto na figura 4.20. É importante observar-se que enquanto o escravo está lendo continuamente a memória compartilhada para verificar a chegada do sinal, é necessário a intercalação de pequenas paradas. Isso porque com a leitura contínua da posição de memória, o adaptador *SCI* não encontra oportunidade para escrever nessa posição visto que o barramento da memória está sempre ocupado. Por isso, pode-se ver na linha 7 uma instrução de `wait` que é responsável por liberar o barramento.

Logo após o recebimento, como pode ser visto na linha 12, o valor da posição de memória onde o sinal foi recebido é alterado para, na próxima rodada, aguardar novamente o recebimento do sinal de sincronismo. Desse ponto em diante, as tarefas executadas pelos escravos são as mesmas do mestre.

Como já dito, o semáforo `new_round_semaph` é sinalizado a cada início de rodada. Esse semáforo, que na inicialização do módulo contém o valor 0, controla a *thread* de geração dos *slots*. Ela é responsável por dividir a rodada no número de

```
1 while (1)
2 {
3 /* Aguarda nova rodada */
4 while (*((char *)LocalBaseBarrier)!=S)
5 {
6 /* Libera o barramento para o SCI (400ns) */
7 rt_busy_sleep(400);
8 }
9
10 /* Sobreescreve o sinal recebido para preparar-se */
11 /* para o próximo ciclo */
12 *((char *)LocalBaseBarrier)=R;
13 (...)
14 /* Libera a thread de geração de slots */
15 rt_sem_signal (&new_round_semaph);
16
17 /* Aguarda pelo próximo período (rodada) */
18 rt_sleep(nano2count(CICLE_PERIOD));
19
20 /* Atualiza contador de rodadas */
21 CicleCounter++;
22 }
```

FIGURA 4.20 – Código simplificado da *thread* de geração de rodadas do escravo

slots apropriado. Para isso utiliza temporizadores oferecidos pelo núcleo do sistema operacional *RTAI*.

Podemos ver na figura 4.21 as principais operações realizadas pela *thread* de geração dos *slots*. Inicialmente a *thread* está esperando na fila do semáforo `new_round_semph`. Como já visto anteriormente, esse semáforo indica o início de uma nova rodada. Após a sinalização do semáforo, é iniciada a geração dos *slots*. No início de cada *slot* gerado, as rotinas de escrita e de leitura no meio físico da rede são chamadas. Essas rotinas ficam na camada de enlace da arquitetura. Logo após, como pode ser visto na linha 16, a *thread* é posta para dormir em um temporizador do *RTAI* até a chegada do próximo *slot*. Sendo assim, essa *thread* é responsável pela indicação do momento de escrita e leitura do meio físico da rede pela camada de enlace. Isso caracteriza a arbitragem do meio físico. Já o controle de transmissão, ou seja, quantas mensagens podem ser postas na rede, é realizado pela rotina `rtc_link_WriteMessages` que fica na camada de enlace.

Conforme visto nesta seção, o módulo de MAC implementa o protocolo TDMA com rodadas divididas em *slots*. Para isso, utiliza duas *threads* de tempo real que são responsáveis respectivamente pela geração das rodadas e, dentro de cada rodada, pela geração dos *slots*. Em cada *slot* somente uma máquina terá direito de colocar os dados no meio físico de comunicação. Juntamente com a geração das rodadas uma sincronização de tempo é feita utilizando-se para isso uma barreira no fim da rodada *TDMA*. Já o controle de qual nodo tem direito de trasmissão em cada *slot* e quantas mensagens podem ser postas nesse intervalo de tempo é uma funcionalidade delegada à camada de enlace que será vista na próxima seção.

```
1  while (1)
2  {
3  /* Aguarda pela sinalização do semáforo */
4  rt_sem_wait(&new_round_semaph);
5  for (i=0;i<NUM_SLOTS;i++)
6  {
7  old = rt_get_time();
8  rtc_link_WriteMessages (...);
9  rtc_link_ReadMessages (...);
10 (...)
11 now = rt_get_time();
12 time = (CICLE_PERIOD / NUM_SLOTS -
13       (count2nano(old) - count2nano(now)));
14
15 /* Espera pelo início do proximo slot */
16 rt_sleep(nano2count(time));
17 }
18 }
```

FIGURA 4.21 – Segmento do código da *thread* de divisão da rodada em *slots* de tempo

Na figura 4.22 tem-se uma visão geral do funcionamento da camada de acesso ao meio. As duas *threads* que a compõem são representadas como retângulos. O algoritmo executado por cada uma esta sumarizado dentro dos retângulos. Esses dois algoritmos são executados dentro de um laço infinito. Abaixo tem-se um diagrama de tempo do *TDMA* gerado com a respectiva relação com o código das *threads*.

As duas *threads* de tempo real devem ter prioridade máxima dentro do sistema, visto que nenhuma tarefa pode as preemptar e assim comprometer a correta temporização do *TDMA*.

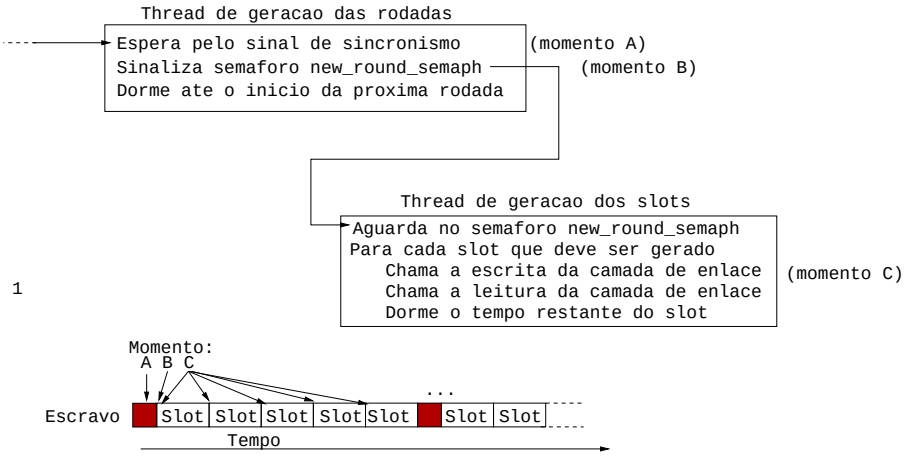

FIGURA 4.22 – Visão geral do módulo MAC

4.7 Camada de gerenciamento de memória

Módulo de núcleo: `rtc_mem.o`

Na seção 3.2.5 do capítulo 3 foram apresentadas algumas restrições nas ações que podem ser executadas dentro de uma tarefa colocada em alguma fila de tarefas do sistema. Dentre as restrições apresentadas está a impossibilidade de alocação de memória dinâmica utilizando as chamadas do núcleo do Linux como por exemplo `kmalloc`. Isso se dá pelo fato dessas chamadas utilizarem semáforos para protegerem seções críticas.

Embora os processos dentro do núcleo do sistema operacional normalmente não sejam preemptáveis, interrupções ou tratamento de exceção podem interromper um processo dentro do núcleo. Por isso, chamadas de alocação de memória dinâmica dentro do núcleo precisam ser protegidas por semáforos. Além disso, as chamadas de alocação dinâmica de memória do núcleo de tempo real *RTAI* também utilizam semáforos para evitar condições de corrida.

Como será visto em seção posterior, existem no módulo que implementa um controlador de dispositivo que gerencia as mensagens sem restrições temporais alguns caminhos que são executados em tempo de interrupção e necessitam alocar memória dinamicamente. Como tanto as chamadas do núcleo normal do Linux quanto as do *RTAI* são proibidas de serem utilizadas nesse contexto, tornou-se necessária a implementação de um módulo de software que gerencie alocações de memória sem a utilização de semáforos. Esse módulo será descrito nessa seção.

4.7.1 Objetivos

Os objetivos do módulo de gerenciamento de memória são:

- Fornecer primitivas de alocação dinâmica de memória que, na sua implementação, não utilizem semáforos como meio de obter exclusão mútua
- Gerenciar listas encadeadas de pacotes

Esse módulo, além de gerenciar a alocação dinâmica de memória, também é responsável por fornecer primitivas que permitam a inserção de pacotes em listas duplamente encadeadas. As mensagens que devem ser enviadas, acrescidas de informações necessárias para sua correta manipulação e transmissão pela arquitetura, formam os pacotes.

Toda a mensagem que é enviada através da arquitetura de comunicação proposta é traduzida para um pacote interno da arquitetura que será transmitido através do adaptador *SCI*. A estrutura dos pacotes será mostrada em seção posterior.

4.7.2 Alocação de memória

Ao contrário das primitivas encontradas dentro do núcleo do Linux como `kmalloc` e dentro do núcleo do *RTAI* como `rt_kmalloc` que alocam blocos de tamanho arbitrário de memória, a alocação de memória requerida pela plataforma tem uma estrutura e tamanho definidos. A unidade que é alocada por outros módulos da plataforma é o pacote. Isso porque a única necessidade de alocação dinâmica é a criação dos pacotes que serão enviados pela rede. Essa criação será detalhada posteriormente. Sendo assim, toda alocação de memória feita pela plataforma sempre retornará um bloco de memória do tipo pacote de tamanho fixo (em outras

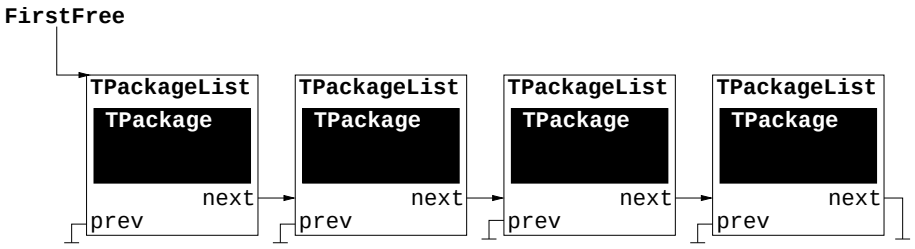

FIGURA 4.23 – Estrutura utilizada no gerenciamento dos blocos de memória livres

palavras, uma instância na memória da estrutura do pacote, que será tratada de agora em diante como alocação de um pacote).

Esse tipo de alocação de memória, que tem sempre tamanho fixo, evita dois problemas conhecidos da alocação de memória: a fragmentação externa e interna. Para alocar páginas contínuas e evitar o problema da fragmentação externa o núcleo utiliza o sistema conhecido como alocação *buddy* e para evitar a fragmentação interna o algoritmo *slab*. Com a alocação somente de blocos de tamanho fixo, essas técnicas que contêm uma complexidade considerável podem ser evitadas sem a ocorrência dos dois problemas citados acima.

As primitivas disponibilizadas para a alocação e liberação de memória são:

```
TPackage *rtc_alloc_package (void);
void rtc_free_package (TPackage *package);
```

onde `rtc_alloc_package` é responsável pela alocação de um pacote cujo endereço é retornado pela função e `rtc_free_package` realiza a liberação de pacote alocado anteriormente.

Internamente ao módulo, uma lista de pacotes livres que podem ser alocados é mantida. A estrutura utilizada para o gerenciamento dessa lista é formada pelo tipo abstrato `TPackageList` que pode ser visto a seguir:

```
typedef struct BTPackageList TPackageList;
struct BPackageList
{
TPackage Package;
TPackageList *next;
TPackageList *prev;
};
```

`TPackage` é um tipo abstrato de dado que corresponde a um pacote que será transmitido pela rede e os ponteiros `next` (próximo na lista) e `prev` (anterior na lista) são utilizados tanto para manter a lista de pacotes livres quanto para outras listas do sistema que são gerenciadas pelas primitivas mostradas na próxima seção. A função de alocação de blocos retira um instância do tipo `TPackageList` da lista de blocos livres e retorna o ponteiro para o elemento `Package` que é um pacote que pode agora ser utilizado. Um exemplo de lista de pacotes livres é mostrado na figura 4.23.

Uma lista simples é suficiente para manter os pacotes livres, portanto o ponteiro `prev` não é utilizado. Ele existe porque o mesmo tipo de dado (`TPackageList`) será utilizado para manter outras listas do sistema que necessitam ser duplamente encadeadas.

Todos os blocos que serão utilizados na composição da lista de blocos livres são alocados de maneira estática pelo módulo. Essa alocação é feita na forma de um vetor declarado no início do módulo. Na função de inicialização, os elementos desse vetor que são do tipo `TPackageList` são incluídos na lista de pacotes livres que inicialmente está vazia. Após essa operação, as primitivas de alocação e liberação de instâncias de pacotes já podem ser chamadas.

A implementação das primitivas de alocação e liberação de pacotes necessita de exclusão mútua. Isso ocorre porque durante uma alocação, uma interrupção, por exemplo, pode ocorrer, e o processador executar outra parte do núcleo. Essa outra parte pode chamar novamente a rotina de alocação de pacotes. Como essa rotina altera a lista de pacotes livres que é global e única no sistema, pode-se nesse caso chegar a um estado inválido dos diversos ponteiros.

A exclusão mútua dentro do núcleo *RTAI*, e em conseqüência do Linux, pode ser obtida de várias formas:

Desabilitação das Interrupções: Através da chamada `rt_global_cli` as interrupções são desabilitadas, sendo que um caminho dentro do núcleo não pode ser interrompido. Se a máquina for multiprocessada, essa chamada adquire o *spin lock* para a CPU que a chamou de modo que todas as outras CPUs sincronizadas por esse método são bloqueadas [BIA 2000].

Semáforo: Conforme visto no capítulo 3, o núcleo do *RTAI* provê um conjunto de chamadas que permitem a criação e uso de semáforos para sincronização e comunicação entre as tarefas de tempo real. Esses semáforos podem ser do tipo contador, do tipo binário ou do tipo recurso.

Embora o núcleo do Linux forneça também uma gama de primitivas que permitem o estabelecimento de exclusão mútua, como operações atômicas, desabilitação das interrupções, semáforos e *spin-locks* no caso de ambientes multiprocessados, essas operações não podem ser utilizadas por tarefas de tempo real. Isso porque como já foi visto, o *RTAI* pode sempre interromper o Linux e portanto não há garantias que essas primitivas irão funcionar dentro de tarefas de tempo real.

Como as primitivas providas pelo módulo de gerenciamento de memória são chamadas de dentro de tarefas de tempo real, um dos dois mecanismos oferecidos pelo *RTAI* deve ser utilizado. A escolha possível recai sobre as instruções que desabilitam/habilitam as interrupções, pois, como já foi dito, semáforos não podem ser utilizados em tempo de interrupção e algumas rotinas que necessitam da alocação dinâmica de pacotes rodam em tempo de interrupção.

Como exemplo, a rotina que aloca os pacotes é mostrada na figura 4.24. Nas linhas 7 e 8 as *flags* do processador são salvas e após a *flag interrupt enable* é zerada, ou seja as interrupções são desabilitadas. No final da rotina, a chamada `rt_global_restore_flags` é utilizada no lugar da `rt_global_sti`. Isso porque, as interrupções poderiam já estar desabilitadas quando a chamada `rt_global_cli` foi invocada. Salvando-se as *flags* e após restaurando se garante que o estado original será restaurado, seja ele com interrupções desabilitadas ou não.

O gerenciamento da memória livre é feito através da lista de pacotes livres cujo início é dado pela variável `first_free`. Já a variável `FreePackagesCounter` contém um contador de pacotes disponíveis para a alocação dinâmica.

A chamada `rtc_free_package` funciona de modo semelhante, só que ao invés de retirar da lista recoloca um pacote livre e incrementa o contador de pacotes livres.

```
1 TPackage *rtc_alloc_package (void)
2 {
3 TPackage *buf;
4 unsigned long flags;
5 rt_global_save_flags(&flags);
6 rt_global_cli();
7 if (first_free==NULL)
8 {
9 rt_global_restore_flags(flags);
10 return NULL;
11 }
12
13 buf = &(first_free->Package);
14 first_free = first_free->next;
15 FreePackagesCounter--;
16 rt_global_restore_flags(flags);
17 return buf;
18 }
```

FIGURA 4.24 – Código responsável pela alocação de pacotes livres

É importante ressaltar que o primeiro campo do tipo `TPackageList` precisa ser o pacote. Isso porque, na hora de retornar o bloco para ser incluído novamente na lista de pacotes livres, o seu endereço é utilizado como endereço do `TPackageList` através de uma conversão, para ser novamente inserido na lista de pacotes livres. Isso é possível porque o C permite um truque: converter o tipo `TPackage` para `TPackageList`. Isso torna possível que um bloco de memória seja visto como um pacote pelos outros módulos e, dentro do módulo de gerenciamento de memória, possa ser visto como um `TPackageList` com dois campos a mais necessários para a manutenção das listas.

Conforme mostrado, com a utilização das rotinas de desabilitação de interrupções do núcleo do *RTAI* foi possível construir um gerenciador simples que permite a alocação de pacotes pelos outros módulos do *RTAI*. Embora essa memória seja alocada estaticamente em um vetor na inclusão do módulo no núcleo, ela é gerenciada e fornecida para os outros módulos da arquitetura sob demanda e de forma otimizada, pois o gerenciamento requer uma simples fila, no lugar de algoritmos complexos necessários para lidar com o problema da fragmentação quando são requeridos blocos de tamanho variável.

4.7.3 Gerenciamento de Listas de Pacotes

Devido à necessidade de se manter múltiplas filas com pacotes a enviar, a receber, etc, tornou-se necessário um gerenciamento centralizado dessas filas. Esse gerenciamento simplificou os diversos módulos que acessavam essas filas. Esses módulos são o enlace de dados, controlador não-RT e a API, que serão vistos nas próximas seções. Devido à possibilidade dos vários módulos necessitarem a inclusão ou exclusão de um elemento de uma determinada fila concorrentemente, a exclusão mútua também foi necessária na implementação das primitivas de manipulação das filas. As primitivas oferecidas são:

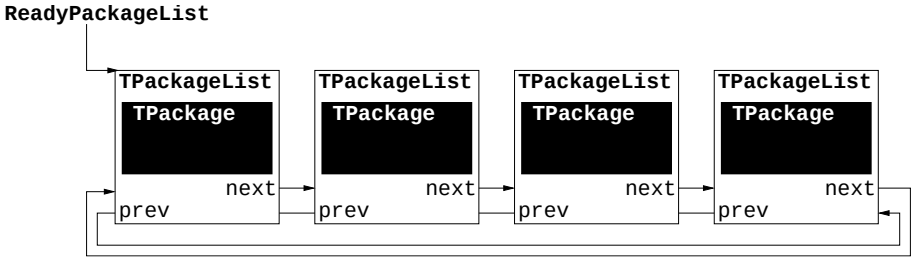

FIGURA 4.25 – Exemplo de fila de pacotes livres gerenciada pelo módulo

```
void InsertToPackageList (TPackageList *packageListHeader,
                          TPackage *pack, int pos)
TPackage *GetFromPackageList(TPackageList
                             *packageListHeader)
```

A primitiva de inserção insere o pacote `pack` na lista de pacotes apontada por `packageListHeader`. O parâmetro `pos` determina a posição onde ele será inserido (início ou fim da lista). Já a primitiva `GetFromPackageList` retorna o último pacote da lista apontada por `packageListHeader` ou *NULL* caso a lista esteja vazia. Na figura 4.25 pode-se ver uma fila. Diferentemente da fila que contém pacotes livres, essas filas necessitam ser duplamente encadeadas pois é necessária a inserção de pacotes tanto no início quanto no fim. Os ponteiros `prev` e `next` estão presentes no tipo `TPackageList` para permitirem sua inserção/remoção da lista.

Novamente o truque do C de permitir a conversão do tipo `TPackage` para `TPackageList` permite que um bloco de memória seja visto como um `TPackage` externamente e como `TPackageList` internamente.

As primitivas `rt_global_save_flags, rt_global_cli e rt_global_restore_flags` foram novamente utilizadas para garantir exclusão mútua.

As primitivas providas para o gerenciamento de filas permitem aos outros módulos da plataforma manterem diferentes filas de pacotes e inserirem/removerem elementos de forma simplificada e com exclusão mútua. A exclusão mútua utilizando a técnica de desabilitar as interrupções não é recomendada para códigos muito longos pois pode conter alguma tarefa de alta prioridade em uma chamada de baixa prioridade. Contudo, as rotinas protegidas por esse método de exclusão mútua no corrente módulo são de execução rápida não causando grandes alterações no escalonamento ou inversões de prioridade ilimitadas.

4.8 Camada de Enlace

Módulo de núcleo: ***rtc_link.o***

No modelo OSI, a camada de enlace tem atribuições como oferecer uma interface de serviço para as camadas superiores, determinar como os dados serão agrupados e transmitidos (enquadramento) e controlar o fluxo (para evitar que receptores lentos sejam atropelados por transmissores rápidos).

Na plataforma proposta muitas dessas tarefas também são realizadas pela camada de enlace. A unidade básica de transmissão é o pacote, que deve ser colocado

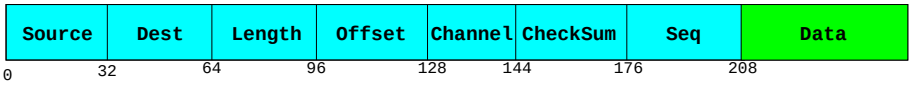

FIGURA 4.26 – Estrutura de um pacote

no meio físico pela camada de enlace. Além disso, como será visto, também é a camada de enlace da arquitetura que controla os limites de transmissão em cada *slot* do *TDMA*. Portanto o módulo descrito nessa seção trabalha em estreita cooperação com a camada de acesso ao meio descrita em seção anterior.

4.8.1 Objetivos

Os objetivos do módulo que implementa a camada de enlace são:

- Gerenciamento dos canais
- Colocar os pacotes no canal de comunicação
- Receber os pacotes vindos de outros nodos
- Detectar erros e controlar o fluxo

A unidade básica de transmissão utilizada é o pacote que sempre pertence a algum canal de comunicação. O conceito de canal foi apresentado no início do capítulo e representa uma conexão simplex fim-a-fim entre dois processos, um emissor e outro receptor [ARV 91].

4.8.2 Pacote RTC

Antes de iniciar a descrição da camada de enlace da arquitetura é importante uma descrição mais detalhada do pacote de dados utilizado na plataforma. A estrutura do pacote pode ser vista na figura 4.26. Como já foi dito, uma mensagem que uma tarefa deseja mandar acrescida de informações relativas a sua entrega formam um pacote RTC. Na plataforma, tanto as mensagens como os pacotes têm um tamanho máximo definido. Os pacotes RTC serão denominados simplesmente de "pacotes".

Os campos presentes no pacote são:

***Source*:** Identificador do nodo emissor da mensagem

***Dest*:** Identificador do nodo receptor da mensagem

***Length*:** Tamanho da mensagem (ou seja, tamanho da área de dados)

***Offset*:** Campo utilizado para remontagem de um pacote quando é necessária sua divisão

***Channel*:** Identificador do canal o qual pertence o pacote

***Checksum*:** Informação utilizada na verificação de erros presentes no pacote

***Seq*:** Número de seqüência

***Data*:** Mensagem propriamente dita (dados úteis do pacote)

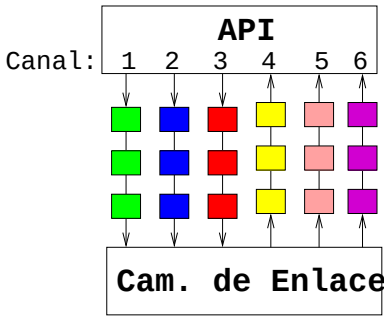

FIGURA 4.27 – Comunicação entre a API e a camada de enlace do sistema

4.8.3 Visão Geral

A principal tarefa da camada de enlace é receber os pacotes da camada que implementa a API do sistema e colocá-los no meio físico da rede. Além disso, pacotes provenientes de outros nodos devem ser lidos e enviados para a API. A comunicação entre as duas camadas é feita através de múltiplas listas, uma para cada canal. Assim, no momento que a aplicação envia uma mensagem através da API, essa mensagem é transformada em um pacote e este é colocado na lista correspondente ao canal que ele pertence. Na figura 4.27 podem ser vistas diversas filas com pacotes. Cada fila pertence a um canal e uma fila pode ser de entrada ou saída (em relação à camada de enlace). A inclusão de elementos na fila é feita utilizando as primitivas apresentadas na seção 4.7.

A camada de enlace é responsável por receber os pacotes das listas de entrada e colocá-los no meio de comunicação, assim como retirar pacotes que chegaram pelo adaptador *SCI* e colocá-los na devida lista de saída.

4.8.4 Gerenciamento dos Canais

A implementação de canais de comunicação de tempo real tem como principal objetivo o gerenciamento da largura de rede com a apropriada reserva para determinada comunicação. Assim, antes de receber pacotes para serem enviados por determinado canal, esse canal deve ser registrado pela API na camada de enlace. Para isso, a camada de enlace oferece duas primitivas que permitem o registro e a liberação de um canal de comunicação:

```
void rtc_link_RegistConnection ( int Id, int Slot,
                                 int BandWidth)
void rtc_link_UnRegistConnection (int Id,int Slot)
```

A primitiva `rtc_link_RegistConnection` permite o registro do canal número `Id` que enviará mensagens no *slot* de número `Slot` e a transmissão desse canal será de `BandWidth` bytes para cada rodada do TDMA. Já a primitiva `rtc_link_UnRegistConnection` é responsável pelo cancelamento de um canal previamente registrado. Uma tabela contendo os canais, seu respectivo *slot* e a banda a ele destinada é mantida pela camada de enlace. Note que somente canais de entrada são registrados com o uso das duas primitivas acima descritas.

A camada de enlace é responsável por retirar os pacotes dos canais registrados e colocá-los na rede de acordo com a reserva de banda feita na ocasião do registro do

TABELA 4.2 – Possível tabela de alocação de envio de canais em slots

Nodo 1				
Capacidade total de um *slot*: 300 bytes				
Canal	*Slot*	Limite de banda	Origem	Destino
3	2	100 bytes/rodada	**1**	5
5	2	180 bytes/rodada	**1**	3
Total:		280 bytes/rodada		
8	3	100 bytes/rodada	**1**	4
Total:		100 bytes/rodada		

canal. Todos os canais registrados em determinado nodo da rede têm como origem o próprio nodo. Como um canal só pode ter um nodo de origem, dois nodos nunca serão responsáveis por colocar na rede pacotes de um mesmo canal.

Os canais de saída não precisam ser registrados. Na ocasião de chegada de um pacote pela interface de rede, o destino é verificado e ele é colocado na lista de saída apropriada.

Para cada *slot* TDMA é definido um máximo de dados que podem ser transmitidos sem comprometer os demais. Como em cada *slot* somente um nodo pode ter canais registrados para utilizá-lo e a soma desses canais não pode ultrapassar o limite do *slot*, é garantido que outros *slots* não serão afetados por essa transmissão.

Um exemplo de uma possível alocação de canais dentro dos *slots* pode ser visto na Tabela 4.2. Nesse exemplo, cada *slot* pode enviar 300 bytes de dados a cada rodada. No nodo 1 existem três canais: o canal 3 cuja origem é no nodo 1 e o destino no 5, o canal 5 cujo destino é o nodo 3 e o canal 8 que tem como destino o nodo 4. Esses canais, pelo requisito de banda mostrado na tabela, não conseguem ser comportados por apenas um *slot*. Assim, dois *slots* são disponibilizados para o nodo 1 enviar seus canais. Obviamente que o *slot* 3, no exemplo dado, terá 200 bytes de sua largura de banda desperdiçada pela inexistência de canais no nodo 1 que possam utilizá-los. Cada nodo possui uma tabela como essa para guiar a colocação de pacotes na rede. É importante ressaltar que, se tratando de adaptadores *SCI*, colocar pacotes na rede significa copiá-los para uma área de memória compartilhada.

4.8.5 Colocação das mensagens no meio físico

Conforme dito na seção anterior, cada nodo somente poderá colocar pacotes no meio físico no *slot* apropriado. Além disso, somente uma quantidade de informação limitada pelo tamanho do *slot* pode ser colocada.

A largura de banda de um *slot* é dada por:

$$B = V \times \Delta t \tag{4.2}$$

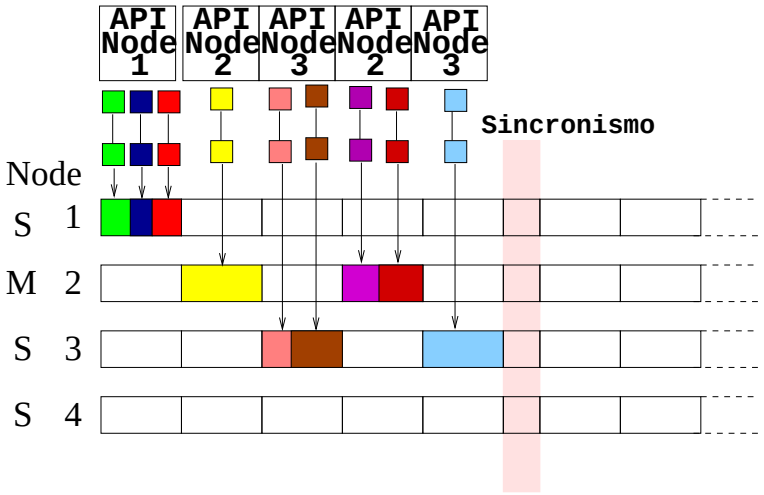

FIGURA 4.28 – Sistema de colocação dos pacotes nos canais no *slot TDMA*

Onde:

B: Largura de banda

V: Velocidade de transmissão do adaptador *SCI*, dada em $\frac{Bytes}{s}$

Δt: Diferença do tempo de início para o fim de um *slot*, ou seja o período do mesmo

Toda vez que um determinado nodo em um dos seus *slots* transmitir uma quantidade de dados maior que B, ele estará entrando no tempo de transmissão do próximo *slot*. Essa situação não pode ocorrer, a soma de todos os canais designados para um determinado *slot* deve ser menor que o seu tamanho. Essa garantia é fornecida pela ferramenta de configuração do sistema que será apresentada posteriormente. Na Tabela 4.2 essa restrição é respeitada, nenhum *slot* tem seu limite ultrapassado.

O algoritmo utilizado para o escalonamento dos canais dentro de um *slot* é uma variação do *Round-Robin* com pesos. Cada canal recebe, pela ordem de identificadores, uma fatia do tempo do *slot* para transmissão de pacotes. Diferentemente do *Round-Robin* tradicional, onde todas as fatias teriam mesmo tamanho, esse tamanho é proporcional à largura de banda que o canal tem direito de transmitir. Essa situação pode ser vista na figura 4.28. Os tons de cinza representam os diferentes canais. Canais com maior largura de banda ocupam proporcionalmente uma parcela maior do tempo do *slot*, pois precisam colocar na rede mais dados. A rotina de colocação das mensagens da rede verifica quais canais foram registrados para utilizar um determinado *slot* em uma tabela interna (como a tabela 4.2) e, para cada canal registrado, coloca no meio físico (que no *SCI* significa copiar para uma área de memória previamente exportada) a quantidade indicada de dados retirados da fila de entrada no canal. É importante ressaltar que nunca dois nodos colocam mensagens ao mesmo tempo, pois isso caracteriza um cenário de contenção.

Uma modificação que foi feita no *Round-Robin* com pesos é que cada canal tem direito a um tempo fixo do *slot*, independentemente dos outros canais terem pacotes para transmitir.

A integração da camada de enlace com a camada de acesso ao meio físico é feita através da rotina de escrita e de leitura do meio físico que é invocada pela

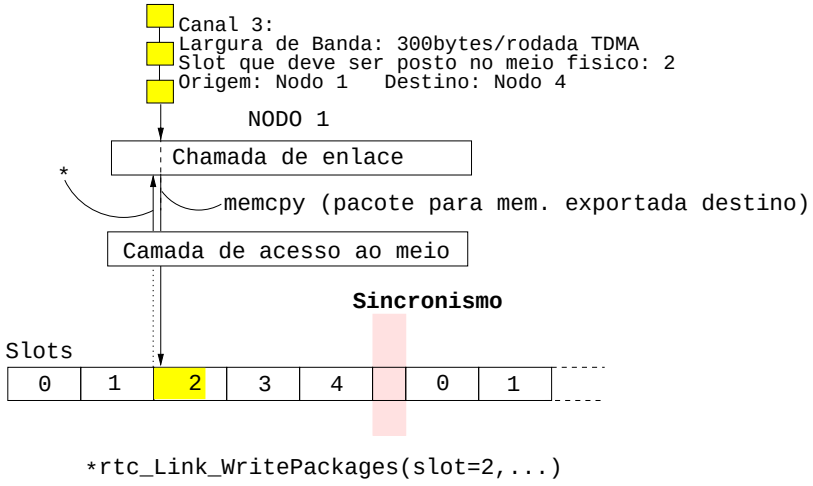

FIGURA 4.29 – Exemplo de colocação dos pacotes do canal 3 no meio físico

camada MAC. A cada início de *slot* a camada de acesso ao meio físico chama a rotina de escrita da camada de enlace passando como parâmetro um vetor de endereços das áreas de memória exportadas pelos outros nodos, o tamanho dessas áreas e o número do *slot* que iniciou. De posse dessas informações, a rotina de escrita no meio físico verifica se existem canais registrados que enviam pacotes nesse *slot*. Se não existirem, significa que esse *slot* será utilizado por outro nodo. Caso contrário, a função de escrita na rede retira pacotes das listas correspondentes aos canais e escreve-os no bloco de memória exportado pelo nodo destino. Isso é feito levando-se em conta a máxima transmissão possível para cada canal.

Na figura 4.29 pode-se ver um exemplo de colocação de pacotes na rede. Com o início do *slot* de número dois, a camada MAC chama a função `rtc_link_WritePackages` passando como parâmetro esse identificador do *slot*. Na tabela de registro dos canais consta que o canal 3 deve ser enviado pelo nodo 1 no *slot* 2. A máxima transmissão (ou seja, a largura de banda do canal) é de, no exemplo, 300 bytes por rodada do *TDMA*. Então, a camada de enlace coloca no meio físico (ou seja, copia para a área de memória exportada pelo nodo destino dos pacotes do canal 3) pacotes retirados da fila do canal 3. No *slot* 2 nenhum outro nodo pode ter registrado canais para serem enviados. Assim, nesse *slot*, o nodo 1 é o único com permissão de transmissão.

Como já foi dito, uma ferramenta de configuração é responsável pela correto escalonamento dos canais para os *slots* disponíveis.

O algoritmo 1 é responsável pela colocação dos pacotes no meio físico (implementado na função `rtc_link_WritePackages`). A cada início de *slot*, quando é invocado, o algoritmo determina quais os canais dos quais pacotes devem ser enviados. No caso de nenhum, o algoritmo termina. Já no caso de existirem canais cujos pacotes devem ser enviados no corrente momento, os pacotes desses canais são retirados das filas de entrada e copiados para a memória exportada do nodo destino. Para cada canal, além de uma fila de entrada, existe um repositório onde pacotes incompletos são armazenados. Antes de se enviar os pacotes de um canal, verifica-se se não existe um fragmento do último pacote da rodada anterior que não pode ser enviado. Esse pacote incompleto é enviado antes de todos os outros.

A necessidade de dividir-se pacotes vem do limite de transmissão de cada canal. Suponha-se que um determinado canal tem direito de colocar mais j bytes

Algorithm 1 Algoritmo de colocação dos pacotes da memória compartilhada

```
for all canal que deve enviar dados no slot atual do
  if lista de pacotes incompletos do canal está vazia then
    retirar um pacote da lista de entrada do canal
  else
    retirar o pacote incompleto do repositório de pacotes incompletos
  end if
  if tamanho do pacote <= total permitido para o canal then
    copia o pacote para área de memória exportada pelo destino
  else
    divide o pacote em dois pedaços de tamanhos apropriados
    copia o primeiro para área de memória exportada pelo destino
    guarda o segundo em um repositório de pacotes incompletos
  end if
end for
```

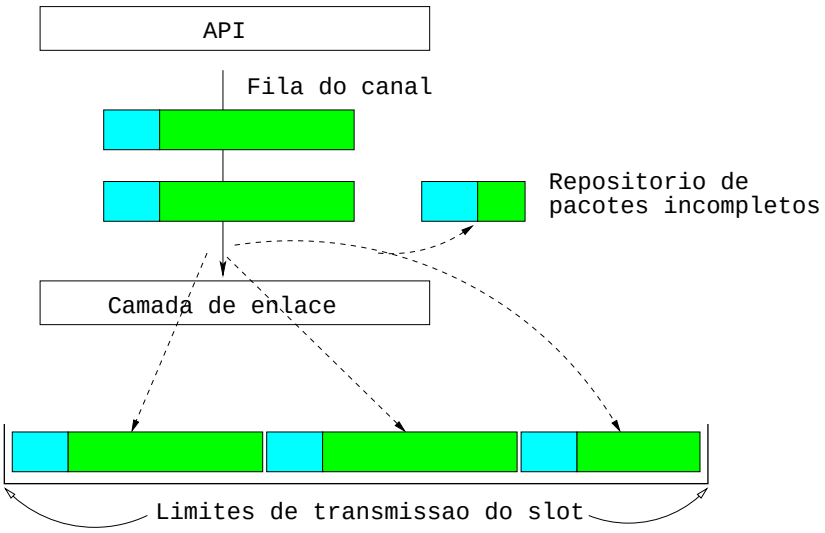

FIGURA 4.30 – Necessidade de divisão na colocação dos pacotes no meio físico

por rodada no meio físico. Se ele já tiver mandado k bytes (onde $k < j$) e o tamanho do próximo pacote na fila do canal é l (onde $l > (j-k)$), esse pacote deve ser dividido para se ajustar à possibilidade de transmissão de $(j-k)$ bytes. Assim, são formados dois pacotes, um com tamanho $(j-k)$ e outro com $l-(j-k)$. O primeiro é colocado no meio físico e o segundo vai para um repositório de pacotes incompletos. Isso é mostrado na figura 4.30. Na próxima chegada do *slot*, esse pacote incompleto é o primeiro a ser enviado e no nodo destino as duas partes são novamente montadas. É importante observar que se $(j-k)$ for menor que o tamanho do cabeçalho do pacote, a divisão não pode ser feita porque não é possível dividir cabeçalhos. No cálculo de reserva de banda, que será visto em seção futura, o overhead causado pelo cabeçalho é levado em conta.

Divisão de pacotes

Conforme visto, se um pacote tem tamanho maior do que a banda permitida de envio, ele será dividido. Para o gerenciamento de pacotes incompletos, o campo

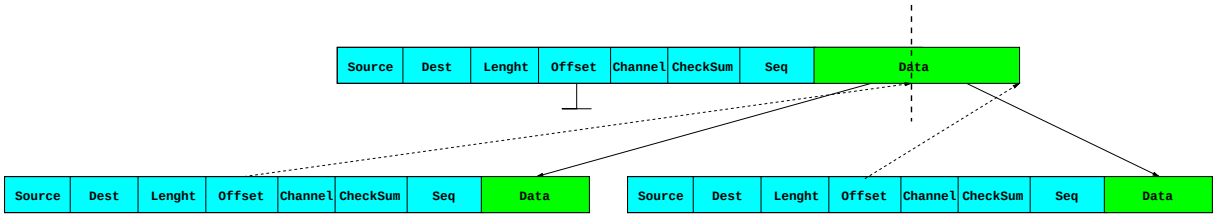

FIGURA 4.31 – Divisão de um pacote em dois fragmentos

`offset` é utilizado.

A figura 4.31 exemplifica como a divisão é realizada. Dois pacotes novos são gerados para substituir o original. Cada um deles contém uma parte dos dados do pacote original. Os dois novos pacotes contêm uma cópia do cabeçalho do pacote original, com exceção do campo `offset` que tem conteúdo diferente. O campo `offset` é um ponteiro para a posição final que os dados do novo pacote representam no pacote original. Assim, quando os diversos pacotes chegarem no destino, será possível encaixá-los novamente baseado na sua informação de `offset`. Pacotes inteiros (que não foram divididos) carregam um código especial em `offset` para indicar essa condição.

A divisão de pacotes permite um aproveitamento maior da banda fornecida pelos adaptadores *SCI*. Isso porque, se somente fosse possível o envio de pacotes completos, o final do período de transmissão poderia ser desperdiçado. Suponha que exista a possibilidade de envio de 200 bytes por rodada em um canal. Se todos os pacotes esperando na fila do canal tiverem tamanho de 101 bytes, somente uma mensagem será enviada a cada rodada do *TDMA*. Assim, o aproveitamento da banda reservada para o canal será aproximadamente 50% (sem se levar em conta outros *overheads*, como o cabeçalho das mensagens). Com a divisão das mensagens, é possível um melhor aproveitamento da banda reservada para cada canal.

Enquadramento

Nessa seção será apresentada a técnica utilizada para a colocação do pacote no meio físico do *SCI*, ou seja, como a cópia do pacote é feita na memória do nodo destino. Uma simples cópia não pode ser feita visto que o receptor necessita conhecer os limites (início e fim) de cada pacote na memória para fazer o correto reconhecimento de cada pacote.

O enquadramento é necessário porque um determinado nodo pode mandar mais de um pacote para um mesmo destino durante um *slot*. Assim, os pacotes são colocados seqüencialmente na memória oferecida pelo nodo destino e é necessário separá-los para o correto reconhecimento. É importante ressaltar que um nodo começa sempre a colocar seus pacotes a partir do primeiro endereço de memória oferecido pelo destino; assim pacotes que já estavam lá, sejam colocados por outros nodos, sejam pelo próprio na rodada anterior do *TDMA* serão sobrescritos. Um nodo não tem como saber se uma área de memória contém pacotes enviados por outros nodos, pois isso necessitaria de leitura remota que é uma operação com grande custo temporal.

O método utilizado para o enquadramento dos pacotes no meio físico (memória compartilhada) é a utilização de caracter inicial e final de quadro. O procedimento

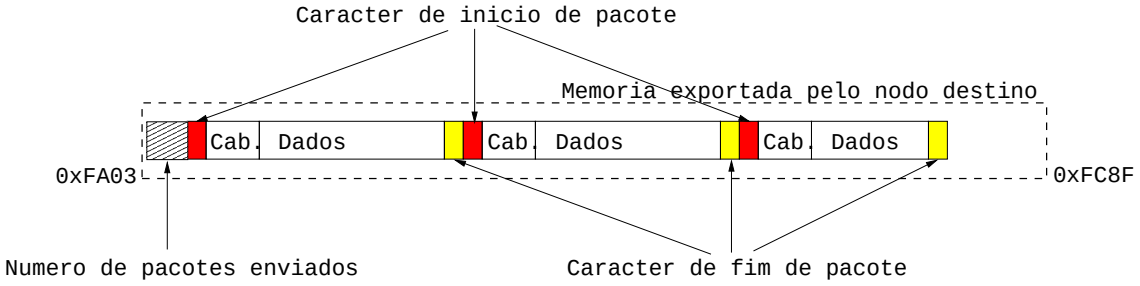

FIGURA 4.32 – Caracteres de início e fim do pacote que auxiliam o reconhecimento na recepção

usual nesses casos seria utilizar inserção de caracteres no pacote (*caracter stuffing*) para evitar a existência de seqüências de início ou fim de pacote no meio dos dados. O problema que essa técnica traz é que o tamanho do pacote no meio físico fica dependente do conteúdo do campo de dados. Pacotes onde os indicadores de início e fim não aparecem no campo de dados não terão nenhum caracter acrescentado; já pacotes onde existe uma grande ocorrência desses identificadores, muitos caracteres serão acrescentados, causando um problema de indeterminismo do tamanho do pacote. Outro problema da utilização pura e simples de caracteres especiais para marcar o início e o fim de quadro é a necessidade da procura de seqüências especiais de caracteres dentro do bloco de memória no nodo receptor.

Pelos problemas apresentados acima, resolveu-se utilizar uma mistura de contagem de caracteres com a utilização conjunta do campo de tamanho para evitar procuras desnecessárias de strings dentro do pacote. Assim, existe um marcador especial de início e fim de pacote. Mas não é feita inserção de outros caracteres. Na recepção, o caracter de início é verificado, após o tamanho é lido, o caracter de fim também é verificado, finalmente o *checksum*. Se algo der errado, esse pacote é ignorado e procura-se o caracter de início do próximo. Para utilizar-se essa estratégia, considerou-se que a taxa de erro da rede é baixa.

A figura 4.32 mostra alguns pacotes colocados na memória exportada pelo nodo destino. No início da área exportada é colocado o número de pacotes enviados para auxiliar a rotina de recepção. Assim, na recepção tem-se uma estimativa do número de pacotes a serem recebidos. Em caso de erro na transmissão desse número, os caracteres de marca (início e fim) mais o tamanho dos pacotes podem ser utilizados para determinar o número de pacotes.

A memória exportada pelo nodo destino é endereçada pela camada de enlace através de um vetor de ponteiros disponibilizado pelo módulo de conexão da plataforma proposta. Independente do modelo de exportação de memória utilizado (que foram apresentados na seção 4.5), o vetor conterá os endereços corretos das memórias compartilhadas por cada nodo na rede.

4.8.6 Recebimento de pacotes

Além do envio de pacotes no momento apropriado, a camada de enlace também é responsável pelo recebimento dos pacotes enviados por outros nodos da rede. No ambiente *SCI*, receber um pacote significa copiá-lo do bloco de memória oferecido ao emissor e colocá-lo na fila apropriada de recepção. O recebimento de pacotes

TABELA 4.3 – Possível tabela de alocação de *slots* de recepção

Nodo 1	
Slot	Recepção
0	–
1	–
2	5
3	–
4	–

também é guiado por uma tabela contendo os tempos corretos de recepção. Essa tabela é gerada pela ferramenta de configuração da plataforma.

Conforme visto, quando um nodo está enviando pacotes de um determinado canal, ele copia os pacotes para as memórias exportadas dos nodos destinos. Se ele mandar mais de um pacote para um mesmo nodo destino, os pacotes são colocados seqüêcialmente na memória do nodo destino. Um outro nodo que deseje mandar pacotes para o mesmo destino iniciará novamente esse processo, sobrescrevendo os pacotes previamente existentes. Portanto, um pacote deve ser retirado da memória compartilhada somente após um emissor ter colocado os pacotes e antes que um outro emissor sobrescreva essa área.

A questão do momento da retirada das mensagens da rede se torna relevante para o modelo de comunicação disponibilizado pelo *SCI*. Normalmente, em uma rede de difusão, a retirada da rede é feita pelo nodo destino no mesmo momento que o emissor está enviando o pacote. Isso acontece no nível físico na placa do *SCI*. Todavia essas operações são guiadas pelo hardware do *SCI* sendo não acessíveis no nível do software. Para o software, o *SCI* oferece o modelo de memória distribuída compartilhada, que é utilizado como meio físico básico na plataforma de comunicação. Assim, uma tabela de recepção em cada nodo indicando o momento correto da recepção é necessária. Os detalhes sobre a geração tanto da tabela de atribuição de canais a serem enviados nos *slots* quanto a de recepção serão vistos na seção 4.11. Após gerada a tabela, ela deve ser registrada na camada de enlace para ser utilizada. O registro dessa tabela na camada de enlace é feito através das seguintes primitivas:

```
void rtc_link_RegistReception (int Slot, int from_node);
void rtc_link_UnRegistReception (int Slot, int from_node);
```

A primitiva `rtc_link_RegistReception` é responsável pelo registro da necessidade de receber pacotes do nodo `from_node` no *slot* dado pelo parâmetro `Slot`. Já a retirada de um registro de uma recepção é feita utilizando-se a primitiva `rtc_link_UnRegistReception`.

Uma possível tabela de recepção pode ser vista na tabela 4.3. Nessa tabela, que está presente no nodo 1, somente no *slot* de número 2 serão retirados pacotes do bloco de memória oferecido ao emissor das mensagens. Um possível motivo para esse *slot* seria uma máquina mandando pacotes para o nodo 1 durante o *slot* 1. Assim, o segundo *slot* seria um momento adequado para a retirada desses pacotes da memória exportada. Esse cenário pode ser visto na figura 4.33.

No cenário exposto na figura, existe um canal que tem como origem o nodo 5 e destino o nodo 1. Assim, a cada rodada do *TDMA*, no *slot* 1 o nodo 5 coloca

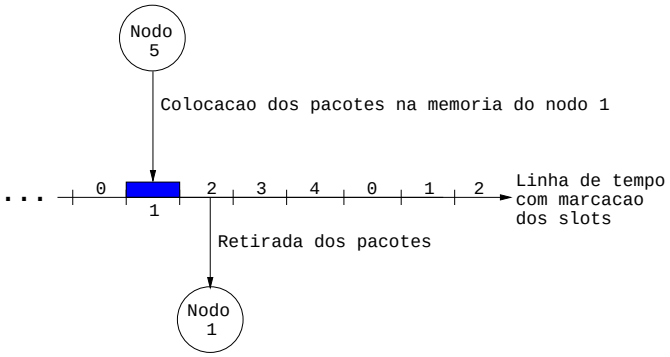

FIGURA 4.33 – Retirada de pacotes no *slot* 2

os pacotes relativos a esse canal na memória exportada pelo nodo 1. Já no *slot* 2 o nodo 1 lê esses pacotes, colocando-os na fila de recepção do canal. Assim, com a correta combinação da tabela de colocação com a tabela de retirada de pacotes dos blocos de memória compartilhados é possível um consistente estabelecimento de canais de comunicação com reserva de banda. Obviamente que no *slot* de número um o nodo 5 pode enviar mais de um canal, assim pode haver mais de um destino. Então, no *slot* de número dois, podem existir vários nodos retirando pacotes das suas áreas locais exportadas simultaneamente. Como a retirada de pacotes de áreas locais exportadas não causa nenhuma transmissão na rede de comunicação, ela pode ser realizada em paralelo em vários nodos.

É importante ressaltar que a necessidade de saber-se o nodo origem das mensagens que se deve retirar depende do padrão de exportação de memória adotado (veja seção 4.5.2). Se cada nodo exportou um bloco que é conectado por todos os outros, essa informação não é necessária visto que existe somente um bloco a ser lido para a retirada das mensagens. Caso o padrão tenha sido aquele onde cada nodo exporta um bloco de memória para cada nodo remoto no *cluster*, é necessário conhecer o emissor para saber de qual bloco as mensagens devem ser retiradas.

A rotina utilizado na retirada dos pacotes tem como função a criação de uma nova entidade pacote dentro do sistema (com a primitiva `rtc_alloc_package`), a cópia do cabeçalho e dos dados da área de memória local compartilhada para o pacote recém criado e sua inclusão na fila de saída do canal ao qual pertence o pacote (utilizando a primitiva `InsertToPackageList`). Antes dessa inclusão, uma verificação de erro deve ser realizada.

O algoritmo de retirada dos pacotes pode ser visto em *algoritmo 2*. Para cada um dos nodos dos quais deve-se receber pacotes no *slot* atual, varre-se todos os pacotes a serem recebidos e retira-se os mesmos um a um do bloco de memória exportado. Os pacotes são copiados para dentro da entidade pacote que é criada. Após o CRC ser testado, é verificado se o pacote recebido está completo. Caso afirmativo, ele é colocado na lista de saída do canal apropriado. Caso contrário, é verificada a lista de pacotes incompletos, pois o pacote pode ser o final de um pacote incompleto recebido na rodada anterior. Quando não existe ainda nenhuma parte desse pacote na lista, ele próprio é acrescentado, ficando à espera de outros que o completem. Se já existir um pacote contendo parte dos seus dados na lista, os dois pacotes são novamente unidos em um novo pacote maior. Se esse novo pacote estiver completo, ele pode ser colocado na lista de saída do respectivo canal. Caso

contrário, ou seja, ele ainda não está completo, ele vai novamente para a fila de pacotes incompletos à espera de outras partes que o completem.

Algorithm 2 Algoritmo de retirada dos pacotes da memória compartilhada

```
for all nodos dos quais deve-se receber pacotes no slot atual do
  for all pacotes que devem ser recebidos do nodo selecionado do
    reconhecer limites de um pacote
    criar uma entidade pacote que acomodará o pacote a ser recebido
    receber pacote (copiar cabeçalho + dados para o novo pacote criado)
    testar o CRC
    if pacotes está completo then
      colocar o pacote na lista de saída do canal ao qual ele pertence
      sinalizar a chegada de um pacote
    else
      if pacote recebido é parte de um pacote incompleto then
        retirar pacote incompleto da lista de pacotes incompletos
        juntar os dois pacotes
        if pacote resultante está completo then
          colocar pacote resultante na lista de saída do canal ao qual ele pertence
          sinalizar a chegada de um pacote
        else
          recolocar pacote na lista de pacotes incompletos
        end if
      else
        colocar pacote na lista de pacotes incompletos
      end if
    end if
  end for
end for
colocar uma marca no bloco de memória compartilhada indicando todos os pacotes
já recebidos
```

É importante ressaltar que mensagens muito grandes podem gerar pacotes que levam um grande número de rodadas do *TDMA* para serem enviados, portanto pode-se ter várias rodadas recebendo-se somente partes de um pacote maior.

Assim como o algoritmo de colocação de pacotes nos blocos de memória compartilhados, o algoritmo de retirada também é chamado no início de cada *slot* do *TDMA* e também recebe como parâmetro o número do *slot* corrente. Na figura 4.34 a retirada dos pacotes do exemplo da figura 4.29 é apresentada. No início do *slot* de número 3 a camada de acesso ao meio chama a função `rtc_ReadPackages` passando como parâmetro o número do *slot*. Essa função então verifica a tabela de recepção, e no nodo 4 essa tabela indica que pacotes provindos do nodo 1 devem ser recebidos. Assim, eles são retirados da memória oferecida pelo nodo e colocados em áreas de memória alocadas com `rtc_alloc_package`. Os pacotes completos são inseridos na lista de saída do canal 3 do nodo, para serem visíveis na API da plataforma. Os pacotes com dados incompletos são inseridos na lista de espera para serem posteriormente completados por outros pacotes das próximas rodadas.

União de pacotes Conforme visto, pacotes contendo apenas um fragmento

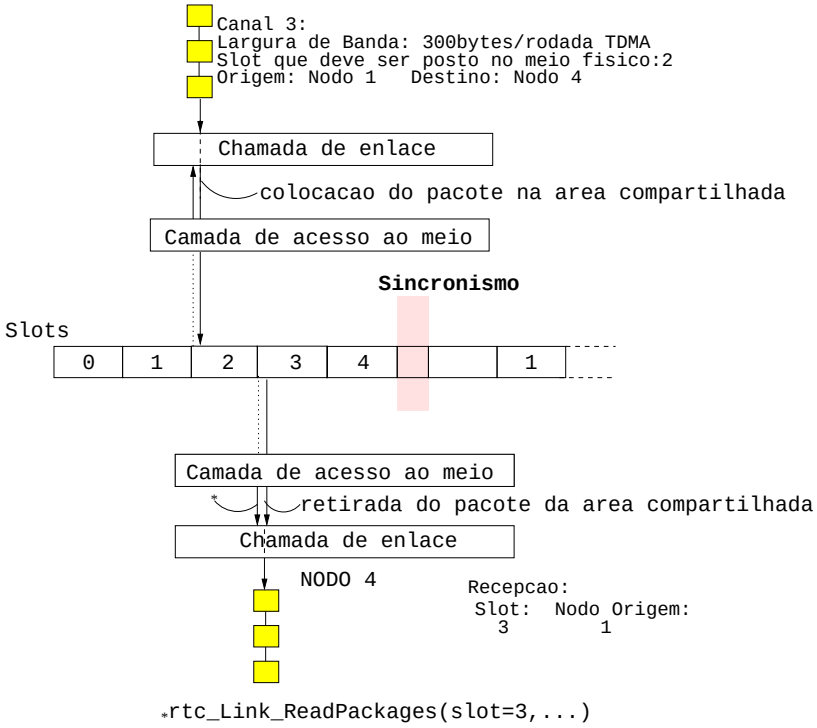

FIGURA 4.34 – Exemplo de retirada dos pacotes do canal 3 do meio físico

do campo de dados original (ou seja, pacotes incompletos) podem chegar no receptor. Esses pacotes não podem ser disponibilizados para a API do sistema pois um dos objetivos da plataforma é garantir que as mensagens transmitidas cheguem sempre de forma inteira no destino. Assim, pacotes recebidos que não contêm o campo de dados completo devem ser postos em fila de espera por outros pacotes que completem o campo de dados.

Para realizar a reunião de diversos pacotes que originalmente eram apenas um e foram divididos para melhor aproveitamento da banda, o campo de `offset` é utilizado. Como foi dito, esse campo indica onde o campo de dados do pacote se encaixa no campo de dados do pacote original. Quando todos os pedaços chegarem, o campo de dados da união de todas as partes irá ser igual ao de origem. Um novo pacote é gerado com o cabeçalho de um dos pacotes parciais (visto que fora o campo `offset` todos os outros são iguais), o campo de dados desse novo pacote é a concatenação dos campos parciais. Já o `offset` desse novo pacote é preenchido com um símbolo especial que sinaliza pacote completo e ele é enviado para a fila de saída do canal ao qual ele pertence. Os pacotes incompletos são então destruídos.

Conforme apresentado na seção da camada de enlace, uma das principais tarefas dessa camada é a colocação dos diversos pacotes no meio físico. Essa colocação é guiada pela temporização gerada pela camada MAC e por uma tabela que prescreve quais canais serão enviados em cada *slot*. Outra função da camada de enlace é a retirada de pacotes do meio físico. Essa retirada é guiada exatamente da mesma maneira que a colocação, através de temporização da camada MAC. É necessário um momento diferente para a retirada das mensagens devido ao paradigma do meio físico que é memória global compartilhada. Os algoritmos utilizados na transmissão e na recepção foram apresentados. Os segmentos de código C não são abordados devido à abundância de detalhes de implementação. Eles podem ser vistos no anexo

A.

4.9 Camada API (*Application Program Interface*)

Módulo de núcleo: `rtc_api.o`
Através dessa camada, aplicações de tempo real podem acessar os serviços de comunicação oferecidos pela plataforma. Essa camada provê um nível de abstração que separa a aplicação de detalhes internos da plataforma.

4.9.1 Objetivos

Os objetivos da interface do programa de aplicação são:

- Disponibilização de primitivas de comunicação para as aplicações de tempo real
- Empacotamento e desempacotamento das mensagens
- Registro das tabelas de comunicação na camada de enlace

4.9.2 Primitivas de comunicação

O principal objetivo do módulo API é o provimento de primitivas de comunicação que serão utilizadas em aplicações de tempo real. O conjunto de primitivas oferecidas caracterizam-se por tratar mensagens de tamanho variável. Uma mensagem pode ser enviada a um *canal* que foi estabelecido entre dois nodos. Uma analogia entre o canal e uma caixa de mensagens pode ser feita. Mensagens são colocadas no canal e ficam armazenadas em uma fila até que o receptor retire-as.

O paradigma utilizado é o da comunicação assíncrona: a operação de envio é não-bloqueante, isso é, o processo emissor é liberado para continuar o processamento assim que a mensagem for copiada para um *buffer* local da plataforma. A recepção é feita através de duas variantes: a primitiva bloqueante e a não bloqueante. No primeiro caso, o receptor continua a execução mesmo que não existam mensagens para serem recebidas. No segundo, o receptor fica bloqueado até existir alguma mensagem a ser recebida no canal de comunicação requerido. Pode-se comparar o envio não-bloqueante com a recepção bloqueante a semáforos que carregam informação: `send` é comparável à primitiva `signal` e `receive` à primitiva `wait` sendo então possível a sua utilização para sincronização.

As primitivas de envio providas pela camada API são:

```
void rtc_channel_send(int channel,void *msg, int msg_size)
void rtc_channel_send_high_priority(int channel,void *msg,
                                    int msg_size)
```

A primeira chamada é responsável pelo envio dos dados apontados por `msg` de tamanho `msg_size` pelo canal `channel`. `msg` é um ponteiro para a área de memória a ser enviada. Sendo um ponteiro genérico, pode-se enviar desde simples *strings* de caracteres até instâncias de tipos de dados abstratos completas. Obviamente no último caso, o problema da ordenação dos dados na memória das máquinas, caso

sejam heterogêneas, deve ser levado em conta, visto que essa primitiva faz um envio bruto dos dados da memória.

A segunda primitiva é muito semelhante à primeira. A diferença é que a mensagem é colocada no início da fila de envio, sendo assim tem uma prioridade maior que as outras que já estavam esperando para serem enviadas. Uma possível utilização dessa chamada é o envio de alarmes de controle de condições anormais na operação de um sistema distribuído.

Já as primitivas de recepção são:

```
int rtc_channel_receive(int channel,void *msg,int msg_size)
int rtc_channel_receive_if (int channel,void *msg,
                                          int msg_size)
```

que correspondem à recepção bloqueante e à não bloqueante respectivamente. O primeiro pacote que esteja na fila do canal `channel` será retirado e sua mensagem copiada para dentro da área de memória apontada por `msg` e pode ter um tamanho máximo de `msg_size`. A quantidade de bytes realmente recebidos é retornada pelas funções. A primitiva não bloqueante pode retornar zero se não existirem pacotes na fila de recepção.

As chamadas de envio e recepção foram inspiradas nas primitivas de comunicação de caixas de mensagem do *RTAI* que podem ser vistas na seção 3.3.

O processo completo de envio e recepção de uma mensagem utilizando as primitivas apresentadas pode ser visto na figura 4.35. Uma mensagem é enviada pela aplicação de tempo real do nodo 1 utilizando o canal 3 cujo destino é o nodo 4 (utilizando a primitiva `rtc_channel_send`). Essa mensagem é transformada em um pacote e este colocado na fila de entrada do canal 3 na camada de enlace. No momento que a camada de acesso ao meio sinalizar o início do *slot* onde pacotes do canal 3 são transmitidos, a camada de enlace os retira da fila e os coloca no bloco de memória exportado pelo nodo 4. Assim, a transmissão está efetuada. Já em um *slot* posterior está programada no nodo 4 a retirada de pacotes provindos do nodo 1. Quando a camada de acesso ao meio do nodo 4 sinalizar esse momento, os pacotes são retirados do bloco de memória compartilhado e colocados na fila de saída do canal 3. Com a chamada da aplicação da primitiva `rtc_channel_receive` um pacote é retirado e entregue para a aplicação.

Primitivas de envio

As primitivas de envio de mensagens realizam as seguintes funções:

- Verificação da validade da mensagem enviada
- Alocação de um novo pacote
- Preenchimento dos campos do pacote (empacotamento da mensagem)
- Colocação na fila de entrada da camada de enlace para o canal ao qual a mensagem pertence.

Algumas dessas operações podem ser vistas no segmento de código mostrado na figura 4.36. Na linha 8 um novo pacote é alocado utilizando primitivas vindas do módulo de gerenciamento de memória. Após isso, verificações de consistência são feitas para verificar se a mensagem requerida é válida e se o nodo foi alocado

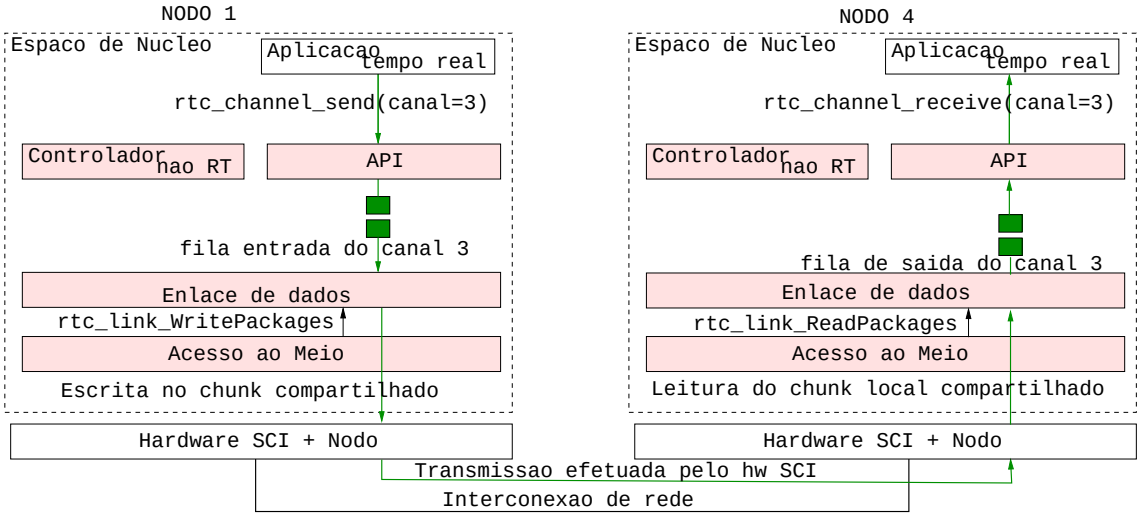

FIGURA 4.35 – Exemplo de envio e recepção de mensagem através do canal 3

corretamente. O próximo trecho é responsável pelo empacotamento da mensagem e será discutido abaixo. Finalmente a mensagem é colocada na lista de entrada da camada de enlace para ser enviada no momento apropriado. Note que esse código implementa tanto envio de mensagens com prioridade normal quanto o envio de mensagens de alta prioridade. A diferenciação é feita pelo parâmetro `priority`. Esse procedimento não é visível externamente ao módulo (por existir o modificador `static` no início do procedimento). As duas rotinas visíveis são `rtc_channel_send` e `rtc_channel_receive` que fazem a chamada do procedimento descrito passando o parâmetro de prioridade correto.

Empacotamento das mensagens O empacotamento é responsável por transformar uma mensagem em um pacote válido para a transmissão. Após alocado, o pacote deve ter seus campos preenchidos adequadamente de acordo com a mensagem a ser transmitida.

Na figura 4.36 entre as linhas 14 e 17 as informações de origem, destino, canal e `offset` são preenchidas. A origem e o destino vêm da tabela de canais gerada pela ferramenta de configuração que será analisada posteriormente. Já o canal vem do parâmetro `channel` e o `offset` é inicialmente 0, o que significa pacote completo.

Já o número de seqüência é obtido de um contador global para cada canal cujo acesso deve ser feito através de semáforos que garantam a atomicidade. O número de seqüência juntamente com o identificador do canal podem ser utilizados para identificar de forma unívoca um pacote dentro da plataforma.

Após a atribuição do número de seqüência, o tamanho da mensagem é copiado para o campo do pacote e a mensagem propriamente dita é enviada através de um `memcpy`.

Primitivas de recepção As primitivas de recepção realizam as seguintes funções:

- Verificação da validade dos parâmetros

```
1  static void rtc_priority_send (int channel,
2                                 void *msg,
3                                 int msg_size,
4                                 int priority)
5                                 /* 0 high   1 low */
6  {
7  TPackage *node;
8  node = rtc_alloc_package();
9
10 (...)
11 /* Checagens de consistência */
12
13 /* Criação do pacote a partir da mensagem */
14 node->header.source = ChannelConfiguration[channel].source;
15 node->header.dest = ChannelConfiguration[channel].dest;
16 node->header.channel = channel;
17 node->header.offset = 0;
18
19 /* Colocação do número de seqüência */
20 rt_sem_wait(&seq_semaphor);
21 node->header.seq = PackageSeqCounter[channel];
22 PackageSeqCounter[channel]++;
23 rt_sem_signal(&seq_semaphor);
24 (...)
25 node->header.datalen = msg_size;
26 /* Colocacao dos dados no pacote*/
27 memcpy (node->data,msg,msg_size);
28
29 /* Colocação na fila de entrada da camada de enlace */
30 /* priority = 0: inicio da fila   1: fim*/
31 InsertToPackageList (&(InputPackageList[channel]),
32                                 node, priority);
33 }
34 /****** Hight level API *****************/
35 void rtc_channel_send (int channel, void *msg,
36                                 int msg_size)
37 {
38   rtc_priority_send(channel,msg,msg_size,1);
39 }
40
41 void rtc_channel_send_hight_priority (int channel,
42                                       void *msg,
43                                       int msg_size)
44 {
45   rtc_priority_send(channel,msg,msg_size,0);
46 }
```

FIGURA 4.36 – Código parcial da rotina responsável pelas primitivas de envio

- No caso da recepção bloqueante, verificar se existem pacotes na fila, se não existirem, dormir em uma fila de espera
- Receber o pacote
- Desempacotamento da mensagem
- Colocação do conteúdo da mensagem no *buffer* da aplicação
- Liberação da memória ocupada pelo pacote

Algumas dessas operações podem ser vistas no segmento de código mostrado na figura 4.37. Esse segmento é parte da rotina de recepção bloqueante. Inicialmente são verfificados os argumentos da chamada (código não mostrado na figura). Para o controle do número de elementos nas filas de saída dos canais da camada de enlace, é utilizado um semáforo contador para cada canal. Toda vez que um pacote é retirado da fila, a primitiva `wait` é chamada, decrementando um do semáforo. A camada de enlace, toda vez que coloca um pacote em uma fila de saída de algum canal, chama a primitiva `signal` do semáforo do canal. Note que a camada de enlace nesse contexto é uma produtora de pacotes e a API consome esses pacotes. O semáforo é utilizado para que, caso não exista nenhum pacote na fila de um determinado canal, a recepção fique dormindo na fila desse semáforo até que a camada de enlace "produza" um pacote nesse canal.

Todos os semáforos controladores da quantidade de pacotes em cada canal são criados pela camada de enlace e inicializados com zero. Na linha 12, a primitiva `wait` testa a quantidade de pacotes no canal requerido. Se ela for positiva, o semáforo é decrementado de um e o pacote é retirado da fila. Se a quantidade for zero, a rotina bloqueia até um pacote ser recebido pelo canal.

Após isso, o pacote é retirado da fila do canal. Seu tamanho é verificado para ver se a aplicação disponibilizou um *buffer* de tamanho suficiente. Se não, o pacote é re-inserido na fila de saída da camada de enlace.

Finalmente a mensagem é desempacotada, copiada para o *buffer* da aplicação receptora e a memória liberada com a primitiva `rtc_free_package`.

A implementação da primitiva não bloqueante é extremamente semelhante. A principal diferença é que no lugar de `rt_sem_wait` a chamada `rt_sem_wait_if` é utilizada. Essa primitiva do *RTAI* decrementa o semáforo se ele for positivo senão simplesmente retorna um valor avisando que o semáforo é zero sem bloquear.

Desempacotamento da mensagem O desempacotamento é bem simples e pode ser visto nas linhas 26 e 27 da figura 4.37. Simplesmente o campo de dados é copiado para o *buffer* da aplicação e o tamanho da mensagem recebida é retornado.

4.9.3 Registro das tabelas de comunicação na camada de enlace

As tabelas de alocação dos *slots* de transmissão dos canais e de alocação dos *slots* para a recepção das mensagens de todos os nodos da rede são geradas pela ferramenta de comunicação do sistema em um arquivo de configuração. A partir desse arquivo, a camada API chama as primitivas `rtc_link_RegistConnection` e `rtc_link_RegistReception` para o registro dos *slots* de envio e recepção do nodo local.

A geração dessas tabelas será tratada em seção posterior.

```
1  int rtc_channel_receive (int channel, void *msg,
2                           int msg_size)
3  /* Recepcao bloqueante . Número de bytes
4  {                              recebidos retornado */
5  TPackage *node;
6  int datalen;
7
8  (...)
9
10 /* Checagem de consistência */
11 /* Existe alguma mensagem no canal? */
12 rt_sem_wait (&receive_sem[channel]);
13 node=GetFromPackageList(&(OutputPackageList[channel]));
14 (...)
15
16 /* Testa se o buffer fornecido pode comportar a msg;
17    se não, ela é reinserida na fila de saída */
18 if (node->header.datalen>msg_size)
19 {
20   InsertToPackageList (&(OutputPackageList[channel]),
21                                              node, 0);
22   rt_sem_signal(&(receive_sem[channel]));
23   rt_printk("Warning: Buffer is too small\n");
24 return 0;
25 }
26 memcpy (msg,node->data,node->header.datalen);
27 datalen = node->header.datalen;
28 rtc_free_package (node);
29 return datalen;
30 }
```

FIGURA 4.37 – Código parcial da rotina bloqueante de recepção

4.10 Módulo controlador não-RT

Módulo de núcleo: `ip_driver.o`

O módulo controlador não-RT é responsável pela capacidade da arquitetura de enviar tráfego sem restrição temporal. Com ela, é possível utilizar comunicação baseada no protocolo de rede *IP* através dos adaptadores *SCI*. Naturalmente, o tráfego *IP* tem prioridade baixa e não pode influenciar a transmissão dos canais de tempo real.

4.10.1 Objetivos

Os objetivos do módulo são:

- Transmissão de pacotes *IP* provindos do núcleo para a camada de enlace da plataforma
- Recepção de pacotes *IP* da camada de enlace e posterior repasse para o núcleo do sistema operacional

Para aproveitar-se a imensa gama de aplicações que utilizam *TCP/IP* ou mesmo *UDP* como base de comunicação, resolveu-se dar suporte à comunicação *IP* dentro da plataforma de comunicação. Assim, o módulo de simulação não-RT cria uma interface de rede que é utilizada para o envio e recepção de pacotes *IP* utilizando a plataforma de comunicação.

4.10.2 Funcionamento

O módulo controlador não-RT é responsável pelo registro de uma interface de rede que permite que pacotes *IP* sejam transportados pelo *SCI* através da plataforma de comunicação. Conforme visto na figura 3.3, uma aplicação de usuário utiliza os *sockets* BSD para abrir um canal de comunicação com outro processo. Se essa comunicação for entre diferentes nodos de máquinas em uma rede *IP*, a família de endereçamento *INET* é selecionada. Assim, a camada de *sockets INET* será utilizada. Essa comunicação pode utilizar tanto o protocolo de transporte *UDP* não confiável quanto *TCP*. Esses protocolos de transporte utilizam a camada de rede *IP* para envio de pacotes endereçados a outros nodos. Os pacotes *IP* são enviados através de uma interface de rede registrada no núcleo. Assim, se a plataforma de comunicação prover uma interface de rede para o núcleo e transportar os pacotes gerados por ele, toda a base de software existente baseada em *IP* poderá automaticamente utilizar o *SCI* como meio físico de comunicação.

Na figura 4.38 temos as camadas internas do núcleo do Linux interligadas com a plataforma de comunicação. Somente os módulos principais da plataforma são mostrados na figura. Após o processo escrever no *socket*, essa mensagem é transformada em pacote(s) TCP/IP que é enviado para a interface de rede. O módulo controlador registra uma interface de rede no núcleo, assim, os pacotes são enviados para a plataforma de comunicação. Eles então são trasmitidos para o destino e novamente entregues para o núcleo do Linux na máquina destino do pacote.

Para o transporte de tráfego não-RT, a camada de enlace disponibiliza um canal especial. Os pacotes são transportados utilizando esse canal para o destino.

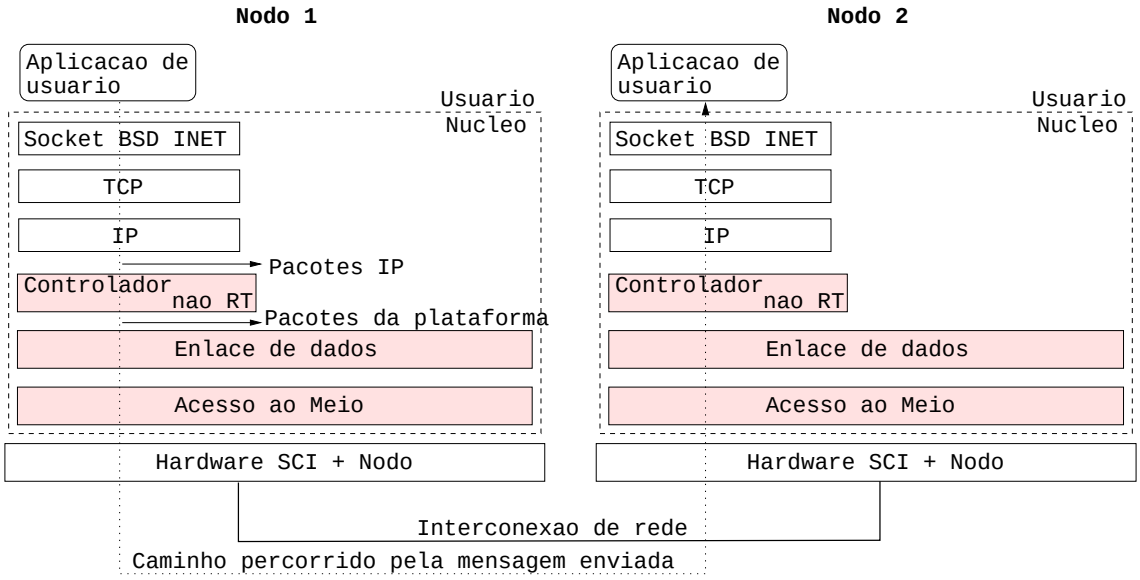

FIGURA 4.38 – Plataforma de comunicação interligada com as camadas de rede internas do núcleo do Linux.

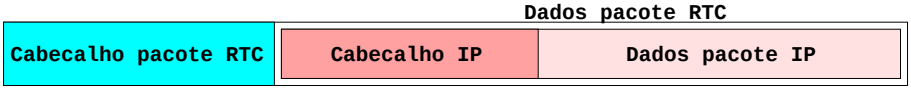

FIGURA 4.39 – Pacote *IP* sendo transportado dentro de um pacote RTC.

Pacotes *IP* não são reconhecidos diretamente pela plataforma, que utiliza um formato próprio de pacotes que foi mostrado na seção 4.8. A camada de enlace só é capaz de reconhecer e transmitir esse tipo de pacote. Assim, os pacotes *IP* devem ser transportados dentro do campo de dado dos pacotes RTC. A figura 4.39 ilustra esse fato.

Conforme visto na figura 4.38, os pacotes *IP* chegam do núcleo do Linux para o módulo controlador não-RT e são colocados dentro de pacotes RTC. Os pacotes RTC então são colocados na fila de um canal de entrada da camada de enlace para serem enviados para o destino. O canal utilizado para transporte de pacotes não-RT é tratado de forma diferente dos canais de tempo real. Seu identificador é 0 no sistema e ele é somente utilizado para comunicação sem requisitos temporais. Por isso ele é chamado de "canal não-RT".

O canal não-RT difere dos canais de tempo real por não existir um emissor e um receptor específicos. Assim, todos os nodos podem colocar pacotes e retirá-los desse canal. No lugar de se utilizar o número do canal para determinar o emissor e o receptor da mensagem, utiliza-se os endereços *IPs* para esse fim. Assim, o módulo controlador é responsável por analisar os pacotes *IP* e determinar o destino desses pacotes (rotear os pacotes). Para isso, o módulo utiliza-se de uma tabela de roteamento disponibilizada pela ferramenta de configuração do sistema.

Para a transmissão e recepção do canal não-RT, são utilizados dois *slots* dentro de cada rodada *TDMA*. No primeiro, um nodo tem direito de transmissão das mensagens não-RT. No segundo, todos os nodos da rede receberão os pacotes enviados no primeiro *slot*. Normalmente são utilizados os *slots* de números zero e um.

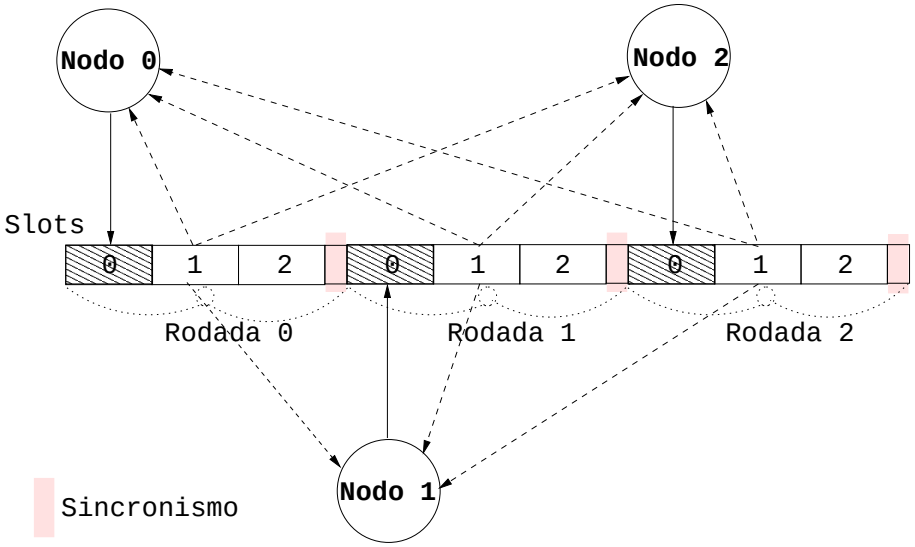

FIGURA 4.40 – *Slots* de transmissão e recepção do canal não-RT

Como o canal não-RT transporta pacotes provindo de aplicações que rodam em modo de usuário em todos os nodos do *cluster*, não existe nesse canal um nodo emissor e um receptor fixo. Para disciplinar o acesso à rede, definiu-se que somente um nodo poderá colocar seus pacotes provienientes do canal não-RT no meio físico a cada rodada do *TDMA*. As rodadas foram particionadas entre todos os nodos da rede.

Para realizar essa partição, inicialmente numerou-se as rodadas de forma crescente. Após isso, realizou-se uma partição do conjunto de números naturais. Seja um *cluster* contendo N máquinas. São necessárias, nesse caso, N classes de equivalência, uma para cada nodo. Assim, cada nodo somente poderá acessar o *slot* de escrita do canal não-RT em rodadas cujo número esteja dentro da sua classe de equivalência. Seja N o número total de nodos e n o identificador de um nodo em particular (identificadores variam de $n = 0..(N-1)$). A relação binária $R = (a, b) : a, b \in \aleph \; e \; a + b = n + (N \times k)$ onde $k \in \aleph$ define a classe de equivalência do nodo n. Assim, por exemplo para $N = 3$:

$n = 0$**:** Nodo pode transmitir pacotes não-RT nas rodadas de número: $0, 3, 6, 9, ...$

$n = 1$**:** Nodo pode transmitir pacotes não-RT nas rodadas de número: $1, 4, 7, 10, ...$

$n = 3$**:** Nodo pode transmitir pacotes não-RT nas rodadas de número: $2, 5, 8, 11, ...$

Com esse rodízio entre nodos, cada nodo terá direito de acesso ao *slot* de transmissão não-RT a cada N rodadas do *TDMA*. A figura 4.40 ilustra esse cenário.

Na figura, as rodadas são numeradas iniciando-se em zero. O *slot* de número zero é utilizado para transmitir o canal não-RT pela camada de enlace. Já o *slot* de número um é utilizado por todos os nodos para a retirada dos pacotes que acabaram de ser enviados. Como no canal não-RT todos os nodos podem transmitir e receber pacotes, é necessário um *slot* especial onde todos possam verificar se receberam algum pacote provindo do canal não-RT. Esse *slot* é normalmente o posterior ao envio.

```
1 static int driver_init (struct net_device *dev)
2 {
3   dev->mtu=BANDWIDTH_SLOT-(sizeof(THeader)+
4                     OVERHEAD + 2*sizeof(int));
5   dev->type=ARPHRD_ETHER;
6   dev->hard_header_len=0;
7   dev->flags = IFF_NOARP;
8   dev->open = driver_open;
9   dev->hard_start_xmit = driver_tx;
10  (...)
11 }
```

FIGURA 4.41 – Alguns campos da estrutura `net_device` sendo preenchidos na inicialização

Com a designação de dois *slots* dedicados ao transporte de pacotes RTC contendo pacotes *IP*, garante-se que o tráfego de tempo real não será influenciado por pacotes sem restrições temporais. Além disso, o tráfego *IP* sempre terá sua pequena fatia de tempo de transmissão. A metodologia utilizada foi inspirada nos mestre utilizados para tratar requisições aperiódicas dentro do escalonamento de tarefas periódicas ([BUT 2000]).

Registro do controlador de rede

Para ser possível a transmissão de tráfego *IP*, uma interface entre a plataforma de comunicação e o núcleo do sistema operacional é necessária. Essa interface foi feita através da implementação de uma interface de rede. Através dessa interface, o núcleo do sistema operacional envia pacotes *IP* que são tratados e transmitidos pela plataforma de comunicação. Além disso, os pacotes *IP* recebidos pela plataforma são repassados para o núcleo.

Conforme visto na seção 3.2.3 do capítulo 3, uma interface de rede precisa registrar-se no núcleo do Linux para ser invocada no momento que pacotes devem ser trocados com o mundo externo. Para isso, deve fornecer uma estrutura de dados que será colocada na lista global de interfaces de rede do núcleo. Esta estrutura é do tipo `net_device` e entre seus campos está o nome do dispositivo de rede registrado. No caso da plataforma, escolheu-se o nome "`scin0`" ("n" de não-RT). Assim, todos os pacotes direcionados para esse dispositivos serão repassados pelo núcleo para a arquitetura.

Além disso, a função `driver_init` é registrada e será chamada pelo núcleo para que o controlador se inicialize. Dentre as tarefas realizadas por essa função está o preenchimento de outros campos estrutura `net_device`. Entre os campos registrados estão ponteiros para as funções de abertura e transmissão do controlador e outras informações como a classe de dispositivo e o mtu (*Maximum Transfer Unit*) da rede. A classe de dispositivo de rede escolhida foi a *ethernet* (linha 4 da figura 4.41).

Uma operação importante realizada na inicialização do *driver* é a limitação do tamanho máximo de um pacote *IP*. Isso é feito definindo-se o mtu do dispositivo de rede que foi registrado. Ele não pode ser maior que o tramanho do *slot* pois os pacotes do canal não-RT não são divididos pela camada de enlace.

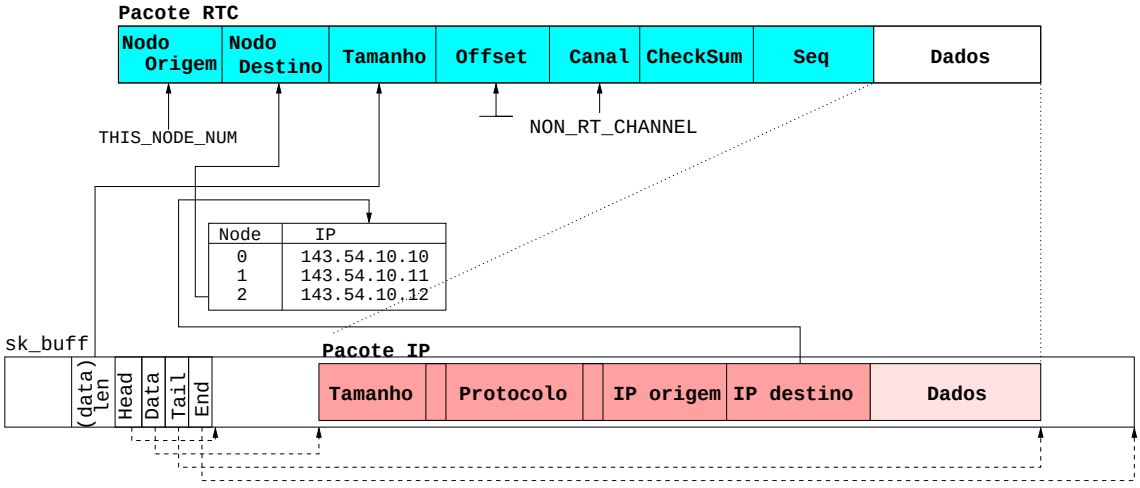

FIGURA 4.42 – Preenchimento dos campos do pacote RTC

Toda vez que a interface de rede `scin0` é inicializada (com o comando `ifconfig`) a função `driver_open` é chamada. É nessa função que os *slots* de transmissão e recepção do canal não-RT são registrados na camada de enlace utilizando-se as primitivas `rtc_link_RegistConnection` e `rtc_link_RegistReception`. Normalmente o *slot* 0 é utilizado na transmissão do canal não-RT e o *slot* 1 na recepção.

Transmissão de pacotes

No momento que o núcleo do sistema operacional deseja enviar um pacote *IP*, a função `driver_tx` do módulo controlador não-RT é chamada. Essa função é responsável por colocar um pacote *IP* dentro do campo de dados do pacote RTC e, após isso, colocar o pacote resultante na fila do canal não-RT.

O pacote entregue pelo sistema operacional para o módulo vem dentro de uma estrutura chamada `sk_buff`. Essa estrutura é utilizada internamente pelo núcleo para permitir que as diversas camadas internas de rede possam adicionar ou retirar cabeçalhos de forma facilitada. A rotina de transmissão do módulo (implementada pela função `driver_tx`) é responsável por tirar o pacote *IP* de dentro dessa estrutura e colocá-lo dentro do campo de dados do pacote RTC.

Para isso ser realizado, é necessária a alocação de um novo pacote com a rotina `rtc_alloc_package`. Após, os campos desse novo pacote são preenchidos. A figura 4.42 mostra o preenchimento do novo pacote. O campo de origem é retirado da constante identificadora do nodo. Já o destino é determinado pela tabela de roteamento preenchida durante a configuração do sistema.

No campo de tamanho tem-se que colocar o tamanho completo do pacote *IP*. Ele pode ser determinado pelo campo *len* da estrutura `sk_buff` pois ela contém na sua área de dados exatamente o pacote *IP* que deve ser enviado. Finalmente, o próprio pacote é copiado de dentro da estrutura `sk_buff` para dentro dos dados do pacote gerado.

Por fim, o pacote resultante é colocado na lista de entrada da camada de enlace para ser enviado no momento adequado.

Recepção de pacotes

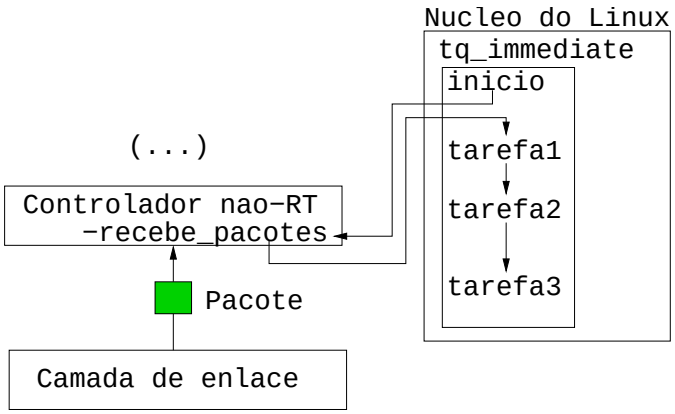

FIGURA 4.43 – Colocação da tarefa de recebimento de pacotes no núcleo.

Todo o pacote recebido que pertencer ao canal não-RT será repassado, através da fila de saída da camada de enlace, ao modulo controlador não-RT. O módulo então é responsável pela entrega do pacote *IP* que vem dentro do campo de dados do pacote RTC ao núcleo do sistema operacional.

No momento que a camada de enlace coloca um pacote na fila do canal não-RT, ela deve provocar a execução da rotina do controlador não-RT responsável por retirar esse pacote e processá-lo corretamente. Procedimento análogo é utilizado por outros controladores de rede, como o controlador de uma placa *ethernet*. No momento de chegada de um pacote, a placa ativa uma interrupção de hardware que provoca a execução da rotina tratadora responsável pela entrega do pacote ao núcleo do sistema operacional.

A camada de enlace poderia chamar diretamente a função de recepção de pacotes dentro do controlador não-RT. O problema é que não é aconselhado processos escalonados pelo *RTAI* (no caso, a camada de enlace) fazerem chamadas a rotinas dentro do núcleo normal do Linux. Isso porque essas chamadas fazem linhas de execução internas do núcleo serem ativadas, o que pode levar a comportamentos temporais não determinísticos, visto que não há como garantir o tempo de resposta da chamada, além de que ela pode ativar outras rotinas inclusive o escalonador do núcleo do Linux. Além disso, é desejável que rotinas do controlador não-RT rodem em baixa prioridade, ou seja só executem quando o núcleo do Linux estiver sendo executado.

Por isso, desenvolveram-se duas técnicas para a solução do problema. Na primeira, toda vez que um pacote é recebido pela camada de enlace, a mesma coloca o pacote na fila do canal não-RT e também instala a rotina de recepção do controlador na fila de tarefas `tq_immediate` do núcleo do sistema operacional. Assim, quando o núcleo do Linux for ativado no momento onde não existe nenhuma tarefa de tempo real executando no processador, a rotina de recepção do pacote dentro do controlador não-RT será executada.

A figura 4.43 ilustra essa abordagem. Toda vez que um pacote do canal não-RT é enviado para o controlador, a rotina de retirada e entrega do pacote é colocada na lista de execução do núcleo para ser executada no momento que o núcleo for escalonado pelo *RTAI*.

Para colocar uma função dentro de uma lista de tarefas do Linux, normalmente utiliza-se a chamada `queue_task`. Mas como não é aconselhavel utilizar chamadas

do núcleo do Linux em tarefas escalonadas pelo *RTAI*, decidiu-se alterar diretamente as estruturas de dados do núcleo para a inclusão da tarefa. A rotina original contém um *spin lock* para garantir a exclusão mútua. Se fosse chamada de dentro do *RTAI*, uma tarefa de tempo real poderia ser bloqueada por uma tarefa normal nesse *spin lock*.

Por existir um *spin lock* que garante a atomicidade na alteração das listas de tarefa do Linux, para alterá-las diretamente primeiro deve-se testar o *spin lock* com uma chamada não bloqueante para verificar se alguma parte do núcleo está acessando a lista de tarefa desejada (`tq_immediate`). Após esse teste, como a tarefa do *RTAI* tem prioridade sobre o núcleo, tem-se a garantia de atomicidade. Se for verificado pelo teste que existe alguém alterando a fila de tarefas, a camada de enlace não modifica nada e tentará novamente com a chegada do próximo pacote.

A segunda solução encotrada para o módulo controlador não-RT reconhecer a chegada de novos pacotes e entregá-los para o núcleo não requisitou modificações na camada de enlace. Nessa solução, o módulo testa periodicamente a fila de saída do canal não-RT para verificar a presença de pacotes colocados pela camada de enlace. Caso positivo, o pacote é tratado e entregue para o núcleo. A periodicidade é alcançada utilizando-se a lista `tq_timmer` do núcleo do sistema operacional.

A vantagem da primeira solução é que evita o acesso periódico à fila do canal não-RT, entretando ela depende de alterações diretas nas estruturas do núcleo, o que não é um método elegante. Por isso, a versão atual do sistema utiliza a segunda solução.

A rotina de recepção de pacotes do módulo controlador deve, após a retirada do pacote RTC da fila do canal, retirar o pacote *IP* do campo de dados e colocá-lo em uma estrutura `sk_buff` para após entregá-lo ao núcleo do Linux.

4.11 Ferramenta de configuração

Para configurar a plataforma de comunicação desenvolveu-se uma ferramenta própria. Com ela é possível a parametrização do sistema antes da sua execução.

4.11.1 Objetivos

- Permitir a configuração de parâmetros gerais da plataforma
- Permitir a configuração dos canais de comunicação e escaloná-los dentro dos *slots* disponíveis

4.11.2 Configuração de parâmetros gerais

Para a configuração de parâmetros gerais do sistema foi desenvolvida uma interface gráfica. Ela foi implementada utilizando-se a biblioteca gráfica `Gtk+` (*Gimp toolkit*). A biblioteca `Gtk+` é composta por um conjunto de rotinas utilizadas para a criação de interfaces de usuário. Embora tenha sido escrita em C, ela utiliza a idéia de classes e funções *callback* (ponteiros em C para funções) [GAL 99].

A ferramenta de configuração permite o usuário informar parâmetros relevantes para o sistema. Com base nessa configuração, a ferramenta gera um arquivo *header C* contendo todas as definições do sistema. Esse arquivo é utilizado na compilação da plataforma.

Os principais parâmetros configuráveis no sistema são:

Número de nodos: Número de máquinas presentes no cluster. Assim, uma versão customizada para o cluster pode ser compilada

Identificador do mestre: Indica qual máquina deve ser o mestre que envia o sinal de sincronização para o *TDMA*

Versão do núcleo: A plataforma pode ser executada em núcleo do Linux de versão 2.2.x ou 2.4.x.

Simulação Local: Se essa opção for habilitada, o sistema rodará em uma máquina só utilizando o módulo de simulação do *SCI*

Único/múltiplos *buffers* de recepção: Opção que indica qual o modelo de exportação de blocos locais de memória será utilizado

Número de *slots*: Define o número de *slots* que cada rodada *TDMA* terá

Largura de banda da rede: Nesse campo o usuário informa qual é a largura da rede utilizada

Período do ciclo: Campo utilizado para o fornecimento do perído de cada rodada *TDMA*

Nível de redundância: Utilizado para definir quantas cópias de uma mensagem serão mandadas para implementação de redundância temporal. Característica ainda não operacional no sistema

A partir dos dados coletados, um arquivo de configuração é gerado. A interface do sistema pode ser vista na figura 4.44.

4.11.3 Configuração do canais de comunicação

A interface de configuração desenvolvida permite também a configuração dos canais de comunicação que serão utilizados pela aplicação. Cada canal é caracterizado por:

Identificador: Identificador do canal

Origem: Nodo origem das mensagens do canal

Destino: Identificador do nodo destinatário das mensagens do canal

Largura de banda: Largura de banda por rodada que deve ser garantida pelo canal

Tamanhos máximo e mínimo de mensagens: Quais são os limites de tamanho de mensagens possíveis para o canal

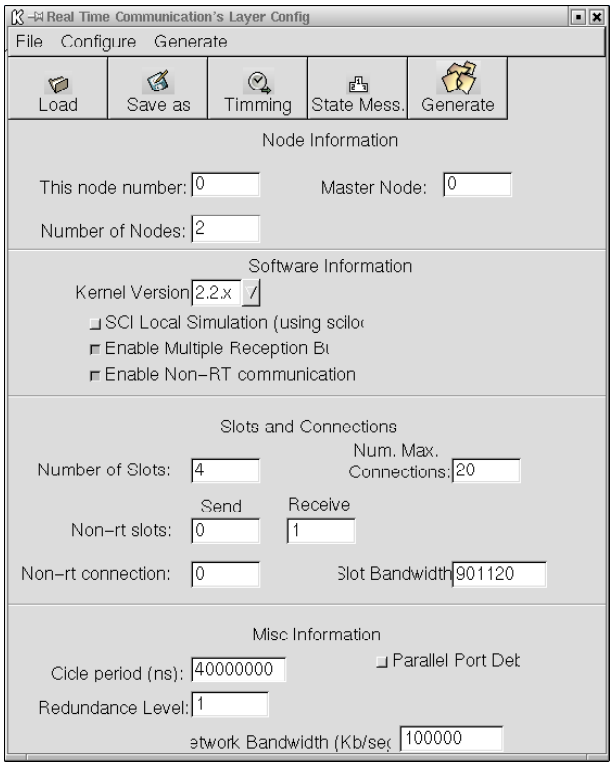

FIGURA 4.44 – Interface de configuração do sistema.

Após todos os canais serem indicados ao sistema, um algoritmo de escalonamento dos canais dentro dos *slots* disponíveis é executado.

Escalonamento dos canais nos *slots* do sistema

O escalonamento dos canais é responsável por indicar quais canais serão transmitidos em cada um dos *slots* do *TDMA*. Somente é necessário o escalonamento de um ciclo *TDMA* pois os demais serão iguais.

Antes de iniciar o escalonamento, acrescenta-se à largura de banda de cada canal o *overhead* causado pelo cabeçalho das mensagens nessa rodada. Para determinar esse *overhead* utiliza-se o tamanho mínimo possível para uma mensagem no canal. Com esse dado é possível saber qual é o número máximo de mensagens necessárias para enviar a largura de banda reservada para o canal. Como cada mensagem tem um *overhead* fixo, fica simples determinar o *overhead* do canal para uma rodada.

Para o escalonamento utilizou-se basicamente o agoritmo guloso com ordenamento. Inicialmente todos os canais são ordenados de maneira que os canais com mesmo nodo de origem fiquem junto na lista. Após, cada canal é retirado da lista e um *slot* é associado a esse canal. O algoritmo é guloso no sentido que atribui os *slots* em ordem crescente aos canais, ou seja, vai retirando canais na lista e preenchendo na ordem os *slots* disponíveis. Quando, por alguma restrição (que serão apresentadas a seguir), ele não conseguir atribuir o próximo *slot* livre ao canal que está sendo escalonado, ele pula passando a analisar o próximo *slot*. Se esse canal não puder ser condicionado em nenhum *slot* significa que não existe escalonamento possível utilizando a heurística desenvolvida.

TABELA 4.4 – Requisito de canais para uma determinada configuração do sistema

Nodo 0			
Id Canal	Origem	Destino	Largura Band.
1	0	1	300
2	0	2	200
Nodo 1			
Id Canal	Origem	Destino	Largura Band.
3	1	2	440
Nodo 2			
Id Canal	Origem	Destino	Largura Band.
4	2	1	486

Algumas modificações no algoritmo básico foram necessárias para levar em conta as seguintes restrições:

1. Em um mesmo *slot* não pode haver mais de um nodo transmitindo

2. Para um nodo origem transmitir para um determinado destino esse deve ter tido a possibilidade de retirar os pacotes previamente enviados para ele em outro *slot*

3. Um canal nunca pode ocupar mais de um *slot*

A primeira restrição impôs a ordenação dos canais, assim, todos os canais de uma mesma origem são escolhidos em seqüência pelo algoritmo.

A segunda restrição impõe que o escalonamento de retirada das mensagens deve ser feito conjuntamente com a escolha do *slot* que agrupará um grupo de canais. Assim, na hora de escolher um *slot* para alocar um canal, se o destinatário do canal não pode receber mensagens naquele *slot*, o próximo é utilizado. O modelo de exportação de memória utilizado afeta o escalonamento exatamente por efeito dessa restrição, como será mostrado em um exemplo adiante.

Já a terceira restrição impõe um limite para a máxima largura de banda de um canal porque ele precisa ser acomodado em um *slot*.

A recepção de um canal é sempre feita no *slot* seguinte ao envio. Um nodo que está transmitindo em um *slot* pode retirar da memória local compartilhada pacotes recebidos no *slot* anterior. Como a transmissão de dados pela rede demora mais que a escrita desses dados na área de memória relativa a um bloco remoto e o *SCI* continua a transmissão após a escrita de forma transparente, sobra tempo no *slot* para leituras em áreas locais compartilhadas (recepção de pacotes enviados no *slot* anterior).

Para exemplificar o escalonamento, considere uma rede contendo 3 nodos ($N = 3$) e uma rodada *TDMA* com 3 *slots*. Considere ainda os canais que devem ser escalonados mostrados na tabela 4.4.

O escalonamento obtido pela ferramenta de configuração para esses canais foi o seguinte:

1. Utilizando o modelo de exportação onde somente um bloco de memória compartilhada é exportado por máquina e todas as outras escrevem nesse bloco,

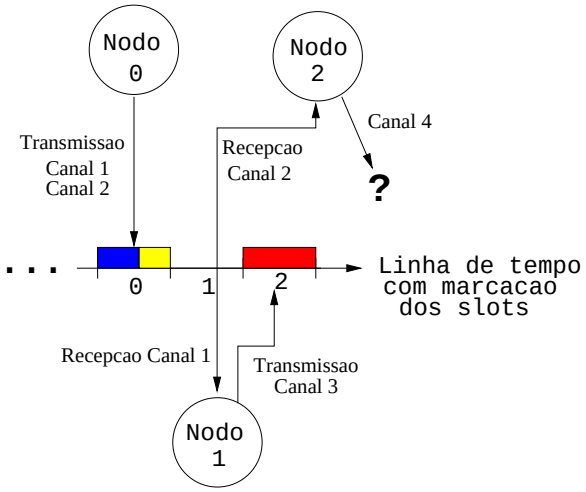

FIGURA 4.45 – Escalonamento de canais com nodos exportando bloco único de memória.

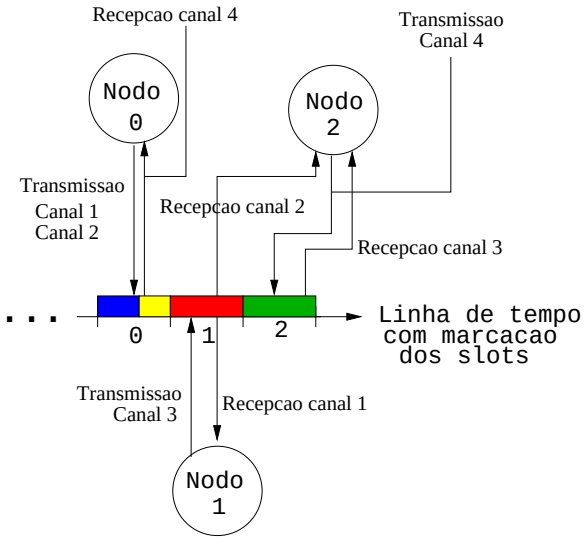

FIGURA 4.46 – Escalonamento de canais com nodos exportando um bloco de memória para cada nodo remoto.

não foi possível achar um escalonamento. A figura 4.45 ilustra o fato. O problema encontrado pelo escalonador é que o *slot* de número 1 não pode ser utilizado nem para a transmissão do canal 3 nem para a do 4. Isso porque as máquinas destino desses canais estão recebendo transmissões enviadas pelo nodo 1 nesse *slot* e como o bloco de recepção é único, ninguém pode escrever nesses blocos durante o *slot* 2.

2. Com o modelo de exportação onde cada máquina exporta um bloco de memória para cada nodo remoto, o mesmo conjunto de canais pode ser escalonado pelo algoritmo utilizado (figura 4.46).

Tanto a configuração dos possíveis canais do sistema como o escalonamento é feito previamente utilizando a ferramenta de configuração. A figura 4.47 mostra a interface do sistema onde os canais podem ser adicionados e após o escalonamento calculado.

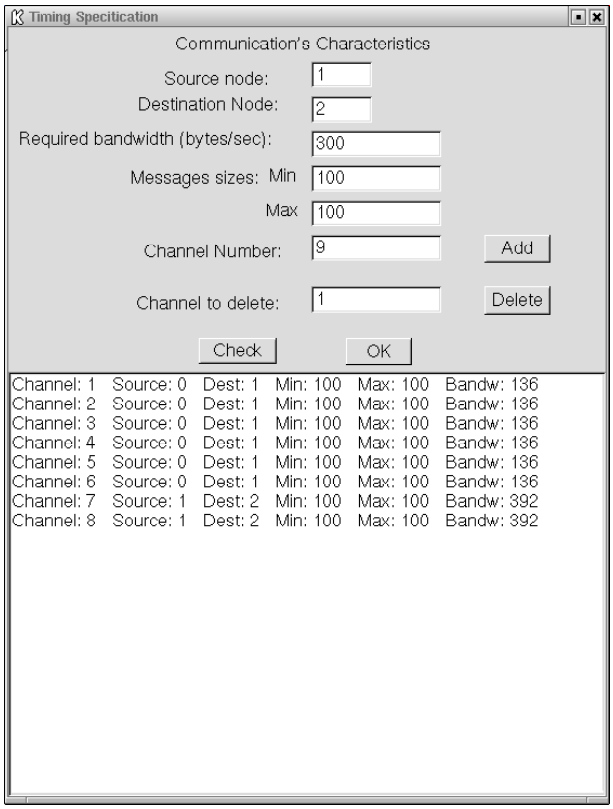

FIGURA 4.47 – Diálogo de configuração dos canais de comunicação.

Após o sistema ter seus parâmetros gerais configurados e o escalonamento dos canais ter sido efetuado, recompila-se a plataforma e ela está pronta para ser posta em execução.

5 Avaliação da plataforma

5.1 Introdução

O objetivo principal da plataforma de comunicação é a disponibilização de canais com garantia de banda para o transporte de dados que requerem garantias temporais. Para atingir esse objetivo, foi apresentada uma proposta de plataforma de comunicação que utiliza o *TDMA* para disciplinar o tráfego nos canais de comunicação.

Para avaliar-se essa plataforma, foram efetivados dois tipos de testes. O primeiro teve o objetivo de verificar o correto funcionamento da camada de acesso ao meio que é essencial para o provimento de determinismo de comunicação. O segundo avaliou o transporte de dados fim a fim utilizando toda a funcionalidade da plataforma, assim, todos os módulos e sua interação foram avaliados de forma global.

5.2 Ambiente de hardware

Para a realização dos diferentes testes, um cluster *SCI* com três nodos foi utilizado. Cada nodo foi composto por microcomputadores Pentium II 400MHz com 64Mb de memória. O sistema operacional utilizado foi Linux 2.2.18 com a extensão de tempo real *RTAI* 1.6. Os três nodos foram conectados através de adaptadores *PCI-SCI* baseados no controlador *PCI-SCI LCII* com *driver* próprio do adaptador.

5.3 Avaliação da camada de acesso ao meio

A camada de acesso ao meio, conforme visto, é responsável pela sincronização temporal dos nodos da rede e pela implementação do *TDMA* para disciplinar o acesso ao meio físico. A sincronização foi implementada através de uma barreira onde todos os nodos escravos esperam pelo sinal do mestre que marca o início de um novo ciclo.

O principal aspecto avaliado na camada de acesso ao meio foi a diferença de sincronismo entre o *TDMA* dos diferentes nodos. Essa diferença está relacionada com a precisão da sincronização dos relógios das máquinas e também com o *drift* existente entre os diversos temporizadores dentro de cada rodada.

Com a determinação do grau de sincronismo entre os nodos é possível avaliar qual o menor período possível para uma rodada *TDMA*, que é um limite importante. Isso porque o tempo de rodada determina qual é o menor garantia de tempo máximo de transmissão fim a fim que a plataforma pode prover.

Para fazer esse tipo de medição é necessária a utilização de uma mesma base de tempo para os três nodos. Por isso, um computador externo foi acrescentado exclusivamente para fazer amostragens temporais simultâneas na camada de acesso dos outros 3 nodos. Cada porta paralela dos computadores do cluster forneceu um bit de saída que foi ligado em bits de entrada da porta paralela do nodo responsável por amostrar dados. Esse cenário pode ser visto na figura 5.1.

Para fazer-se a marcação dos diversos *slots*, definiu-se que cada nodo inverteria

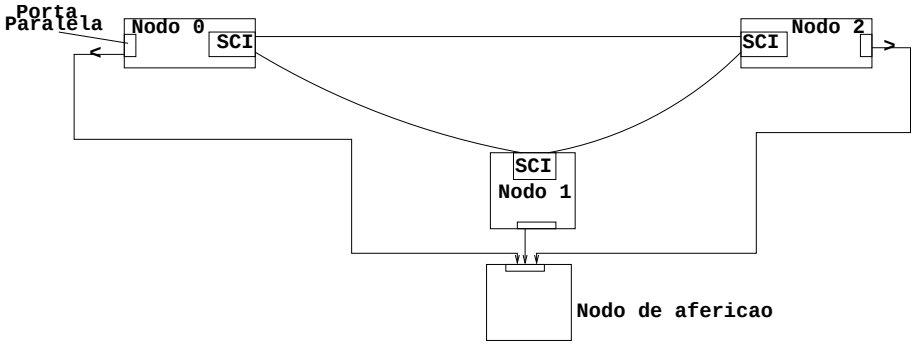

FIGURA 5.1 – Arquitetura utilizada para obtenção de medidas de desempenho

o seu bit de saída na porta paralela no início de cada *slot*. Assim, após a amostragem feita, é possível a confirmação do funcionamento da camada de acesso ao meio e ainda a verificação da diferença de relógio entre as camadas MAC dos diversos nodos.

5.3.1 Programa de aferição

Para realizar a amostragem dos bits enviados pelos nodos com informação sobre o início de cada *slot*, um programa de aferição para rodar no nodo especialmente colocado para esse fim foi desenvolvido.

O programa de aferição é composto por uma tarefa de tempo real que realiza a amostragem da porta paralela de forma contínua, para obtenção do maior número de amostras possíveis. Essa tarefa é a de maior prioridade no sistema, assim, ela roda continuamente sem perder o processador até que a quantidade de amostras requisitada seja adquirida. Nesse momento, o processador já pode ser liberado para as outras tarefas.

Um problema encontrado foi o acesso aos dados amostrados pela tarefa de tempo real, visto que ela armazena todas as amostras em uma área de memória alocada dentro do núcleo, não sendo visível na área de usuário. Para solucionar essa limitação, utilizou-se a chamada `rtai_kmalloc` que é responsável por alocar uma área de memória para ser utilizada por uma tarefa de tempo real que posteriormente, através de um identificador, pode ser também acessada por tarefas rodando no espaço de usuário.

Assim, um programa de usuário pode acessar as amostras lidas através de uma memória compartilhada com a tarefa de tempo real dentro do núcleo.

5.3.2 Metodologia

Para a realização das medições, utilizou-se um milhão de amostras que foram coletadas em $1.5s$ (1 amostra a cada $1,5\mu s$). As aferições foram feitas no *TDMA* rodando com ciclos de $40ms$, $10ms$, $1ms$. Foram utilizados 4 *slots* para cada ciclo *TDMA*. Os testes foram rodados 10 vezes para a confirmação dos resultados obtidos.

5.3.3 Resultados

Comportamento geral do *TDMA*

Inicialmente, observou-se o comportamento geral do *TDMA* para verificar-se o

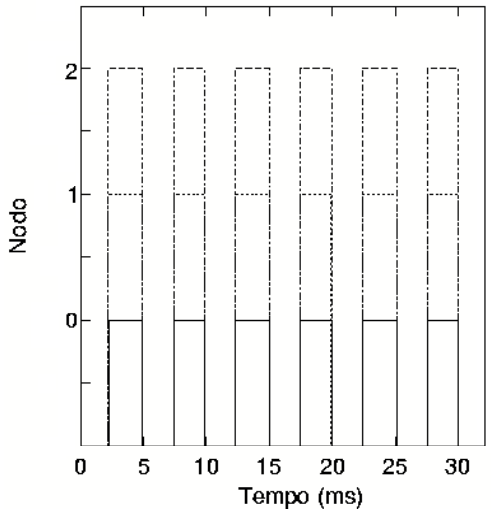

FIGURA 5.2 – Evolução temporal das rodadas *TDMA* dos 3 nodos utilizados no teste

seu correto funcionamento. A figura 5.2 mostra um gráfico onde a evolução temporal do protocolo *TDMA* dos 3 nodos é mostrada.

A cada início de *slot* a camada de acesso ao meio foi modificada para inverter o bit de saída da porta paralela. O gráfico mostra a evolução temporal de 3 bits. Cada mudança de nível em cada um dos 3 sinais apresentados no gráfico indica o momento de início de um *slot*. A figura mostrada foi obtida utilizando o período da rodada de $10ms$. Com outros períodos, obtém-se gráficos semelhantes. Para esse gráfico utilizou-se apenas parte das amostras coletadas.

O funcionamento geral do protocolo *TDMA* pode ser comprovado por meio do resulado apresentado.

Determinação do sincronismo entre o *TDMA* dos diferentes nodos

A determinação da diferença máxima de tempo entre os inícios dos *slots* em máquinas distintas foi feita com os mesmos dados coletados no ítem anterior. A cada início de *slot* a camada de enlace de cada nodo invertia um bit de saída da porta paralela que era coletado pela máquina destinada a esse fim. Com a análise desses dados, foi possível obter-se a diferença máxima e mínima entre os tempos das máquinas. Assim, pode-se determinar o *jitter* máximo entre o início dos *slots*.

A diferença temporal do início de cada *slot* entre dois nodos deu origem ao gráfico mostrado na figura 5.3.

Esse gráfico foi gerado com rodadas de $10ms$. Para rodadas de $1ms$ e $40ms$ e para outros pares de nodos, o resultado é similar. A máxima diferença temporal entre os nodos nesse teste é de $108\mu s$.

No resultado encontrado pode-se ver o reflexo de um aspecto de implementação. Na implementação da barreira de sincronização, utilizou-se uma espera de $100\mu s$ para a verificação do recebimento do sinal de início de rodada para evitar-se que as máquinas bloqueassem no caso da não operacionabilidade do mestre. O segmento de código relevante é mostrado a seguir:

```
1 /* Verifica recepção do sinal */
2 while (*((char *)LocalBaseBarrier)!=S)
```

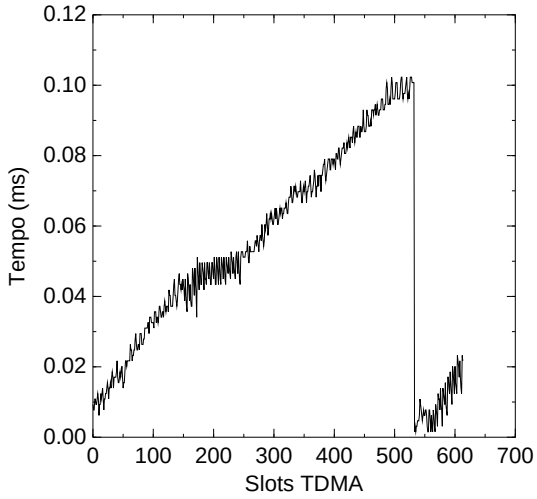

FIGURA 5.3 – Diferença temporal entre o início dos *slots* entre o nodo 0 e o nodo 2.

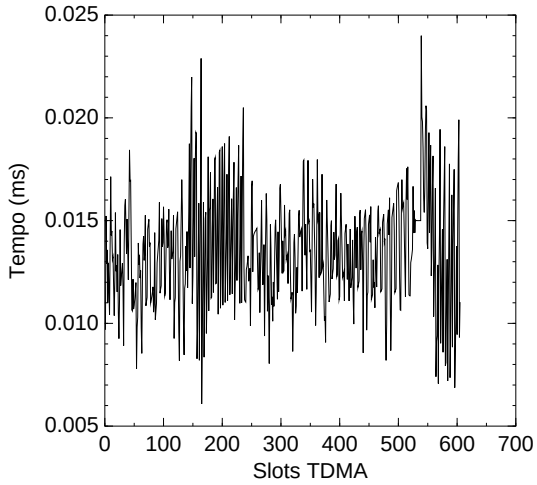

FIGURA 5.4 – Diferença temporal entre o início dos *slots* entre o nodo 0 e o nodo 2 após a modificação da camada MAC.

```
{
/* Dorme para outras tarefas poderem executar */
   rt_sleep(nano2count(100000)); /* (0,1 ms) */
}
```

Como pode-se observar, essa espera se reflete diretamente no gráfico apresentado em 5.3. A dente de serra gerada reflete o fato de que em cada rodada o sinal chegava um pouco além no tempo de dormência que na rodada anterior. Pode-se ver também um ruído de alta freqüência em torno desse sinal dente de serra. Esse ruído provavelmente é a verdadeira diferença de sincronismo, não a inserida pelo `rt_sleep`.

Assim, modificou-se o código tirando a espera não bloqueante de $100\mu s$ e no lugar utilizou-se uma espera bloqueante (*busy wait*) de $1\mu s$ que é responsável por liberar por algum instante o barramento de memória para o *SCI* ter tempo para acessar. O resultado encontrado está no gráfico da figura 5.4.

Conforme pode ser visto no gráfico, a maior diferença entre tempos de início de *slot* de dois nodos é $24\mu s$ e a menor é $6\mu s$. Os valores estão dentro do esperado visto que o valor típico de *latência* de entrega indicado no manual do *SCI* é de $5\mu s$. Esse valor é satisfatório para a execução de rodadas de $1ms$ de *TDMA*, pois aí a margem de segurança no envio de mensagens considerando essa diferença temporal pode ser de cerca de 3%.

No valor encontrado temos a influência da latência do *SCI*, o tempo de reconhecimento do sinal, o tempo de iniciar as tarefas geradoras de *slots* e o *drift* que acontece entre os relógios dentro de uma rodada.

Com os mesmos dados o *jitter* no sincronismo foi determinado. O valor é $24\mu s - 6\mu s = 18\mu s$.

5.4 Avaliação da transmissão de dados da plataforma

A comunicação de tempo real proporcionada pela plataforma baseia-se no suporte de canais de tempo real com reserva de largura de banda. Esses canais precisam garantir a banda reservada. A reserva de banda na transmissão de dados foi avaliada utilizando-se uma aplicação sintética que utiliza um canal para o envio de mensagens.

5.4.1 Aplicação sintética de avaliação

A aplicação desenvolvida é composta por dois módulos. O primeiro é responsável pelo envio de mensagens por um canal enquanto que o segundo recebe as mensagens enviadas. Cada mensagem contém o número da rodada atual, uma etiqueta com o tempo de colocação no meio físico (colocado na camada MAC) além de dados randômicos para preencher o tamanho determinado. Na recepção, o segundo módulo etiqueta cada mensagem com o tempo de recebimento e o número da rodada *TDMA* na qual o pacote foi recebido.

Para ser possível a colocação da rodada atual pelo emissor e pelo receptor, modificou-se a camada MAC a fim de ser possível obter-se essa informação que normalmente é privada do módulo de acesso ao meio. Outra modificação foi feita para que a camada de acesso ao meio coloque em um lugar determinado da mensagem uma marca de tempo com o momento que ela foi colocada no meio físico.

Na recepção, as mensagens são armazenadas em uma memória compartilhada para que um programa no nível de usuário possa coletar os valores e armazená-los em um arquivo para a posterior verificação.

As informações disponíveis sobre cada mensagem no final de seu percurso são:

- Identificador do ciclo *TDMA* na emissão
- Tempo no momento da colocação no meio físico (colocado na camada MAC)
- Identificador do ciclo *TDMA* na recepção
- Tempo no momento do recebimento da mensagem pela aplicação

O envio das mensagens é feito pela primitiva `rtc_channel_send` enquanto que a recepção utiliza a chamada `rtc_channel_receive`. Assim, a tarefa de recepção fica bloqueada até o recebimento de cada pacote. Após o recebimento, a mensagem

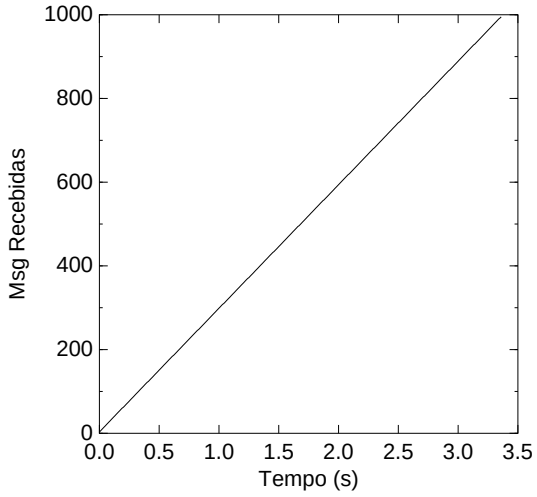

FIGURA 5.5 – Gráfico cumulativo de mensagens recebidas

é etiquetada com o tempo de recepção e armazenada na memória compartilhada com o nível de usuário para serem acessíveis pelo programa de leitura feito no nível de usuário.

Dois tipos de envio foram utilizados:

Envio sem espera: Todas as mensagens são enviadas em um laço contínuo no início da execução do módulo de envio

Envio com espera: Uma tarefa periódica com o mesmo período do *TDMA* é utilizada para fazer o envio das mensagens

5.4.2 Metodologia

Para a realização do teste, utilizou-se o *TDMA* com $10ms$ de período e 4 *slots*. Foi criado um canal de tempo real com $300\frac{Kb}{s}$ de reserva de banda por onde a aplicação sintética envia 1000 mensagens de $1Kb$. Dois cenários foram utilizados: no primeiro, existe somente esse canal no cluster. No segundo, uma carga alta de tráfego *IP* é colocada (carga acima do limite da rede), além da existência de outro cana RT que ocupa todo o último *slot* disponível.

5.4.3 Resultados

Garantia da banda

O gráfico da figura 5.5 mostra a evolução temporal cumulativa do recebimento dos dados na tarefa de recebimento para o envio com espera em uma rede com carga. Na abcissa temos o tempo de recebimento das mensagens enquanto que nas ordenadas o número de mensagens de $1Kb$ recebidas até o momento. Pode-se observar no gráfico que os 1000 pacotes enviados são recebidos em $3,33$ segundos, resultado esperado. A declividade da reta é $300mensagens/segundo$ que é exatamente a banda reservada. Além disso, não existem rajadas de envio a não ser se analisarmos no nível de temporização do *TDMA*.

Analisando-se a tabela de envio, verifica-se que cada rodada *TDMA* foi responsável exatamente pelo transporte de 3 mensagens de *1Kb*, ou seja, cumpriu a

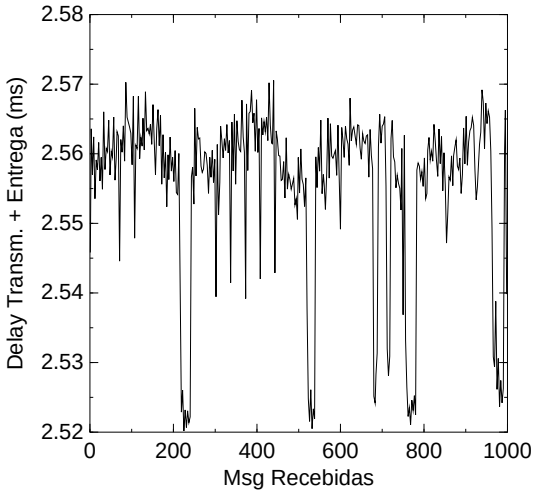

FIGURA 5.6 – Atraso de transmissão junto com atraso de entrega das mensagens

reserva de banda estipulada mesmo na presença maciça de outros tráfegos.

Mudando-se o cenário, ou seja, tirando-se a carga ou modificando-se o emissor para enviar todas as mensagens sem espera o resultado encontrado foi o mesmo, comprovando que o canal de tempo real não é influenciado por nenhuma outra transmissão realizada pela arquitetura de comunicação.

Atraso de transmissão e entrega

Conforme visto na revisão bibliográfica, o atraso fim a fim de uma mensagem é formado pelo atraso de geração, atraso de fila, atraso de transmissão e atraso de entrega. Devido ao fato do atraso de fila depender da quantidade de mensagens enviadas pelo emissor em relação à largura de banda do canal utilizado, decidiu-se medir o atraso de transmissão juntamente com o de entrega.

Esse medida possibilita um diagnóstico do funcionamento correto da plataforma.

Para essa medida, utilizou-se o tempo de colocação no meio físico juntamente com o tempo de recepção pela aplicação. Esses dois tempos são medidos utilizando a base de tempo sincronizada da camada MAC, assim é possível compará-los.

Pode-se ver o resultado desse teste na figura 5.6. Esse resultado foi obtido com 1000 mensagens de $1Kb$ e somente essa comunicação acontecendo. O receptor utilizado utilizou um laço contínuo de recepção para evitar o enfileiramento dos pacotes esperando pelo processo receptor.

A soma máxima dos atrasos de transmissão juntamente com o de entrega foi de $2.57ms$. O mínimo foi de $2.52ms$. Levando-se em conta que cada *slot* tem duração de $2.5ms$, a soma dos atrasos resultou aproximadamente no período do *slot*, resultado esperado visto que o canal era transmitido no *slot* de número 2 e recebido no de número 3. Obviamente mensagens maiores que ocupem mais de um *slot* para serem transmitidas terão atrasos maiores.

5.5 Avaliação

Os diversos testes apresentados nesse capítulo permitiram uma avaliação de aspectos funcionais da plataforma desenvolvida e comprovação de seu correto funcionamento. Como trabalho futuro, uma análise mais detalhada dos limites de

utilização da rede sem o comprometimento da largura de banda reservada pode ser feito.

Além disso, uma análise para verificar a interferência da plataforma de comunicação no comportamento temporal de outras tarefas de tempo real também pode ser realizada como trabalho futuro.

6 Conclusão

A disponibilidade de comunicação de tempo real em *clusters* de alta velocidade é um requisito para a execução de tarefas distribuídas de tempo real de alta performance, como por exemplo aplicações multimídia. Essa dissertação propôs uma plataforma de comunicação que permite o estabelecimento de canais de tempo real sobre clusters conectados com a tecnologia *SCI*.

O *SCI* é um padrão de interconexão baseado em memória compartilhada distribuída. Ele permite a disponibilização de blocos de memória local para máquinas remotas. Assim, do ponto de vista do software somente blocos de memória podem ser lidos e escritos para a troca de informação.

A inexistência de priorização do tráfego nos adaptadores *SCI* torna-o pouco adequado para o estabelecimento de comunicação de tempo real. Isso porque mensagens de baixa prioridade (não-RT) podem bloquear outras de alta (RT), o que caracteriza os fenômenos da contenção. Algumas propostas de modificação do *hardware SCI* foram feitas para solucionar esse problema, mas não existe nenhum fabricante dessas placas *SCI/RT*.

Apresentou-se no presente trabalho uma plataforma completa de comunicação baseada em camadas que permite a utilização do hardware comum *SCI* para a comunicação de tempo real baseada no paradigma de canal com reserva de banda. Esses canais são escalonados dentro dos *slots* gerados pelo *TDMA* previamente à execução por uma ferramenta de configuração. Após esse passo, é possível que uma aplicação envie mensagens por canais com largura de banda garantida. O tempo máximo de envio de um pacote pode ser calculado com base no tamanho da rodada *TDMA*.

O projeto da aquitetura foi feito de forma que a ordenação e o tamanho das mensagens são garantidos, ou seja, as mensagens enviadas sempre serão recebidas na mesma ordem e cada `send` do emissor corresponde a um `receive` no receptor.

Como as primitivas criadas pela plataforma para a emissão e o recebimento de mensagens têm semântica semelhante com a comunicação utilizando caixa postal do *RTAI*, uma aplicação pode utilizar a plataforma para comunicações remotas de maneira semelhante ao uso local das caixas postais.

Após o envio, as mensagens são transformadas em pacotes e enviadas para o destino através de escrita em blocos de memória compartilhados com máquinas remotas. Essa escrita deve ser disciplinada para evitar o fenômeno da contenção não determinística. Devido à natureza distribuída do meio de acesso, o escalonamento desse meio não pode ser facilmente feito utilizando-se um método que utiliza informações globais[MUK 99]. Em vez disso, decidiu-se utilizar um algoritmo guiado por tempo (*clock-driven*) para o escalonamento dos canais no meio de comunicação. Esse algoritmo é o *TDMA*. A desvantagem desse método é que mesmo quando um canal não está sendo utilizado sua banda está reservada e não é utilizada por outros canais, o que causa um desperdício.

Dentro de cada *slot* gerado pelo *TDMA* os diversos canais de comunicação são escalonados por uma heurística criada. A solução ótima da alocação estática desses canais nos *slots* de comunicação é difícil de ser obtida, pois problemas desse tipo são NP-difícil. Assim, a heurística utilizada nem sempre traz a melhor solução para o problema.

Além de permitir o estabelecimento de canais de tempo real, a plataforma apresentada permite que o tráfego normal *IP* seja transmitido através do *SCI*. Isso é feito através da implementação de um *driver* de dispositivo que fornece para o núcleo do Linux um novo dispositivo de rede. Todos os pacotes enviados para esse dispositivo serão tratados pela plataforma e transmitidos por um canal especial de comunicação não-RT.

A plataforma é composta por um conjunto de módulos Linux/RTAI relacionados de forma hierarquica inspirado na pilha de protocolos OSI.

A avaliação da plataforma mostrou que, mesmo com a existência de uma grande carga de canais de tempo real assim como comunicação não-RT, foi garantida a reserva de banda do canal observado. Isso torna a plataforma adequada para, por exemplo, o processamento de aplicações multimídia de tempo real. Além disso, a implementação do controlador de dispositivo para o uso do *SCI* no envio de pacotes *IP* tornou possível a utilização de bibliotecas de comunicação baseadas em *IP*, como o *MPI*, em conjunto com o tráfego de tempo real.

Como foi dito, o tempo de transmissão de uma mensagem na plataforma está relacionado com o tamanho do ciclo *TDMA*. Assim, ciclos menores permitem tempos menores. O problema é que o tamanho do ciclo é limitado pela granularidade da sincronização dos relógios que foi medido no capítulo de avaliação da plataforma.

A existência do *TDMA* implica que o relógio dentro da camada de acesso ao meio deve ser sincronizado. A sincronização dos relógios dos nodos é feita através de uma barreira onde os escravos esperam pelo sinal de sincronismo do mestre. Isso traz alguns efeitos, como por exemplo um certo *overhead*, pois esse tempo de sincronismo não pode ser utilizado para mandar nenhuma outra mensagem e também, dependendo da implementação, nenhuma tarefa pode processar. Esse tempo, juntamente com o tempo das outras tarefas de alta prioridade da plataforma, deve ser utilizado em uma análise de escalonamento para verificar se determinada aplicação cumprirá os seus requisitos temporais.

A utilização de uma mensagem de difusão facilitaria a implementação da sincronização e diminuição do atraso entre os diversos relógios sincronizados da rede. Mas o *SCI* não provê método de difusão, inviabilizando essa solução.

O *TDMA* implementado mostrou-se eficaz no impedimento da ocorrência de contenção no acesso ao meio. O problema encontrado foi o desperdício da banda quando certos nodos tinham pouco a transmitir. A utilização de canais alocados nos *slots TDMA* ajudou a minimizar o desperdício, pois se um nodo nada tem a transmitir nenhum *slot* é alocado para ele. A utilização do *Token Ring* tornaria o sistema mais dinâmico, embora a necessidade do uso de interrupções *SCI* na sua implementação.

A implementação da plataforma teve dificuldades adicionais devido à complexidade de se depurar código de tempo real rodando no espaço de núcleo. A grande extensão da plataforma implementada foi também responsável pela grande dificuldade de depuração.

Já o módulo de simulação do *SCI* mostrou-se uma ferramenta muito eficiente no processo de desenvolvimento por permitir toda essa fase ser desenvolvida localmente sem a necessidade do uso real do *cluster SCI*.

Várias melhorias podem ser adicionadas à plataforma no futuro. A criação e destruição de canais de tempo real feita durante o tempo de execução aumentaria a dinamicidade da arquitetura e faria o desperdício de tempo causado pelo *TDMA*

diminuir. Se o tempo de espera do mestre para enviar o sincronismo fosse calculado dinamicamente, no lugar de ser fixo num valor seguro para todas as faixas possíveis de período *TDMA*, diminuaria o overhead de uso do processador causado por essa espera.

A avaliação do limite do uso da banda do *SCI* pela plataforma sem comprometer o determinismo temporal é assunto de futura pesquisa.

Devido à existência de uma interface bem definida entre os diversos módulos é permitida a troca de um módulo por outro de forma facilitada. Isso permite que o sistema seja evoluído de forma a ser uma arquitetura configurável. Assim seria possível, por exemplo, a troca do módulo MAC baseado em *TDMA* por outro baseado em *Token bus* que seja mais apropriado para o tipo de tráfego atual da rede. Se essa troca for realizada durante a operação do sistema, um estado lógico e temporal consistente deve ser preservado. Para garantir isso, diversas melhorias devem ser acrescentadas à plataforma.

Outra característica que pode ser implementada é a tolerância à perda de mensagens devido a falhas nas comunicações. Uma solução para isso é a implementação de redundância temporal dos pacotes enviados através da plataforma.

Para uma aplicação de tempo real distribuída ser executada sobre o *cluster SCI*, trocando mensagens entre diferentes tarefas em diversos nodos, é necesssária a divisão do problema em processos de tempo real que são distribuídos pela rede. Uma análise da comunicação e das relações de precedência desse sistema é necessária. Uma ferramenta que automatize esse processo complementaria a plataforma e facilitaria o desenvolvimento de aplicações.

A determinação da banda necessária para determinada aplicação de tempo real precisa ser determinada antes da criação de de canais de tempo real que comportarão as mensagens enviadas por essa aplicação. Essa tarefa é chamada *SBA* (*Synchronousn Bandwidth Allocation*)[ZHE 00] e uma ferramenta que auxilie nessa determinação ajudaria no desenvolvimento de aplicações.

Além disso, uma camada de software que torne transparente a localização dos processos e facilite a nomeação dos canais de comunicação (como o *MPI*) pode ser desenvolvida no topo da plataforma de comunicação, ajudando o desenvolvimento de aplicações mais independentes de detalhes da infra-estrutura utilizada, como o número de nodos que formam o *cluster*.

Anexo 1 Segmentos de código

A.1 Colocação dos pacotes na área de memória compartilhada

```
1  static TPackage *LastWritePackage[NUM_CONNECTIONS];
2  void rtc_link_WriteMessages (void **buffer, int msglen,
3                               int slot, int cicle)
4  {
5
6    int OffsetChunk[NUM_NODES];
7    int counter[NUM_NODES];
8    int alreadySent[NUM_CONNECTIONS];
9    int connection,i;
10   int connection_counter;
11   int sendsize;
12   int data_offset;
13
14   TPackage *buf;
15
16 #ifdef DEBUG
17   printk("Begin␣WriteMessages␣%d\n",THIS_NODE_NUM);
18 #endif
19   if (slot==1) //That means, non-rt read time
20     return;
21
22   rt_sem_wait(&Map_sem);
23
24   for (i=0;i<NUM_NODES;i++)
25   {
26     OffsetChunk[i]=sizeof (int);;
27   }
28
29   for (i=0;i<NUM_CONNECTIONS;i++)
30   {
31      alreadySent[i] = 0; /* I have sent 0 bytes */
32   }
33
34   for (i=0;i<NUM_NODES;i++)
35   {
36      counter[i] = 0; /* ...and 0 packages */
37   }
38
39   connection_counter=0;
40
41   for (connection=0;\
42        connection<NUM_CONNECTIONS;connection++)
```

```
43  {
44    /*This connection is not working */
45    if (SlotMap[slot][connection]<=0)
46        continue;
47
48    if (connection==NON_RT_CONNECTION)
49      if (cicle%(THIS_NODE_NUM+1)!=0)
50      /* it is not time to send */
51        continue;
52
53    connection_counter++;
54
55    do
56    {
57      /* I have a partial package to send */
58      if (LastWritePackage[connection] != NULL)
59      {
60        buf=LastWritePackage[connection];
61      }
62      else
63      {
64        /* Take from Buffer */
65        buf = GetFromPackageList
66                (&(InputPackageList[connection]));
67        if (buf==NULL)
68        {
69          break;
70        }
71
72        buf->header.offset=0;
73                      /* partial len already sent */
74      }
75
76      sendsize = buf->header.datalen-buf->header.offset;
77      LastWritePackage[connection] = NULL;
78      if (sendsize + alreadySent[connection] +
79      sizeof (THeader) +
80      OVERHEAD>SlotMap[slot][connection])
81      /* There is no enought space to send this package,
82      /* should be divided */
83      {
84        sendsize = SlotMap[slot][connection] -
85                    (alreadySent[connection] +
86                     sizeof(THeader) + OVERHEAD);
87        LastWritePackage[connection] = buf;
88        if (connection==NON_RT_CONNECTION)
89        /* I can not send divided */
90        /* packages in non-rt connection */
```

```
91          break;
92        }
93
94        if (sendsize<=0||sendsize>( buf->header.datalen -
95            buf->header.offset))
96            /* There is no enought space even */
97            /* for the header and the */
98            /* overhead or some error */
99          break;
100
101       data_offset = buf->header.offset;
102       buf->header.offset += sendsize;
103       /* Size of this small package */
104       buf->header.part_len = sendsize;
105       rt_printk("Sendsize:␣%d\n",sendsize);
106
107        if (buf->header.dest==THIS_NODE_NUM)
108        {
109          rt_printk("This␣package␣is␣to␣myself!\n");
110           break;
111        }
112
113        if (buffer[buf->header.dest]!=NULL)
114        {
115
116         /* Begin of the packet */
117          *(int *)(buffer[buf->header.dest]+ \
118          OffsetChunk[buf->header.dest])=BEGIN_TAG;
119          OffsetChunk[buf->header.dest]+=sizeof(int);
120
121
122          /*Send the header of the package*/
123          memcpy (buffer[buf->header.dest]+ \
124          OffsetChunk[buf->header.dest],
125          &(buf->header),sizeof (THeader));
126
127          /*Send the data */
128          OffsetChunk[buf->header.dest]+=sizeof(THeader);
129          memcpy ((buffer[buf->header.dest]+
130          OffsetChunk[buf->header.dest]),
131          buf->data+data_offset,sendsize);
132          OffsetChunk[buf->header.dest]+=sendsize;
133
134          *(int *)(buffer[buf->header.dest]+ \
135             OffsetChunk[buf->header.dest])=END_TAG;
136             /* End of the packet */
137
138          OffsetChunk[buf->header.dest]+=sizeof(int);
```

```
139
140         alreadySent[connection]+=sizeof(THeader)+ \
141                                  sendsize + OVERHEAD;
142         counter[buf->header.dest]++;
143
144         if(buf->header.offset >= buf->header.datalen)
145         {
146            rtc_free_package(buf);
147            buf=NULL;
148            LastWritePackage[connection]=NULL;
149         }
150       } else
151       {
152         if (buf!=NULL)
153           rtc_free_package(buf);
154         buf=NULL;
155         LastWritePackage[connection] = NULL;
156         break;
157       }
158     } while (LastWritePackage[connection] == NULL &&
159      alreadySent[connection] < SlotMap[slot][connection]);
160
161   }
162
163   rt_sem_signal(&Map_sem);
164
165 #ifdef DEBUG
166   if (counter>0)
167     rt_printk("Node␣%d␣has␣sent␣%d␣packages.\n",
168                          THIS_NODE_NUM,counter);
169 #endif
170
171   if (connection_counter>0) /* Active connections */
172   {
173     for (i=0;i<NUM_NODES;i++)
174       if (i!=THIS_NODE_NUM)
175       {
176         /*Number of messages to receive */
177         *((int *)(buffer[i])) = counter[i];
178         *((int *)(buffer[i]+OffsetChunk[i]))=END_TRANS_TAG;
179         /* end of transmition */
180       }
181   }
182 }
```

A.2 Retirada dos pacotes da área de memória compartilhada

```
1  static TPackage *LastReadPackage[NUM_CONNECTIONS];
2  void rtc_link_ReadMessages (void **buffer, int msglen,
3                              int slot, int cicle)
4  {
5
6    TPackage *buf;
7    char *pointer;
8    int counter;
9    int i, npkg,sender;
10   THeader *PHeader;
11
12 #ifdef DEBUG
13   printk("Begin␣ReadMessages␣%d\n",THIS_NODE_NUM);
14 #endif
15
16   counter=0;
17
18 #ifdef MULTIPLE_RECEPTION_BUFFERS
19   for (sender=0;sender<NUM_NODES;sender++)
20   /* For all possibles senders */
21   {
22     if (sender==THIS_NODE_NUM)
23     /* I will not recept from myself */
24       continue;
25 #else
26   {
27     sender = 0;
28 #endif
29
30     rt_sem_wait(&Map_sem);
31
32 /* I dont have to receive in this slot */
33 /* from this sender */
34     if (ReceptionTable[slot][sender]<1)
35     {
36       rt_sem_signal(&Map_sem);
37 #ifdef MULTIPLE_RECEPTION_BUFFERS
38       continue;
39 #else
40       return;
41 #endif
42     }
43     rt_sem_signal(&Map_sem);
44
45     if (buffer[sender]==NULL)
46       return;
```

```
47
48    pointer = buffer[sender] + sizeof (int);
49    npkg = *((int *)buffer[sender]);
50    if (npkg>0)
51      rt_printk("Recebendo␣do␣nodo:␣%d
52            Numero de packs
53            a receber: %d\n",sender,npkg);
54
55    for (i=0;i<npkg;i++)
56    {
57      buf=NULL;
58      if (*(int *)pointer==BEGIN_TAG)
59      /* Everythings is ok! */
60      {
61        pointer += sizeof(int);
62        PHeader = (THeader *)pointer;
63        pointer+=sizeof(THeader);
64
65        /* This is part of the last package! */
66        if (LastReadPackage[PHeader->connectionID]!=NULL)
67        {
68          buf = LastReadPackage[PHeader->connectionID];
69        } else
70        {
71          buf = rtc_alloc_package();
72          if (buf==NULL)
73          {
74            rt_printk("No␣memory!\n");
75            return;
76          }
77          memcpy(&buf->header,PHeader,sizeof(THeader));
78          buf->header.part_len = 0;
79          /* The actual size is 0 (new package)*/
80        }
81
82        rt_printk("Receive␣size:␣%d\n",PHeader->part_len);
83
84        if (buf->header.datalen<=MAX_SIZE_PACKAGE)
85        {
86          memcpy (buf->data+buf->header.part_len,
87                        pointer,PHeader->part_len);
88          pointer+=PHeader->part_len;
89          buf->header.part_len+=PHeader->part_len;
90          if (*(int *)pointer==END_TAG)
91          /* everything ok */
92          {
93            pointer +=sizeof(int);
94            if (buf->header.part_len>=buf->header.datalen)
```

```
95            /* All the package has been received */
96            {
97              rt_printk("End␣of␣package\n");
98              InsertToPackageList (&(OutputPackageList
99                            [buf->header.connectionID]),
100                         buf,1);
101           /*Insert in the begin of the buffer[sender]*/
102
103             counter++;
104             LastReadPackage[PHeader->connectionID] = NULL;
105             if (PHeader->connectionID!=NON_RT_CONNECTION)
106             /* if is not non-rt connection */
107               rt_sem_signal
108                  (&(receive_sem[PHeader->connectionID]));
109               /* One package has been received */
110
111           } else
112           {
113             LastReadPackage[PHeader->connectionID] = buf;
114           }
115         } else
116         {
117           rt_printk("No␣END_TAG!!\n");
118           if (buf!=NULL)
119             rtc_free_package(buf);
120           LastReadPackage[PHeader->connectionID] = NULL;
121         }
122       } else
123       {
124         rt_printk("Package␣so␣big!!\n");
125         rtc_free_package(buf);
126         LastReadPackage[PHeader->connectionID] = NULL;
127       }
128
129     } else
130     {
131       rt_printk("ERROR!␣No␣BEGIN_TAG.\n");
132       LastReadPackage[PHeader->connectionID] = NULL;
133       break;
134     }
135   }
136   *((int *)buffer[sender]) = 0;
137   *((int *)buffer[sender] + sizeof (int))=END_TRANS_TAG;
138 }
139
140 #ifdef DEBUG
141   printk("I␣have␣received␣%d␣msg.
142           Node:%d\n",counter,THIS_NODE_NUM);
```

```
143 #endif
144 }
```

Bibliografia

[ARA 93] ARAS, Çaglan; KUROSE, James. REEVES, Douglas. SCHULZRINNE, Henning. **Real-Time Communication in Packet-Switched Networks**. Carolina do Norte, USA: Universidade de Carolina do Norte, 1993.

[AND 91] ANDREWS, Gregory R. **Concurrent Programming**. 2nd ed. Redwood City, California: The Benjamin/Cummings Publishing Company, 1991.

[ARV 91] ARVIND, K.; RAMAMRITHAM, Krithi; STANKOVIC, John A. A local area network architecture for communication in distributed real-time systems. **Journal of Real Time Systems**, Boston, USA, v.3, n.4, p.115, 1991.

[BAC 98] BARCELOS, Patrícia Pitthan de Araújo. **Comunicação em sistemas distribuídos tempo real**. 1998. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[BAR 2000] BARRETO, Marcos Ennes. **Deck**: Um Ambiente de Programação Paralela em Agregados de Multiprocessadores. 2000. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[BEC 99] BECK, Michael; BÖHME Harald; DZIADZKA, Mirko; KUNITZ, Ulrich; MAGNUS, Robert; VERWORNER, Dirk. **Linux Kernel Internals**. Harlow, Inglaterra: Addison-Wesley, 1999.

[BIA 2000] BIANCHI, E.; MANTEFAZZA, P.; DOZIO, L. **DIAPM-RTAI**. Milão, Itália: Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, 2000.

[BIA 2001] BIANCHI, E.; DOZIO, L.; MANTEGAZZA, P. **RTAI- A hard real-time support for Linux**. Milão, Itália: Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano, 2001.

[BOD 95] BODEN, N. et al. **Myrinet**: a gigabit-per-secound local-area network. Los Alamitos, USA: IEEE Micro, 1995.

[BOV 2001] BOVET, Daniel P.; CESATI, Marco. **Understanding the Linux Kernel**. Sebastopol, USA: O'Reilly, 2001.

[BUR 2001] BURNS, Alan; WELLINGS, Andy. **Real-Time Systems and Programming Languages**. 3rd ed. Harlow, Inglaterra: Addison-Wesley, 2001.

[BUT 2000] BUTTAZZO, Giorgio C. **Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications**. Boston, USA: Kluwer Academic Publishers, 2000.

[CAM 98] CAMPELLO, Rafael Saldanha. **RealGroup:** uma Ferramenta para Comunicação de Grupo Tempo Real no Sistema Operacional QNX. 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[COU 2001] COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems - Concepts and Design**. 3rd ed. Harlow, Inglatera: Addison-Wesley, 2001.

[CUN 99] CUNNINGHAM, David; LANE, William. **Gigabit Ethernet Networking**. Inglaterra: Macmillan Technical Publishing, 1999.

[GAL 99] GALE, Tony; MAIN, Ian. **Gtk v1.2 Tutorial**. Disponível em: <`www.gtk.org`>. Acesso em: 15 fev. 2002.

[HEL 99] HELLWAGNER, Hermann; REINEFELD, Alexander. **SCI**: Scalable Coherent Interface. Alemanha: Springer, 1999.

[HUM 2001] HUMBERTO, Carlos; MITIDIERI, Athaíde. **Comunicação em sistemas de tempo real**. 2001. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[KAN 93] KANDLUR, Dilip D.; SHIN, Kang G.; FERRARI, Domenico. **Real-time communication in multi-hop networks**. USA: IEEE, 1993.

[KOP 97] KOPETZ, Hermann. **Real-Time Systems**. Norwell, USA: Kluwer Academic Publishers, 1997.

[LAN 2002] LANKES, Stefan; JABS, Andreas. **A time-triggered Ethernet protocol for real-time corba**. Washington DC, USA: ISORC, 2002.

[LAN 2000] LANKES, Stefan; PFEIFFER, Michael; BEMMERL, Thomas. **Design and implementation of a sci-based real-time corba**. Aachen, Alemanha: RWTH, 2000.

[LIU 2000] LIU, Jane W. S. **Real-Time Systems**. 2nd ed. USA: Prentice Hall, 2000.

[MAL 95] MALCOLM, Nicholas; ZHAO, Wei. **Hard Real-Time Communication in Multiple-Access Networks**. Boston, USA: Kluwer Academic Publishers, 1995.

[MAN 2000] MANTEGAZZA, Paolo. **Diapm rtai - beginner's guide**. Milão, Itália: Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano, 2000.

[MAN 2000a] MANTEGAZZA, Paolo; BIANCHI, E.; DOZIO, L.; ANGELO, Mike; BEAL, David; CLOUTIER, Pierre; HUGHES, Stuart; KISS, Gabor; MOTYLEWSKI, Tomasz; PAPACHARALAMBOUS, Steve; SCHLEEF, David. **DIAPM RTAI Programming Guide 1.0**.

Milão, Itália: Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano, 2000.

[MIT 2002] MITTAL, A.; MANIMARAN, G.; MURTHY, C. **Dynamic real-time channel establishment in multiple access bus networks**. USA: IEEE Computer Communications, 2002.

[MOU 2000] MOUROT, Patrick. **RTAI Internals Presentation**. França: Alcatel, 2000.

[MUK 99] MUKHERJEE, Sarit; SAHA, Debanjan; SAKSENA, Manas C; TRIPATHI, Satish K. A distributed scheduling algorithm for real-time communication on slotted shared medium. **Journal of Parallel and Distributed Computing**, USA, v.58, p.1–25, 1999.

[NOR 2000] NORDEN, S.; MANIMARAM, G.; MURTHY, C. Dynamic planning based protocols for real-time communication in lan and switched lan environments. **Computer Communications**, USA, v.24, n.13, p.1256–1271, 2001.

[OLI 2001] OLIVEIRA, Rômulo Siva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. Porto Alegre: Instituro de Informática da UFRGS, 2001.

[RUB 98] RUBINI Alessandro. **Linux Device Drivers**. Sebastopol, USA: O'Reilly & Associates Inc., 1998.

[RUS 99] RUSLING, David A. **The Linux Kernel**. Wokingham, Inglaterra: [s.n.], 1999. Disponível em: <`www.ldp.org`>. Acesso em: 20 mar. 2002.

[RYA 97] RYAN, Stein Jorgen. **The design and implementation of a portable driver for shared memory cluster adapters**. Oslo, Noruega: Universidade de Oslo, 1997. Relatório técnico.

[SAN 91] SANTOS, J. **Redes Locales en Tiempo Real**. Nova Friburgo: EBAI, 1991.

[STA 84] STALLINGS, William. Local networks. **ACM Computing Surveys**, USA, v.16, n.1, 1984.

[TAN 97] TANENBAUM, Andrew S. **Redes de computadores**. 2nd ed. Rio de Janeiro: Campus, 1997.

[TAN 2001] TANENBAUM, Andrew S. **Modern Operating Systems**. 2nd ed. USA: Prentice Hall Inc., 2001.

[TIN 94] TINDELL, Ken. **Analysis of Hard Real-Time Communications**. Inglaterra: Universidade de York, 1994.

[TOD 2000] TODD, Robert W.; CHIDESTER, Matthew C.; GEORGE, Alan D. **A direct control for real time SCI**. [S.l.]: High Performance Computing and Simulation Research Laboratory, 2000.

[VEN 96] VENKATRAMANI, Chitra. **The Design, Implementation and Evalutation of RETHER**: A Real-Time Ethernet Protocol. 1996. Tese (Doutorado em Ciência da Computação) – Departamento de Ciência da Computação, Universidade Estadual de Nova York, USA.

[VER 91] VERÍSSIMO, P.; RODRIGUES, L.; RUFINO, J. **Delta-4 - A Generic Architecture for Dependable Distributed Computing**. Alemanha: Springer, 1991.

[ZHE 00] ZHENG, Qin.; SHIN, Kang G. **Synchronous bandwith allocation in FDDI netwoks**. USA: The University of Michigan, 2000.