Canal Virtual de Tempo-Real

Fernando Sales Panont

Canal Virtual de Tempo-Real

Fernando Sales Panont

Orientador: Prof. Dr. Célio Estevam Móron

São Carlos, 24 de novembro de 2008

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P195cv

Panont, Fernando Sales.
Canal virtual de tempo-real / Fernando Sales Panont. -- São Carlos : UFSCar, 2008.
78 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2008.

1. Tempo real. 2. Rede de computadores. 3. Protocolos de comunicação. 4. Sistemas de transmissão de dados. I. Título.

CDD: 004 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Canal Virtual de Tempo-Real"

FERNANDO SALES PANONT

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Célio Estevan Moron
(Orientador – DC/UFSCar)

Prof. Dr. Luis Carlos Trevelin
(DC/UFSCar)

Profa. Dra. Itana Maria de Souza Gimenes
(Din/UEM)

São Carlos
Março/2006

Dedicatória

Dedico esta dissertação, de todo o coração, a Deus e a meus pais.

Resumo

Devido ao crescimento de suporte nos sistemas operacionais de propósito geral às aplicações de tempo-real e ao aumento do número de interfaces de rede com múltiplas tecnologias de comunicação em um mesmo dispositivo computacional, um *middleware* com características de tempo-real foi projetado e desenvolvido. Este *middleware* encapsula estas tecnologias e protocolos de comunicação de modo a aumentar a tolerância a falhas e, consequentemente, melhorar o cumprimento dos prazos de entrega de mensagens através do fornecimento de alternativas de conectividade. Além disso, Canal Virtual de Tempo-Real provê um mecanismo de endereçamento virtual de pontos de acesso, onde cada endereço faz o mapeamento de um conjunto de pontos de acesso reais da ponta consumidora. Para a tradução destes endereços em pontos de acesso reais, servidores de nomes, conhecidos como *Brokers*, também foram utilizados.

Abstract

Due to the increase of general purpose operational systems support for real-time applications and by the increasing number of network interfaces with multiple communication technologies present in one single computational device, a *middleware*, with real-time characteristics, was designed and developed. This *middleware* encapsulates these communication technologies and protocols in order to improve fault tolerance and consequentially comply with deadlines for message transmission through connectivity options supply. Moreover, the Real Time Virtual Channel provides a virtual access point address mechanism, where each address maps a set of real access points from the consumer endpoint. For the translation of these addresses in real access points, name servers, known as *Brokers,* were also implemented.

Sumário

Lista de Figuras

LISTA DE TABELAS

1 INTRODUÇÃO

Esta seção apresenta a contextualização, motivações e os objetivos para a realização do trabalho.

1.1 Contexto do trabalho

Os sistemas de tempo-real vêm evoluindo há várias décadas, de pequenos dispositivos isolados para sofisticados sistemas distribuídos de tempo-real compostos por computadores comuns fabricados em larga escala (COTS – *commercial-off-the-shelf*). Durante esta evolução, os sistemas operacionais de propósito geral também evoluíram no suporte às aplicações de tempo-real não-crítico, disponibilizando recursos como prioridades de execução entre processos, relógios de alta precisão, *kernel preemptivo*, entre outros, altamente desejáveis para que se possa cumprir os prazos (*deadlines*) de suas tarefas. Além disso, os dispositivos computacionais que primordialmente continham apenas um tipo de tecnologia de dispositivos de rede começaram a incorporar mais de uma tecnologia em um mesmo dispositivo. Exemplo disso são os *handhelds* Palm LifeDrive e Palm TX [1] dotados de tecnologias de comunicação sem-fio *Bluetooth*[2], Irda [3] (Infravermelho) e *Wi-Fi* 801.11b[4], ou *laptops* com interfaces de rede *Wi-fi,* Infravermelho e *Ethernet*. Com isso, aumenta-se o leque de escolhas de tecnologias para uma transmissão de dados.

1.2 Motivação

Devido a estas várias alternativas de conectividade proporcionadas pelo aumento do número de dispositivos de rede em um único aparelho e ao suporte a aplicações de tempo-real pelos sistemas operacionais de propósito geral (sistemas voltados para o usuário comum), surgiu a idéia do desenvolvimento de um *middleware* com características de tempo-real não-crítico que encapsulasse estas tecnologias de comunicação e protocolos de comunicação, de modo a aproveitar esta gama de meios de transmissão e protocolos de comunicação presentes nos dispositivos computacionais para aumentar a tolerância a falhas e, conseqüentemente, melhorar o cumprimento dos prazos de entrega de mensagens através do fornecimento de alternativas de conectividade.

1.3 Objetivo

Este trabalho tem como objetivo modelar e implementar um *middleware* de Tempo-Real que auxilie as trocas de mensagens de tempo-real não-crítico entre tarefas, permitindo que haja continuidade das transmissões de dados sob falha na comunicação. Para alcançar tais objetivos, o Canal Virtual de Tempo-Real será dotado das seguintes características:

- ***Independência de* protocolos de comunicação:** Através de um *Framework* de protocolos de Comunicação, o *middleware* de tempo-real consegue uma independência de protocolos de comunicação, fazendo com que as mesmas funções para leitura e escrita de dados pela aplicação usuário sejam utilizadas com diferentes protocolos de comunicação. Para completar, um endereçamento virtual é oferecido pelo *middleware*, não necessitando assim que o mesmo fique sujeito a um determinado endereçamento de algum protocolo em específico, como por exemplo, o endereçamento do protocolo TCP/IP IPv4.
- **Tolerância a falhas por redundância de pontos de acesso de conexão:** Permite que uma conexão, que normalmente seria encerrada devido à degradação no meio de transmissão de dados, possa continuar através de um ponto de acesso alternativo que ofereça outra rota do canal Produtor de dados ao canal Consumidor.
- **Transparência de localização**: Os Servidores de nomes ou *Brokers* oferecerão transparência de localização aos processos envolvidos na comunicação.

1.4 Estrutura da Dissertação

O texto desta dissertação encontra-se dividido da seguinte forma:

- Capítulo 2 – Sistemas de Tempo-Real e características de tempo-real do sistema operacional Linux: apresenta uma visão geral e questões relacionadas aos sistemas de Tempo-Real, aos sistemas operacionais de tempo-real e às características de tempo-real suportadas pelo sistema operacional Linux, utilizado na implementação da arquitetura do *middleware*.

- Capítulo 3 – Estado da Arte: mostra algumas arquiteturas de *middleware* e suas funcionalidades. Estas arquiteturas serviram como modelo para a arquitetura descrita no capítulo 4.
- Capítulo 4 – Projeto dos Canais Virtuais de Tempo-Real: definição da arquitetura do *middleware* de Tempo-Real, e funcionamento interno do Canal Virtual de Tempo-Real.
- Capítulo 5 – Estudo de Caso: Discorre sobre o estudo de caso utilizado para avaliação da arquitetura do Canal Virtual de Tempo-Real.
- Capítulo 6 – Conclusões e trabalhos futuros: Apresenta conclusões finais sobre o trabalho assim como propostas para a sua extensão.

2 SISTEMAS DE TEMPO-REAL E CARACTERÍSTICAS DE TEMPO-REAL DO S.O. LINUX

Este capítulo visa esclarecer alguns conceitos e problemas básicos importantes sobre os sistemas de tempo-real, conceitos básicos sobre sistemas operacionais de tempo-real e as características de tempo-real encontradas no sistema operacional de propósito geral Linux, utilizado no desenvolvimento do Canal Virtual de Tempo-Real.

2.1 Sistemas de Tempo-Real

Os sistemas de tempo real estão cada vez mais presentes no cotidiano da sociedade, das mais variadas formas imaginadas, como os sistemas de controle embutidos em lavadoras de roupas, forno microondas, aparelhos de DVDs, sistemas de controle de frenagem ABS, para citar alguns.

Um sistema de tempo-real é qualquer sistema ou atividade de processamento de informação a qual tem que responder a estímulos de entrada com um finito e especificado período [5]. Dessa forma, o sistema que garante o cumprimento destes requisitos temporais é referenciado como Sistema de Tempo Real (STR).

2.2 Classificação dos sistemas de Tempo-Real

Os Sistemas de Tempo-Real variam muito em relação à complexidade e às necessidades de garantia no atendimento de restrições temporais. Devido a estas variações pode-se subdividir os sistemas de tempo-real em duas classes:

- Tempo-Real Rígido (*Hard Real-Time*): segundo [6], sistemas em que a perda de um único prazo (*deadline*) resulta em uma falha catastrófica. Exemplo deste tipo de sistema são os sistemas de usinas nucleares, de controle de aeronaves, entre outras. A perda de um sinal para resfriamento do reator de uma usina nuclear é catastrófica.
- Tempo-Real não Crítico (*Soft-Real Time*): Sistemas que necessitam atender o maior número possível de *deadlines*, em que perder uma não causa uma falha

catastrófica, mesmo que isto não seja desejável [6]. Um fluxo de imagens em uma apresentação de vídeo precisa ser mostrado ao usuário sem muitas interrupções, às quais inviabilizariam a sua apresentação.

2.3 Problema da Inversão de Prioridade

Nos sistemas de tempo-real, é possível que de uma tarefa com maior prioridade fique bloqueada a espera de uma de menor prioridade. A esta anomalia, dá-se o nome de inversão de prioridade [7]. Este fenômeno ocorre frequentemente na troca de mensagens entre processos, no qual um produtor de mensagens de alta prioridade fica bloqueado a espera do consumidor que é postergado pelo escalonamento de outros processos de maior prioridade, aumentando assim as chances da tarefa produtora perder seus *deadlines*. Para minimizar esta anormalidade o conceito de Herança de Prioridades [8] pode ser utilizado. Esta técnica consiste em aumentar a prioridade da tarefa de menor prioridade ao nível da de maior prioridade até que a de menor prioridade libere o recurso compartilhado. Finalmente, ao término da dependência de recursos entre os processos de maior e menor prioridade, o processo de menor prioridade retoma sua prioridade original.

2.4 Sistemas Operacionais de Tempo-Real

Os sistemas operacionais de tempo-real (SOTR) diferem dos sistemas operacionais de propósito geral basicamente pela sua eficiência e previsibilidade temporal no atendimento a requisições de seus serviços.

A maioria das aplicações de tempo real possui uma grande parte de suas funções sem restrições temporais. Logo, é preciso considerar que um SOTR deveria, além de satisfazer as necessidades dos processos de tempo real, fornecer funcionalidade apropriada para os processos convencionais, tais como sistema de arquivos, interface gráfica de usuário e protocolos de comunicação para a Internet.

Aspectos temporais estão relacionados com a capacidade do SOTR em fornecer os mecanismos e as propriedades necessários para o atendimento dos requisitos temporais da aplicação tempo real. Uma vez que tanto a aplicação como o SOTR compartilham os mesmos recursos do *hardware*, o comportamento temporal do SOTR afeta o comportamento temporal

da aplicação. Isto significa que a capacidade da aplicação atender aos seus *deadlines* depende da capacidade do sistema operacional em fornecer os serviços solicitados em um tempo que não inviabilize aqueles *deadlines*.

2.5 Sistema Operacional de propósito geral Linux

O critério principal do projeto do *kernel* do Linux é o *throughput*, sem se preocupar com o determinismo e previsibilidade, como visto em [9]. Apesar do linux ser um sistema operacional de propósito geral, este contêm algumas características de tempo-real que são importantes para tarefas de tempo-real não-crítico. Algumas delas já fazem parte do *kernel* do sistema operacional enquanto outras podem ser adquiridas através de extensões de *kernel*.

2.5.1 Escalonador O(1)

O novo escalonador O(1), introduzido por Ingo Molnar, fornece um escalonamento completo com complexidade algorítmica O(1) [6].

O núcleo do novo escalonador é composto de dois vetores - 'vetor ativo' que contém todas as tarefas que são afiliadas da CPU e 'vetor expirado' que contém todas as tarefas, que já consumiram suas fatias de tempo. Esta solução de divisão em dois vetores permite haver um número arbitrário de tarefas, e o recálculo de fatias de tempo pode ser feito imediatamente quando a fatia de tempo esgota.

Este novo escalonador também oferece três políticas de escalonamento diferentes, uma para processos normais e duas para aplicações de tempo-real. Um valor estático de prioridade é atribuído a cada processo. Todo escalonamento é preemptivo: Se um processo com uma prioridade estática estiver pronto para executar, o processo corrente deixará o processador e retornará a sua fila de espera. A política de escalonamento apenas determina o ordenamento junto a lista de processos executáveis de mesma prioridade estática.

Como dito previamente, o escalonador oferece três políticas de escalonamento.

- SCHED_FIFO – Escalonamento *First in-First Out*: Podem somente ser usadas com prioridades estáticas maiores que zero. Isso significa que quando um processo com tal política se tornar ativo, este sempre interromperá um processo de política não-tempo real SCHED_OTHER [10]. SCHED_FIFO é um algoritmo simples de

escalonamento sem fatias de tempo [10]. Um processo em SCHED_FIFO executará até ser bloqueado por uma requisição de E/S, por ser interrompido por um processo de maior prioridade ou por sua própria desistência do processador.

- SCHED_RR – Escalonamento *Round Robin*: O SCHED_RR é um aperfeiçoamento do SCHED_FIFO, utilizando fatias de tempo em uma lista circular, onde o processo corrente é colocado no final da fila de escalonamento quanto acabar sua fatia de tempo e posteriormente reaver o processador.
- SCHED_OTHER – Escalonamento padrão do Linux: Pode ser usado somente a uma prioridade zero. O escalonamento SCHED_OTHER é utilizado por todos os processos do sistema que não necessitem de mecanismos de prioridade estática de tempo-real. O processo a ser executado é escolhido da lista estática de prioridade zero, baseada em uma prioridade dinâmica que é determinada somente dentro desta lista.

2.5.2 Memória Compartilhada

Um dos principais métodos de comunicação usados por aplicações de tempo-real é memória compartilhada. Com isso múltiplos processos podem mapear e compartilhar dados em memória, sendo um meio eficiente para transmitir grande quantidade de dados entre os processos.

2.5.3 Travamento de tarefas em memória

Tarefas de aplicações devem ser travadas em memória devido ao longo atraso aleatório introduzido quando a RAM é exaurida e o *swap* é requerido, sendo inaceitável em um sistema de tempo-real.

Dessa forma, o Linux suporta funções que desativam a paginação de um trecho de memória em especifico. Assim, toda memória “travada” ficará lá até que o processo termine ou que este destrave a memória.

2.5.4 *Threads* POSIX

As *threads* POSIX são mapeadas cada uma para um processo de Linux. Isto significa que cada *thread* terá o seu próprio identificador e serão escalonadas pelo escalonador do Linux, utilizando uma das políticas de escalonamento definidas anteriormente.

LinuxThreads implementa a maior parte da API de POSIX: Mutex, variáveis de condição, cancelamento, sinais, etc. A biblioteca também fornece semáforos de POSIX para sincronização entre as *threads*.

3 MECANISMOS DE COMUNICAÇÃO DE TEMPO-REAL

Este capítulo visa descrever alguns mecanismos de comunicação de tempo-real, que foram tomados como base na definição da arquitetura do Canal Virtual de Tempo-Real.

3.1 Virtuoso™ VSP

Criado e desenvolvido pela Eonic Systems e atualmente mantido pela empresa Wind River Systems Inc, o Virtuoso VSP é um *kernel* de tempo-real totalmente distribuído. Sua arquitetura genérica é baseada em um *nanokernel* pequeno, mas muito rápido e um *microkernel* portável e preemptivo. Este sistema operacional de tempo-real (SOTR) permite escrever aplicações totalmente paralelas com nenhuma ou pequenas alterações no código fonte, até mesmo quando é mudado o tipo dos processadores utilizados. Dessa forma, desenvolvedores podem aumentar ou diminuir sua aplicação e seguir a curva de tecnologia com um pequeno esforço e sem precisar conhecer todos os detalhes do hardware [11].

Este *kernel* trabalha com o modelo VSP (*Virtual Single Processor*), onde o desenvolvedor trabalha com o sistema paralelo da mesma forma que se estivesse trabalhando em um sistema mono-processado podendo ser um sistema heterogêneo ou não. Para facilitar a construção deste tipo de sistema, o Virtuoso apresenta as seguintes características:

- Multiprocessamento para qualquer número de tarefas;
- Escalonamento preemptivo de tarefas por prioridade;
- Comunicação e sincronização entre processos através de semáforos, mensagens filas e *timers*;
- Recursos para gerenciamento de memória;
- Troca rápida de contexto;
- Necessidade de pouca memória para execução;
- Facilidades para o gerenciamento do *hardware* em diversos níveis de abstração.

O Virtuoso oferece um sistema de programação organizado em diversos níveis, como ilustra a Figura 1.

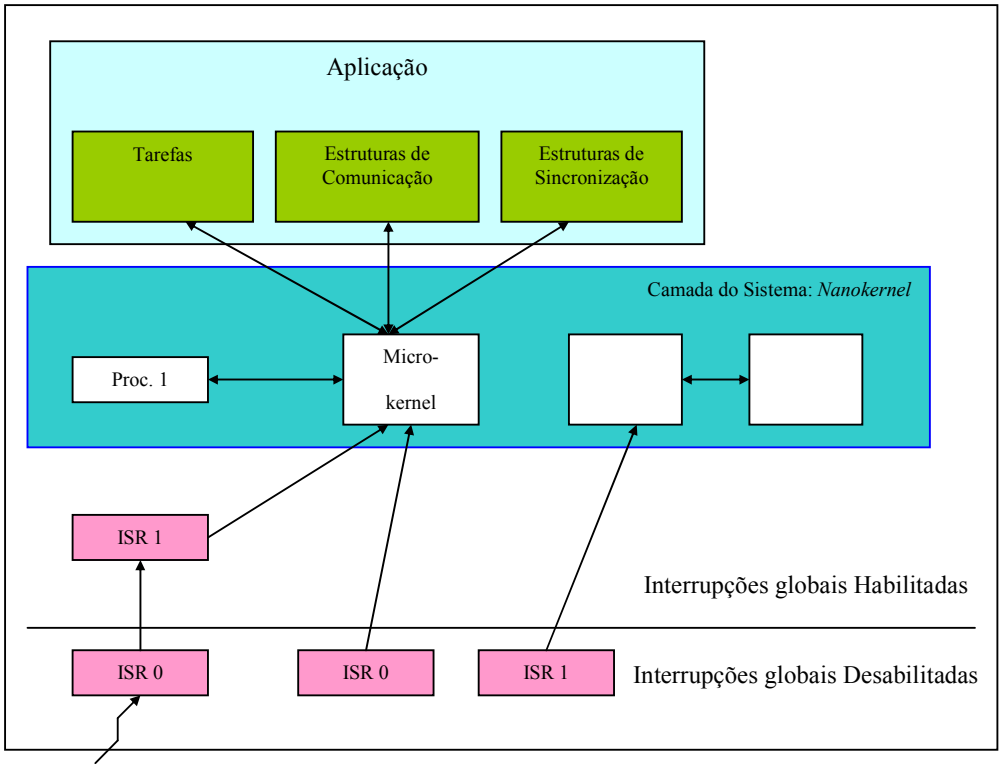

Figura 1: Níveis de programação do kernel Virtuoso

Os níveis ISR (*Interrupt Service Routine*) são usados para controlar diretamente as interrupções de *hardware* do processador. Dependendo da arquitetura utilizada, pode-se ter mais de um nível de ISR.

O nível 3 (*nanokernel*) é composto de diversas tarefas de contexto reduzido, camadas e processos. Cada processo começa e termina como uma rotina *assembly*, pode chamar funções em linguagem C [12]. O processo de nível 3 responsável pelo gerenciamento do escalonamento de tarefas é chamado de *microkernel*.

O nível 4 é composto pelos serviços oferecidos pelo *kernel* Virtuoso. São mais de 70 serviços do *microkernel*, que podem ser chamados a partir da linguagem C. Os componentes básicos deste nível são os objetos do *microkernel*. Cada objeto é identificado por um nome único no sistema. Estes podem ser vistos como sendo instâncias de classes de objetos definidos pelo Virtuoso. Cada classe tem um conjunto de atributos e suporta um conjunto específico de operações.

Na fase de desenvolvimento, o programador define quais objetos serão usados e escolhe em qual processador estes objetos serão direcionados, no caso de um sistema multiprocessado.

Para prover transparência na comunicação interprocesso distribuída, o SOTR Virtuoso provê um objeto denominado Canal de Comunicação, que será descrito a seguir.

3.1.1 Canais de Comunicação do Virtuoso™

Um Canal de Comunicação do Virtuoso™ (Eonic, 2000) se comporta muito como um objeto do *kernel* Virtuoso FIFO [12] (*first-in-firs-out*) unidirecional, que pode ser usado para troca de dados entre tarefas. Estes canais são considerados como "*pipes*" de software que proporcionam a tarefa colocar dados de um lado e a outra retirá-los do outro. Além disso, eles podem se comunicar anonimamente, isto é, o emissor envia uma mensagem sem identificação da origem da mensagem. As diferenças entre Canais e objetos FIFO no Virtuoso™ são:

- Um número variável de (8 bits) bytes pode ser escrito e lido de um canal, diferentemente ao tamanho fixo de blocos por tempo dos objetos FIFOs.
- Um determinado canal pode ser configurado pelo usuário como *bufferizado*[1] ou não-*bufferizado*.
- A API do canal tem um conjunto rico em opções que habilitam influenciar no comportamento em caso de leituras e escritas incompletas.
- Pode haver comunicação síncrona ou assíncrona.

3.1.2 Mecanismo dos Canais

Como ilustra a Figura 2, o canal é um objeto virtual gerenciado pelo Virtuoso™, que toma conta da transferência de dados do escritor em uma ponta do canal para o leitor na outra ponta. O canal consiste em uma fila de escritores e leitores e um *buffer*[2] do canal opcional. Estes são enfileirados na ordem com que chegam ao canal e por nível de prioridade.

[1] *Bufferizado*: utilizar-se de memória intermediária
[2] *Buffer*: Memória intermediária para armazenamento de informação

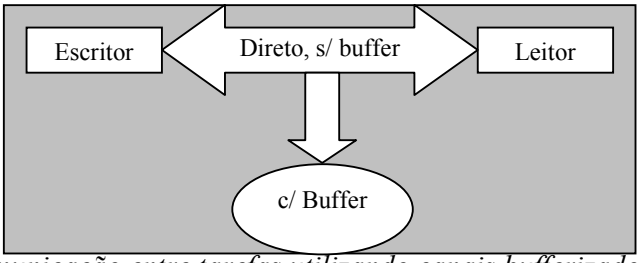

Figura 2: Comunicação entre tarefas utilizando canais bufferizados/não-bufferizados

Utilizando comunicação não-*bufferizada*, os dados irão fluir diretamente do escritor para o leitor.

Quando usando a opção de *buffer* no canal, os dados possivelmente serão copiados primeiramente para o *buffer*, e em seguida transferidos para o leitor. Explicaremos melhor a seguir.

3.1.3 Bufferizando dados

Quando criamos um projeto no Virtuoso, podemos decidir se um objeto canal pode ser ou não com *buffer* e definir o tamanho do *buffer*. Isto não pode ser modificado em tempo de execução. *Bufferização* pode ser útil, por exemplo, quando uma tarefa está gerando dados em pequenos pacotes, enquanto a tarefa processante opera em grandes blocos.

3.1.4 Com Bufferização

Quando um canal é *bufferizado*, a seqüência de eventos é a seguinte:

- O escritor chama *KS_ChannelPut()* e o leitor chama o *KS_ChannelGet()* (não necessariamente nesta ordem).
- Se o leitor já está esperando, o *buffer* não é usado. Os dados são diretamente transferidos para a área dos leitores.
- Se não há leitores esperando, os dados são transferidos para o *buffer* primeiro. Neste caso, o escritor pode ser liberado assim que os dados são copiados para o *buffer*. Quando um leitor chega posteriormente, os dados são transferidos do *buffer* para o leitor.

3.1.5 Sem Bufferização

Quando um canal não é *bufferizado*, a seqüência de eventos é a seguinte:

- O escritor chama o *KS_ChannelPut()* e o leitor o *KS_ChannelGet()* (não necessariamente nesta ordem).
- Se o leitor está esperando os dados, estes são transferidos diretamente para a área dos leitores.
- Se não há um leitor esperando, não poderá haver transferência, e o escritor deve esperar (se desejado).

Poderá ocorrer uma situação complexa onde existe uma incompatibilidade entre os dados providos pelo escritor e o que pode ser consumido pelo leitor. Neste caso, os dados transferidos são carregados em estágios, como por ex, uma grande escrita (1) sendo consumida em pequenas porções pelo leitor (1A, 1B, 1C), como ilustra a Figura 3.

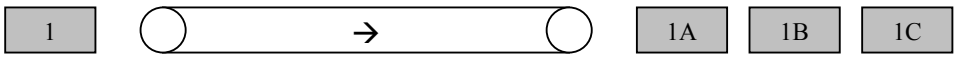

Figura 3: Incompatibilidade entre os dados provenientes pelo escritor e o que pode ser consumido pelo leitor.

3.2 Sistema Distribuído de tempo-real Java baseado em CSP

Processos Seqüenciais que se Comunicam [13], ou CSP, é uma linguagem formal usada para descrever sistemas paralelos criada por C. A. R. Hoare no começo dos anos 80. O CSP tem sido largamente usado na especificação, na análise e na verificação de sistemas concorrentes de tempo-real e para entendimento que situações que podem acontecer quando a concorrência se faz presente. Ele provê uma boa notação e suporte teórico, ajudando especificações e projetos serem claramente expressados e entendidos. Isto viabiliza a análise e validação do projeto [14].

Os canais CSP implementados em Java retiram os perigos do uso da programação multitarefa por um conceito de projeto não ambíguo de fácil entendimento [15]. O conceito de canal que foi implementado em Java trata de ambientes com um único processador com múltiplos processadores e também faz escalonamento de tempo-real baseado em prioridades. Este escalonador não está anexado ao sistema operacional, mas sim aos canais de comunicação. Assim, pode haver vários escalonadores executando em diferentes partes do programa [15].

3.2.1 Canais CSP

A idéia básica do CSP como definida por Hoare [13] é que processos paralelos ou concorrentes podem trabalhar em conjunto sincronizando suas respectivas entras e saídas. A comunicação entre um processo A e um processo B somente ocorre se A estiver pronto para enviar o dado e B para receber. Assim que uma dessas condições seja verdadeira, o processo é colocado para esperar (desescalonado) até que o outro processo esteja pronto.

O pseudocódigo abaixo ilustra dois processos comunicantes via um canal.

```
CHANNEL chan        //declaração do canal
PAR                 // construção PAR: os  processo abaixo desta construção
                    //executam em paralelo
ProcessoA(chan)
        { chan ! x;...}     //escrevendo x no canal
ProcessoB(chan)
        { chan ? y;...}     //lendo x do canal
```

3.2.2 Canais Java

O termo Canais Java implica que estes utilizam os canais CSP exclusivamente. Os canais Java são objetos intermediários, passivos, compartilhados pelos objetos ativos comunicantes, como ilustra a Figura 4. Os Canais Java são unidirecionais, inicialmente não-*bufferizados*, e totalmente sincronizados. O resultado disso é que o programador é liberado de construções complicadas de sincronização e escalonamento [15].

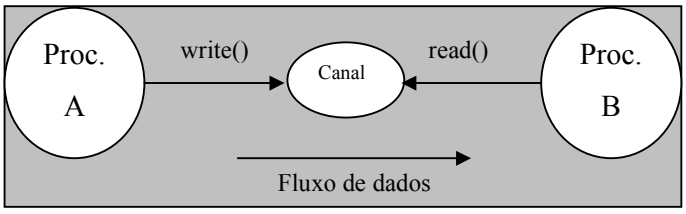

Figura 4: Abstração de um Canal de comunicação

3.2.3 O conceito de Link Driver

Para evitar o desenvolvimento de canais especiais para cada periférico, um *framework* de *device driver* é desenvolvido. Os *device drivers*, que chamamos de *link drivers*, são objetos especiais dependentes que podem ser anexados ao canal. O objeto canal cuidará da sincronização e escalonamento enquanto o *link driver* da transferência de dados.

Canais entre processos em um mesmo processador usam o *driver* de memória compartilhada e os canais entre processos em processadores distintos usam um *driver* periférico. O resultado são processos independentes do *hardware*. Haverá uma visível separação entre os objetos dependentes de *hardware* e os independentes, como ilustra a Figura 5.

Os métodos `read()`e `write()` são, quando permitidos pelo mecanismo de sincronização do objeto, delegados ao *link driver*. As Figuras 6 e 7 ilustram respectivamente a comunicação entre dois processos em um mesmo processador e a comunicação entre dois sistemas.

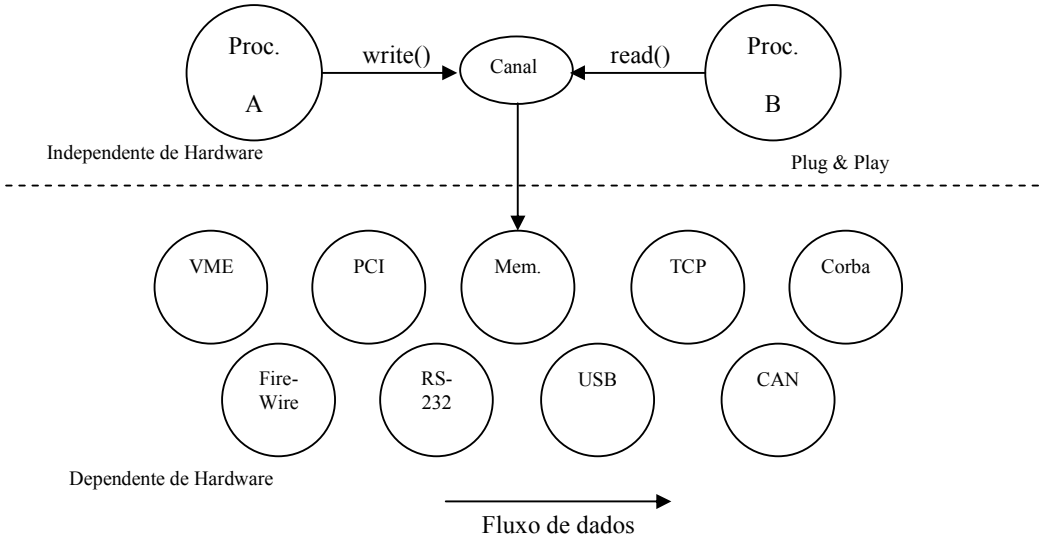

Figura 5: Framework plug & play para dispositivos

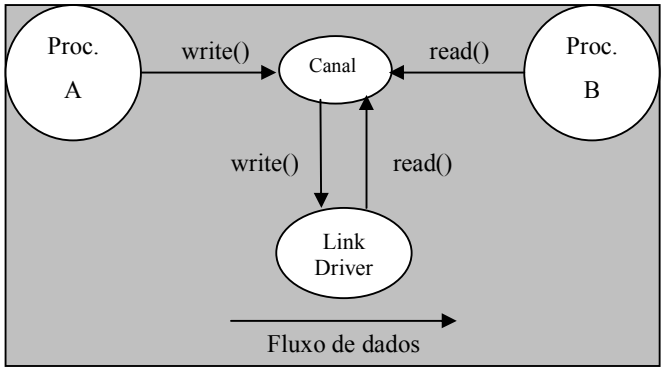

Figura 6: Transferência de dados sobre link drivers para um sistema uniprocessado

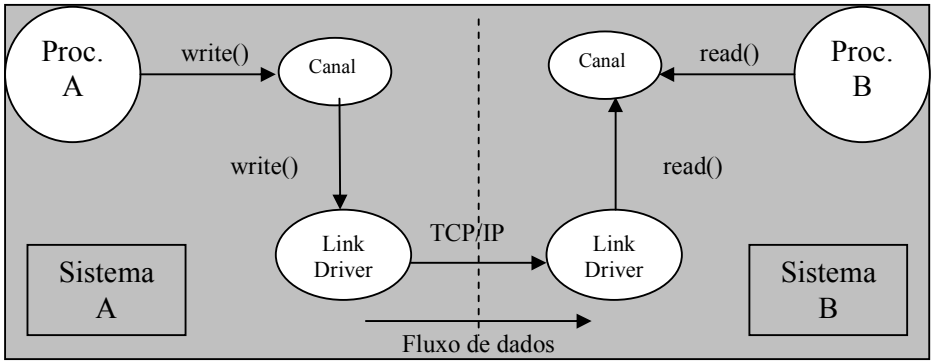

Figura 7: Transferência de dados sobre link drivers para um sistema com múltiplos processadores

3.3 RTC: UM MIDDLEWARE de comunicação de tempo-real sobre o Linux de tempo-real RTAI

Conforme [16], o RTC é um *middleware* de comunicação inspirado no padrão ISO/OSI [OSI], que implementa uma plataforma completa de comunicação de tempo-real em um cluster executando sobre o Linux de tempo-real RTAI [18]. Para prover uma comunicação de tempo-real, várias camadas foram implementadas, desde o gerenciamento de *hardware* e controle de acesso ao meio até a API (*Application User Interface*). Estas camadas foram implementadas por meios de componentes reusáveis. Para a camada MAC, RTC usa um protocolo TDMA (*Time Division Multiple Access*) ligeiramente modificado, apesar disso, isto pode ser facilmente modificado pela plataforma ser baseada em componentes. No nível de aplicação, uma abordagem orientada a canal com reserva de largura de banda é utilizada, adequada para aplicações multimídia. Para tráfego não tempo-real, poderá ser utilizado, por exemplo, o protocolo TCP/IP.

Atualmente, o RTC utiliza padrão IEEE 1596-1992 SCI (*Scalable Coherent Interface*) [19] [20] como suporte à rede de comunicação. O SCI foi escolhida devido a sua latência de sub-micro segundo sobre distâncias de dez metros, baixo *jitter* e velocidade de *link* de 1GB/s, conforme [21]. No entanto devido a estrutura modular do RTC, outras redes de comunicação como, por exemplo, *Infiniband* [22] [23], também poderão ser suportadas, simplesmente substituindo o mesmo módulo.

3.3.1 Arquitetura RTC

A plataforma de comunicação RTC baseada em componentes consiste em um conjunto de camadas que adicionam serviços necessários a um sistema operacional de tempo-real. Para isto, um sistema operacional de tempo-real que habilite esse tipo de modificação era necessário. Assim, o Linux com a extensão de tempo-real RTAI foi utilizado.

A plataforma RTC tem dois componentes principais: os protocolos de comunicação e a API em nível de usuário. Os protocolos de comunicação são implementados como um conjunto de camadas, onde cada camada é implementada como um módulo do sistema operacional Linux/RTAI [24], cada qual correspondendo a uma funcionalidade específica do sistema. A API em nível de usuário é baseada na característica LXRT [25] do RTAI. A Figura 8 dá uma visão geral da arquitetura da plataforma RTC, que será detalhada nas sessões subseqüentes.

Além das funcionalidades de tempo-real, a camada poderá prover também mensagens que não são de tempo-real. Estas mensagens, por não serem de tempo-real, obtém menor prioridade que as demais de tempo-real. Para prover essa funcionalidade, um *driver* de rede para Linux que implementa uma comunicação não tempo-real foi implementada. Para a conexão não tempo-real entre nós da rede poderá utilizar-se, por exemplo, o protocolo TCP/IP.

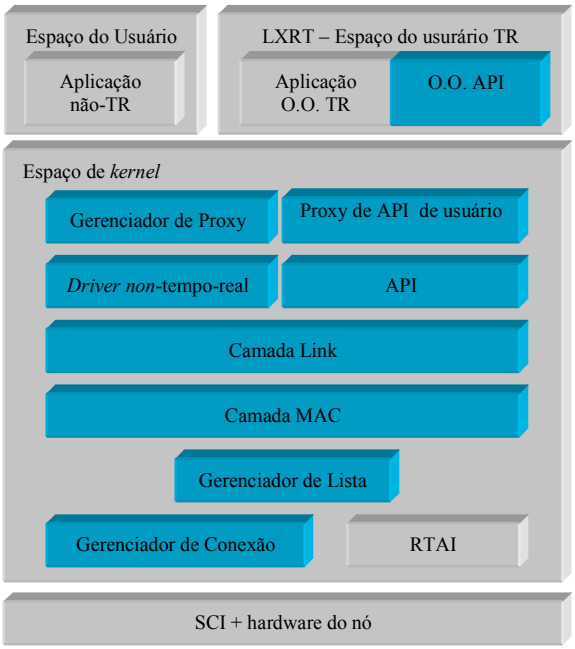

Figura 8: Visão geral da arquitetura da plataforma RTC

O paradigma de comunicação provido pelo hardware é escondido pela plataforma RTC. Assim, do ponto de vista do usuário, as conexões entre tarefas são baseadas em canais (orientados a conexão) que conectam diferentes nós na rede.

Na plataforma RTC, cada canal é unidirecional, e é definido por:

- Nó origem: nó que envia a mensagem.
- Nó destino: nó que recebe a mensagem.
- Identificação do canal: utilizado para gerenciamento interno
- Largura de banda: a largura de banda reservada para este canal

Na plataforma RTC, o sistema deve ser configurado estaticamente, isto é, todos os canais que serão utilizados devem ser definidos com o sistema desligado e a configuração de suas conexões é feita durante o início do sistema. Durante a fase de definição, todo o sistema é analisado de forma a confirmar a disponibilidade de recursos que são necessários para preencher as necessidades da aplicação.

3.3.2 Gerenciador de conexão

Este módulo gerencia a alocação de um seguimento de memória dedicado e o compartilhamento deste seguimento, para que uma comunicação possa ser efetuada entre dois nós. Esta camada também provê aos demais módulos os endereços dos seguimentos de memória oferecidos. Este módulo é padrão do Linux e não tem restrições de tempo-real, por ser relevante somente na fase inicial do sistema.

O reconhecimento da chegada de pacotes é manipulado por outros meios e será explicado mais adiante nas sessões subseqüentes.

3.3.3 MAC (*Media Access Control*)

O Controle de Acesso ao Meio, ou MAC, é o principal módulo da plataforma RTC. Este módulo é responsável pelo controle do acesso ao meio compartilhado (neste caso a memória compartilhada), isto é, disciplinar o acesso aos seguimentos compartilhados de memória. As principais tarefas deste módulo são:

- Inicia o sistema de tempo-real RTAI
- Controle de acesso ao meio compartilhado.

- Sincronização de relógios entre os nós SCI da rede.

Para o protocolo MAC, o protocolo TDMA foi escolhido devido a suas vantagens já conhecidas, que são a alocação estática de largura banda e seu pequeno *jitter* de transmissão.

Na abordagem TDMA, a largura de banda é estaticamente dividida em um número fixo, chamado lacunas (*slots)*. Dessa forma, durante um período fixo de tempo (*time-slot*), somente um nó terá acesso ao meio de comunicação. O número de lacunas não depende do número de nós, assim um nó pode usar mais de uma lacuna. Depois que todos os nós tiverem usado suas respectivas lacunas, o ciclo é repetido. O ciclo completo, onde todos os nós têm acesso à suas lacunas, é chamado de rodada-TDMA (TDMA-*round*) ou somente rodada (*round*). Na atual implementação do RTC, é assumido que todas as rodadas são iguais [16].

Como o protocolo TDMA é baseado na divisão de tempo, os relógios dos nós da rede devem ser sincronizados. Na implementação corrente do RTC, é tido como barreira para sincronização o final de cada rodada, onde cada nó espera por um sinal do nó mestre (que é um dos nós da rede) para começar uma nova rodada. É importante frisar que o tempo absoluto do sistema de cada nó não é sincronizado por este método, somente o contador de tempo dos módulos MAC nos vários nós sincronizados.

Como visto anteriormente, somente um nó deve usar o acesso ao meio de comunicação a cada lacuna de tempo. No entanto, este acesso representa a escrita no seguimento compartilhado de memória remota. Para receber o pacote, o nó destino deve usar qualquer lacuna seguinte (usadas para acesso de escrita dos outros nós). Dessa forma, a disciplina do meio deve ser aplicada somente aos acessos de escrita, pois a operação de leitura é local. Na situação onde um nó recebe informação de vários outros nós, para cada nó deve ser reservado um seguimento de memória para ser possível ler seus pacotes.

3.3.4 Gerenciamento de lista de pacotes

O principal objetivo desta camada é gerenciar a lista de pacotes a serem enviados/recebidos pelo sistema. A necessidade de manter múltiplas listas de pacotes faz isto necessário implementar um gerenciador centralizado de listas.

3.3.5 Camada de Link

A utilização correta da largura de banda de cada lacuna, para determinar como os dados serão fragmentados em uma transmissão e recompostos no nó receptor, é a principal finalidade da Camada de *Link*. As principais tarefas deste módulo são as seguintes:

- Gerência de canal
- Envio de pacotes no canal de comunicação
- Recebimento de pacotes enviados por outros nós
- Detecção de erro

Esta camada recebe os pacotes do módulo API e os inserem no meio de comunicação. Quando o pacote é recebido por completo este é passado para o módulo API via filas de entrada e saída, onde cada fila pertence a um canal. A camada de Gerenciamento de Lista de Pacotes oferece primitivas para acesso destas listas.

3.3.6 API

O objetivo desta camada é fornecer a aplicações do *kernel* um conjunto de primitivas de comunicação de alto nível e também esconder todos os detalhes internos dos procedimentos RTC. Ela também monta as mensagens da aplicação, transformando-as em pacotes a serem enviados à camada abaixo contendo o cabeçalho com a informação requerida. As principais tarefas da camada API são:

- Prover primitivas de comunicação para a aplicação
- Montar/desmontar as mensagens para /da camada de *Link*
- Registrar as tabelas de comunicação na camada de link.

As primitivas disponíveis para a aplicação permitem receber e enviar mensagens de tamanho variável e de forma assíncrona. Para enviar mensagens, duas primitivas são providas:

- **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 função faz uma requisição de envio de uma mensagem sobre um canal, que pode conter uma simples *string* até um tipo de dado complexo, armazenada na memória

apontada pela variável *msg*, de tamanho *msg_size*. A mensagem é montada e enfileirada de maneira FIFO (*first in first out*). A segunda função tem o mesmo comportamento, contudo, a mensagem é colocada na frente da fila, assim esta mensagem terá uma prioridade maior do que as demais mensagens da fila. Ambas as funções são não bloqueantes.

Para receber mensagens, a camada API disponibiliza duas primitivas:

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

Estas primitivas são usadas para receber uma mensagem de forma bloqueante ou não bloqueante, respectivamente. Ambas as funções recebem as mensagens pelo canal *channel*, com o tamanho de *msg_size* e copiam esta para a área de memória apontada por *msg* e retornam o número de bytes que puderam ser lidos.

A camada API é responsável também por registrar tabelas que gerenciam as operações de leitura e escrita.

3.3.7 Gerenciador de Proxy

O Gerenciador de *Proxy* é responsável por criar uma nova tarefa de *kernel* para cada aplicação executando no espaço de usuário. A aplicação de usuário de tempo-real chama o Gerenciador de *Proxy* pela API de usuário orientada a objeto que é uma biblioteca ligada junto a aplicação. Este *Proxy* irá, em fila, mediar a comunicação entre a aplicação do usuário que executa no espaço de tempo-real de usuário provido pelo LXRT e os módulos RTC em nível de *kernel*.

3.3.8 API Proxy de Usuário

O objetivo deste módulo é chamar a API *kernel* que esta debaixo da aplicação do usuário. Cada *Proxy* fica no estado de espera até que uma aplicação requeira um serviço RTC. Quando uma aplicação de tempo-real de usuário faz uma requisição de um serviço de comunicação, o Gerenciador de *Proxy* acorda a API de *Proxy* de Usuário que chama a API *kernel*. O resultado da chamada é retornado para aplicação. Em outras palavras, o *Proxy*

fornece uma ponte entre o espaço de usuário e o espaço de *kernel*, dando suporte a chamada da API de aplicações que executam no espaço de usuário de tempo-real.

3.3.9 API de Usuário Orientada a Objeto

A API de Usuário é uma biblioteca ligada junto a aplicação com o propósito de prover uma API de maneira orientada a objeto a aplicação em nível de usuário de tempo-real.

Quando uma aplicação executa um procedimento de envio da API, dependendo do destino, duas ações podem ser feitas. Caso o destino for o mesmo nó, a API de Usuário resolve a comunicação utilizando os serviços internos de comunicação providos pelo RTAI. Caso contrário, se o destino for outros nós, a API de Usuário executa o Gerenciador de *Proxy* para acessar os serviços RTC. Para as funções de recebimento, o procedimento é análogo ao descrito para as funções de envio de mensagens.

3.4 SERVIÇOS de comunicação e *middleware* ARMADA

ARMADA é um conjunto de serviços de comunicação e *middleware* que provêem suporte para tolerância a falhas e garantias fim-a-fim para aplicações distribuídas de tempo-real embutidas [26]. Os desafios abordados pelo projeto ARMADA incluem o envio temporizado de serviços fim-a-fim com restrições de tempo-real não-crítico ou tempo-real crítico (*hard real-time*), dependência de serviços em presença de falhas de *hardware* ou *software*, escalabilidade de recursos computacionais e comunicação, e exploração de sistemas abertos e padrões emergentes em sistemas operacionais e serviços de comunicação.

Este projeto foi divido em três áreas. A primeira é focada no desenho e desenvolvimento de serviços de comunicação de tempo-real para *microkernel*. Uma arquitetura genérica é introduzida para desenhar a comunicação do subsistema nos *hosts*. Assim a previsibilidade e qualidade de serviço são garantidas. A arquitetura é independente de um serviço de comunicação em particular. Ela é ilustrada no contexto do projeto ARMADA como canais de tempo-real, ou seja, um serviço de baixo nível que implementa uma conexão virtual simples, ordenada entre dois *hosts* em rede, provendo garantias de atraso determinísticas ou estáticas fim a fim. A segunda área do projeto tem seu foco em uma coleção modular de serviços de *middleware* que podem ser montadas como blocos de

construção para desenvolver aplicações para sistemas embutidos [27]. Uma arquitetura aberta em camadas suporta inserção ou implementação de novos serviços modulares conforme os requerimentos evoluem no decorrer da vida do sistema. Os serviços do *middleware* ARMADA incluem um conjunto de serviços de comunicação em grupo tolerantes a falha com garantias de tempo-real, chamado RTCAST, para dar suporte a aplicações embutidas com requerimentos de tolerância a falhas. RTCAST consiste em uma coleção de *middleware* incluindo um serviço de afiliação a grupo, serviço de *multicast* atômico temporizado, um módulo de controle de admissão e escalabilidade, e um serviço de sincronização de relógios. O *middleware* ARMADA inclui também um serviço de replicação de *backup* primário de tempo-real, chamado RTPB, que assegura a consistência de forma temporária de objetos replicados em nós redundantes. A terceira e última área do projeto são as ferramentas de validação e avaliação de restrições de tempo e capacidade de tolerância a falhas no sistema alvo.

A Figura 9 resume a estrutura do ambiente ARMADA.

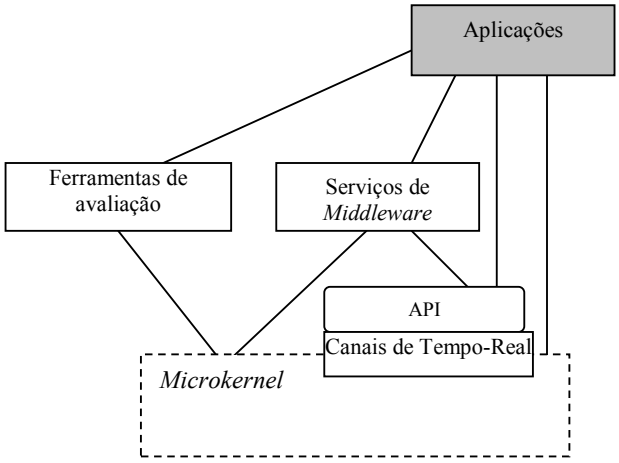

Figura 9: Visão geral do ambiente ARMADA

Para facilitar o desenvolvimento de serviços orientados a comunicação, o subsistema de comunicação do ARMADA é implementado utilizando o *x-kernel*, *framework* de rede de comunicação orientado a objeto estendido para alocação controlada de recursos de sistema. A vantagem do uso do *x-kernel* é a fácil composição de pilhas de protocolos. O subsistema de comunicação *x-kernel* é implementado como um grafo de objetos de protocolos configurável.

Isto facilita a reconfiguração da pilha de protocolos por deixar possível a inserção e remoção de protocolos.

Seguindo a filosofia de *microkernel*, os serviços foram projetados como servidores multitarefa no nível de usuário. Clientes dos serviços são processos em separado que comunicam com o servidor via *kernel* utilizando a biblioteca. Esta biblioteca exporta a API *middleware* desejada. Serviços orientados a comunicação geralmente implementam sua própria pilha de protocolos que residem sobre o *driver* de comunicação no nível do *kernel*. Entretanto, o *framework x-kernel* permite que execução da pilha de protocolos multitarefa execute em nível de usuário e permite que esta migre para o *kernel* do sistema operacional. Dessa forma, servidores e protocolos de comunicação podem ser desenvolvidos no nível de usuário e posteriormente podem migrar para o *kernel*. Essa migração melhora a desempenho por eliminar a troca de contexto e também devido as *threads* terem maior prioridade (por padrão) no nível de *kernel* [26]. Pelas *threads* terem maior prioridade no nível de *kernel,* o servidor executa de forma muito mais previsível, fazendo com que o serviço não fique à espera de processamento durante sobrecargas.

As Figuras 10 e 11 ilustram respectivamente as configurações dos servidores em nível de usuário e em nível de *microkernel*.

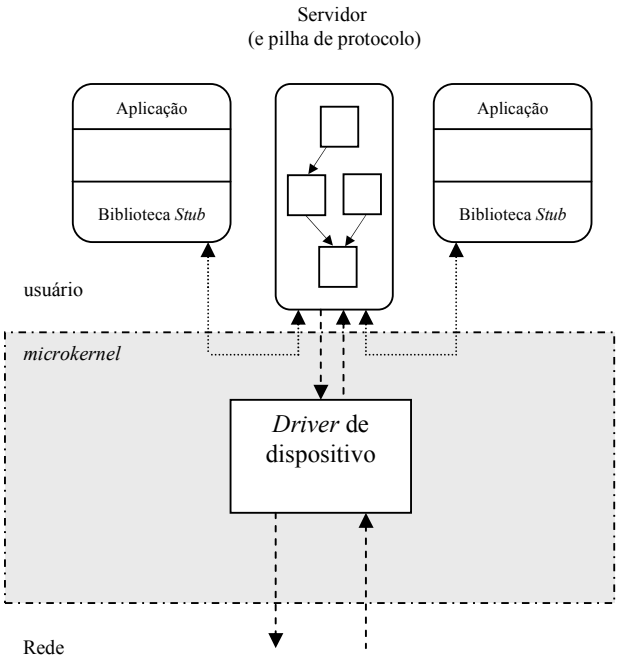

Figura 10: Configuração do servidor em nível de usuário

Um exemplo de migração de servidor para o *kernel* é a migração do servidor RTCAST, que, depois de desenvolvido no espaço de usuário, foi reconfigurado e integrado ao *kernel* (como na Figura 11). Independente do servidor estar executando em nível de usuário ou de *kernel*, a aplicação usará a mesma API para se comunicar com ele. O serviço é colocado no *kernel,* e, dessa forma, uma troca de contexto de ou para o servidor em nível de usuário é salva. O *Stub*, gerado automaticamente, conecta a biblioteca do usuário (que implementa a API de serviço) ao *microkernel* ou ao processo servidor. Estes *stubs* escondem detalhes do mecanismo local de comunicação do programador, assim faz o código do serviço independente das características específicas do *microkernel* utilizado.

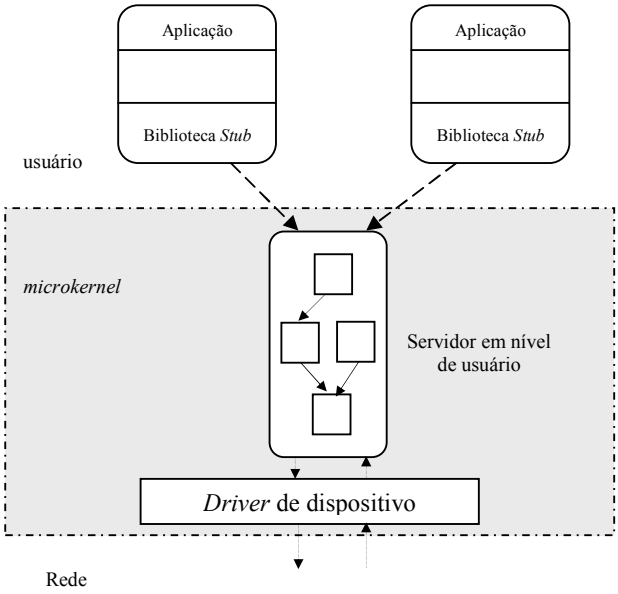

Figura 11: Servidor em nível de kernel

3.4.1 Arquitetura de comunicação de tempo-real ARMADA

A arquitetura primária do serviço de comunicação de tempo-real consiste em quarto componentes: uma API de comunicação de tempo-real (API RTC), um protocolo para sinalizar e reservar recursos (RTCOP), suporte para gerenciamento de recursos e transferência de dados em tempo de execução (CLIPS), e suporte a perfil de execução. Estes componentes, juntos, com um conjunto de políticas específicas de usuário, podem implementar vários modelos de comunicação de tempo-real. A Figura 12 dá uma visão geral da arquitetura de comunicação de tempo-real ARMADA. As linhas pontilhadas indicam o caminho de controle

tomado na arquitetura durante a fase de configuração. Depois da fase de configuração, os dados são transferidos da API RTC para o CLIPS, como representados pelas linhas sólidas.

3.4.1.1 CLIPS – Biblioteca para Implementação de Semânticas de Prioridades

Segundo [26], a Biblioteca de Comunicação para Implementação de Semânticas de Prioridades, ou CLIPS, foi desenvolvida com o propósito de disponibilizar mecanismos de gerenciamento de recursos para prover transferência de tempo-real de dados com qualidade de serviço em uma conexão estabelecida. Uma ponta da comunicação chamada *clip*, garante um determinado rendimento em termos de número de pacotes enviados por ele por período. Um *clip* é criado para cada um dos dois comunicantes do canal de comunicação de tempo-real. Uma ou mais conexões (ou *sockets*) podem ser "ligados" ao mesmo *clip*, e cada *clip* reserva recursos do processador e memória no sistema final, suficientes para garantir qualidade de serviço na conexão. Internamente a cada *clip*, existe uma fila de mensagem para armazenar temporariamente as mensagens geradas ou recebidas no canal correspondente, uma tarefa manipuladora de comunicação para processar estas mensagens, e uma fila de pacotes para dirigir pacotes à espera para serem recebidos ou transmitidos.

O CLIPS é utilizado para implementar o serviço de comunicação baseado na garantia de qualidade serviço chamado Canal de Tempo-Real [26]. Um Canal de Tempo-Real é uma conexão virtual *unicast* entre duas tarefas (de origem e de destino) com garantias de desempenho associadas ao atraso da mensagem, e a largura de banda disponível. A comunicação de tempo-real via canais de comunicação é feita em três fases. Na primeira fase, o *host* de origem "O" cria um canal junto ao *host* destino "D", especificando os parâmetros de tráfego do canal e requerimentos de qualidade de serviço. Sinais de requisição são enviados de "O" para "D" via um ou mais nós intermediários e, depois disso, respostas são entregues na direção reversa, ou seja, de "D" para "O". A segunda fase baseia-se no envio de mensagens de "O" para "D", caso a conexão se estabeleça com sucesso. A última fase se dá no encerramento explícito da conexão por parte da origem "O". Dessa forma, os recursos reservados para este canal são liberados.

3.4.1.1.1 Escalonamento sensível a qualidade de serviço de CPU

A tarefa que lida com a comunicação de um *clip* executa em um laço contínuo, retira as mensagens de saída da fila de mensagens do *clip* e as fragmenta, ou retira os pacotes que

chegam da fila e os remonta em mensagens. Cada mensagem deve ser enviada dentro de um prazo de tempo (*deadline*) interno. De forma a alcançar uma melhor utilização do escalonador, o escalonamento da comunicação é baseado na política de *earliest-deadline-first* (EDF), onde é a mensagem com o menor prazo de envio é processada primeiro. Como a maioria dos sistemas operacionais não implementa este tipo de escalonamento, o CLIPS o implementa em nível de usuário. O escalonador EDF executa em uma prioridade estática, e mantém uma lista em ordem crescente de *deadline* de todas as tarefas registradas junto a ela. Em um dado certo momento, o escalonador CLIPS bloqueia todas as tarefas, exceto a tarefa com o prazo mais curto. Esta tarefa executa até o seu final ou até liberar voluntariamente o processador. Ao término de sua execução, a tarefa é bloqueada, e a próxima com o menor prazo de termino execução começa a executar.

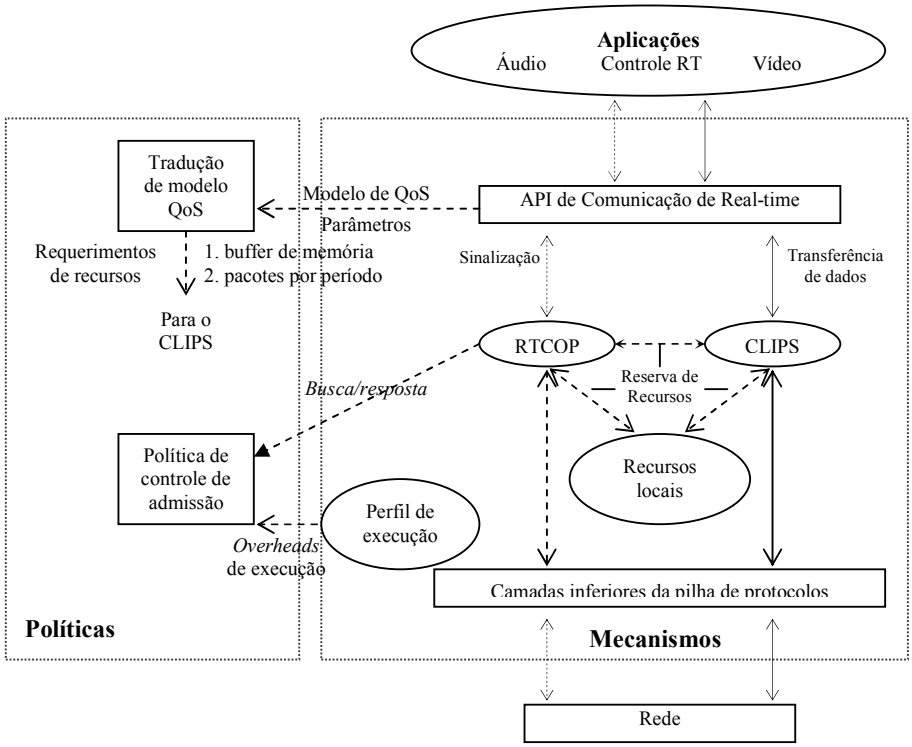

Figura 12: Arquitetura do serviço de comunicação de tempo-real

3.4.1.1.2 Reserva de Recursos

Manipuladores de comunicação (implementados pelo CLIPS) executam uma pilha de protocolos definida pelo usuário, e então retornam o código CLIPS depois de processar cada

mensagem ou pacote. Idealmente, cada *clip* pode ser atribuído a um reservador de CPU para prevenir que um cliente de comunicação monopolize a CPU. Como a maioria dos sistemas operacionais não implementa reserva de capacidade de processador, a reserva é indiretamente expressa em termos de número máximo de pacotes processados dentro de um período de tempo. O manipulador bloqueia a si mesmo depois de processar o número máximo de pacotes permitido dentro do seu período de tempo.

3.4.1.1.3 Policing

Este mecanismo assegura as políticas das conexões para que estas não excedam a sua especificação de tráfego feita na reserva de recursos de processador.

3.4.1.2 Alocação de largura de banda sensível a qualidade de serviço

Os sistemas operacionais atuais implementam a transmissão de pacotes como um FIFO (*First In First Out*) sobre o link de comunicação. Como isto não pode ser evitado, o CLIPS implementa um escalonador baseado em prioridade dinâmica, na parte inferior da pilha de protocolos de usuário, de forma a escalonar os pacotes a serem enviados de maneira priorizada. O escalonador de *link* implementa a política de escalonamento EDF utilizando uma pilha com prioridade para pacotes a serem enviados. Para prevenir uma acumulação na fila de saída de pacotes (ex.: enquanto o link está ocupado), o escalonador de *link* CLIPS não lança um novo pacote até que o pacote anterior seja transmitido e ele seja notificado. Os pacotes baseados na política de melhor esforço (*best effort*) são mantidos em uma pilha em separado, junto ao escalonador de *link* no nível de usuário e executado em uma prioridade menor que os demais *clips* de tempo-real.

3.4.1.3 Interface de Aplicação RTC

A interface de programação exportada aos aplicativos engloba rotinas de conexão para estabelecimento de conexão e desconexão, transmissão de mensagem e recepção durante a transferência de dados em conexões estabelecidas, iniciação e rotinas de suporte. A API tem duas partes: a parte superior e a parte inferior. A parte superior faz a interface entre as aplicações, e é responsável por validar as requisições das aplicações e criar um estado interno.

A parte inferior faz a interface para o RTCOP fazer a sinalização (ex.: configuração e desconexão), e faz interface com o CLIPS para uma transferência de dados sensíveis a qualidade de serviço.

O desenho da API RTC é baseado em grande parte na API *socket* do Unix BSD. Cada conexão é representada pelo par (Endereço IP, porta). O endereço IP e um número sem sinal de 16 bits do *host* e a porta é um número único no *host*.

O principal aspecto que diferencia a API RTC da API *socket* é que a aplicação explicitamente aprova o estabelecimento e finalização da conexão.

3.4.1.4 RTCOP – Protocolo de Ordenação à Conexão de Tempo-Real

Conforme [26], o RTCOP é um protocolo de sinalização fim-a-fim distribuído. Sua principal função é criar e destruir conexões através de sinais. A Figura 13 ilustra a composição interna do RTCOP, composta primariamente de dois módulos relativamente independentes. O módulo Manipulador de Requisições e Respostas gerencia os estados de sinalização e faz a interface entre o módulo de política de controle de admissão, enquanto o módulo de comunicação lida com a tarefa de encaminhar mensagens de sinais de forma confiável.

O modulo de comunicação implementa a supressão de duplicações de forma a assegurar que múltiplas reservas não são instaladas para a mesma requisição de conexão. De forma similar, isto também é aplicado ao encerramento de conexão onde todos os nós ao longo da rota devem liberar os recursos reservados e liberar o estado de conexão.

O RTCOP exporta uma interface a API RTC para especificação de requisições de requisições e respostas de estabelecimento e encerramento de conexão, e seleção de portas lógicas para conexão com a ponta final. O RTCOP também a interface para uma *engine* de roteamento para buscar uma rota que possa suportar os requerimentos de qualidade de serviço desejáveis, antes de iniciar a sinalização por novas conexões.

3.4.2 Serviços de Comunicação de Grupo RTCAST

Os serviços e comunicação de Grupo RTCAST é uma coleção de serviços que disponibilizam qualidade de serviço na comunicação, e um *middleware* modular e passível de composição para construir aplicações embutidas. O *middleware* pode ser dividido em dois

conjuntos de serviços relativamente independentes, os serviços de comunicação de grupo RTCAST e o serviço de replicação e *back-up* primário de tempo-real RTPB.

3.4.2.1 Protocolos RTCAST

O RTCAST provê serviços de comunicação por *multicast* e serviços de associação a grupos para aplicações de tempo-real tolerantes a falha. Como ilustrado na figura 13, o conjunto de serviços RTCAST inclui um serviço *multicast* atômico temporizado, um serviço de filiação a grupo e um serviço de controle de admissão. Os primeiros dois serviços são fortemente acoplados e assim considerados como um único serviço. Sincronização de relógios é tipicamente requerida por protocolos de tempo-real e é reforçada pelo serviço de sincronização de relógio. Para suportar portabilidade, uma camada de interface virtual de rede exporta uma abstração uniforme da rede. Finalmente, a camada do topo provê uma interface de programação para os processos de tempo-real do grupo.

O RTCAST procede como emissores em um anel lógico. Um turno de um processo começa quando um *token* [17] lógico chega, ou quando o seu tempo de espera pelo *token* termina. Depois da última mensagem, cada emissor envia por *multicast* um batimento cardíaco (*heartbeat*) que é usado para detecção de queda. Os destinatários detectam as mensagens perdidas utilizando números de seqüência e quando um processo detecta uma omissão de recebimento, este quebra. Cada processo, quando seu turno começa, checa por batimentos cardíacos não recebidos e elimina os membros quebrados, se algum, do grupo por enviar uma mensagem por *multicast*.

Em um anel de *token*, mensagens enviadas têm uma ordem natural definida pela rotação do *token*. A ordem da mensagem é reconstruída utilizando a camada de protocolo abaixo do RTCAST que detecta as mensagens que chegam fora de ordem e as reordena, encaminhando-as para o RTCAST. O RTCAST assegura que os membros “corretos” podem alcançar um acordo nos estados replicados por formular o problema como um membro do grupo. Desde que o estado de um processo é determinado pela seqüência de mensagens que este recebe, um processo que detecta uma omissão de recebimento de mensagem tira a si mesmo o grupo, assim mantendo o acordo entre os demais. Finalmente, a mudança de integrantes do grupo é comunicada exclusivamente por mensagens de mudança de membro utilizando este mesmo mecanismo de *multicast*. Isto garante um acordo na visão do grupo.

O serviço de controle de admissão é um protocolo distribuído que fica supervisionando os recursos de todos os processos do grupo. O protocolo transparentemente cria um *clip* em cada *host* que executa o grupo de processo para assegurar a vazão na comunicação e processamento de mensagens em um tempo restrito. Se não há necessidade de garantias temporais, esta camada pode ser omitida da pilha de protocolo para evitar *overhead.* A comunicação procederá à base do melhor esforço (*best effort*).

3.4.3 Serviço de Replicação backup-primário de Tempo-Real (RTPB)

Manter uma grande quantidade de estados consistentes de aplicação em um sistema distribuído, como na abordagem de máquinas de estado, pode envolver um *overhead* significante. Muitas aplicações de tempo-real, no entanto, conseguem tolerar pequenas inconsistências nos estados replicados. Assim, para reduzir *overhead* de gerenciamento redundante, o serviço de replicação *backup*-primário explora as semânticas de dados da aplicação por permitir que a cópia de segurança mantenha uma pequena cópia dos dados que residem no primário. A aplicação pode ter diferentes tolerâncias de estado para diferentes dados de objetos. Com dados suficientemente recentes, a cópia de segurança pode seguramente superar o *backup* primário. O *backup*[3] pode então reconstruir um estado consistente do sistema explorando os valores prévios e os valores novos. No entanto, o sistema deve assegurar que a distância entre os dados do primário e do *backup* é limitada dentro de uma janela de tempo pré-definida.

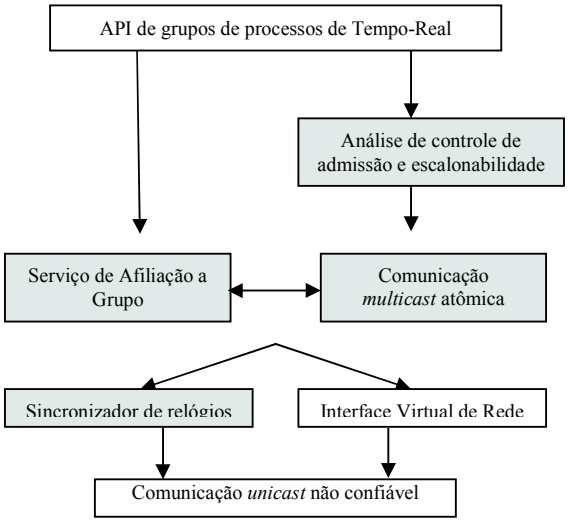

Figura 13: Conjunto de serviços RTCAST

[3] Cópia de segurança

Um *backup*-primário satisfaz a consistência temporal externa, ou seja, a consistência vista de uma perspectiva do tempo contínuo, de um objeto *i* se o *timestamp*[4] de *i* no servidor não for anterior a um pré-determinado tempo do *timestamp* no cliente, de forma que os dados do *backup* fiquem muito próximos de refletir o mundo atual do *backup* primário.

3.4.3.1 Implementação do RTPB

O serviço RTPB inclui um servidor primário e um servidor de *backup*. A aplicação cliente reside na mesma máquina que o primário. O primário é responsável por fazer uma cópia de segurança dos dados no *backup* e limitar a inconsistência dos dados entre os dois lados dentro de uma janela de tempo especificada.

Existem duas versões idênticas da aplicação cliente residindo nos *hosts* primário e *backup* respectivamente. Normalmente, somente a versão do cliente primário está executando. Contudo, quando o *backup* toma conta no caso de falha do primário, este também ativa uma versão de *backup* do cliente e atualiza-o com o estado mais recente do sistema.

3.4.4 Ferramentas de avaliação

Estas ferramentas têm o propósito de avaliar e validar a capacidade de cumprir as *deadlines* a capacidade de tolerância a falhas do sistema alvo. Duas ferramentas foram desenvolvidas para isto: ORCHESTRA, uma ferramenta de injeção de falhas para avaliação e validação dos protocolos de comunicação e *middleware*, e o COGENT, um gerador de carga de tráfego de rede.

3.4.4.1 Orchestra

ORCHESTRA é um injetor de falhas de ambiente que pode ser usado para fazer injeção de falhas nos protocolos de comunicação e nas aplicações distribuídas. Ele é baseado em um *framework* chamado *script-driven probing and fault injection*. A ênfase desta abordagem é

[4] Definição explicita de tempo ligada a uma informação.

nas técnicas experimentais com a intenção de identificar problemas no protocolo ou em sua implementação.

Na abordagem do ORCHESTRA, uma camada de injeção de falha é inserida dentro da pilha de protocolo de comunicação, abaixo do protocolo a ser testado. Assim que as mensagens são trocadas entre os participantes do protocolo, elas passam pela camada de injeção de falha. Cada momento que uma mensagem é enviada, o ORCHESTRA executa um script chamado *send filter* na mensagem. Da mesma maneira, também é aplicado o *receive filter* em cada mensagem recebida. O script faz os seguintes passos:

- Filtragem de Mensagem: para interceptar e examinar uma mensagem
- Manipulação de mensagem: para descartar, atrasar, desordenar, duplicar ou modificar uma mensagem.
- Injeção de Mensagem: para proibir um participante de introduzir uma nova mensagem no sistema.

3.4.4.2 CONGENT: Geração Controlada de Tráfego de Rede

CONGENT é um gerador de carga de trabalho sintética de rede para avaliar o desempenho de maneira controlada e reproduzível. O gerador é baseado em um modelo simples de cliente servidor que permite ao usuário modelar flexivelmente os recursos de rede de forma a avaliar vários aspectos da rede e da computação distribuída.

4 CANAL VIRTUAL DE TEMPO-REAL

O Canal Virtual de Tempo-Real é um *middleware* de serviços para comunicação interprocesso de forma transparente, independente de protocolos de comunicação e tolerante a falhas de comunicação. Este mecanismo de tolerância a falhas é alcançado através de redundância de pontos de acesso, cada qual ligado a uma interface de rede distinta.

Para atingir estes objetivos, uma especificação em módulos foi projetada. Cada módulo desta arquitetura tem um propósito específico e serão detalhados mais adiante.

Antes de detalhar a arquitetura e seus componentes, alguns tópicos importantes serão abordados para melhor compreensão da arquitetura e seu funcionamento.

4.1 Comunicação Interprocesso

Para ocorrer a comunicação interprocesso utilizando o *middleware* de tempo-real, deverá haver duas pontas prontas para a troca de dados, uma ponta produtora e outra consumidora, que daqui por diante serão chamadas de canal Produtor e canal Consumidor. Os canais Produtores serão responsáveis pelo envio de dados, enquanto os consumidores pelo recebimento destes dados.

4.2 Endereçamento Virtual dos Processos e Máquinas

De forma a promover uma independência de endereçamento de pontos de acesso para comunicação interprocesso, um sistema de endereçamento dos canais Produtores/Consumidores foi desenvolvido. Neste sistema, cada máquina recebe uma numeração única, chamada máscara e cada Canal Produtor/Consumidor é representado por um endereço virtual único, composto por um número decimal de nove dígitos, que será denominado endereço virtual. Este endereço virtual é formado pela concatenação da máscara com um número identificador local único do Canal Produtor/Consumidor no sistema local. Assim, os três dígitos de mais alta ordem deste endereço virtual representam a máquina à qual

Processo pertence e os demais dígitos de menor ordem o número único local do canal, como ilustrado na Figura 14 e exemplificado na Figura 15.

Figura 14: Composição de um número Exemplo de um Número identificador de canal

End. Virtual de canal: 666000123
- Máscara de Máquina: 666
- Id local do Canal Prod./Cons: 000123

Figura 15: Exemplo de um Número identificador de canal

4.3 Arquitetura dos Canais Virtuais

A arquitetura dos Canais Virtuais é baseada em módulos, onde cada um é encarregado de uma tarefa específica no *middleware*, que serão detalhadas mais adiante. A Figura 16 ilustra a arquitetura dos Canais Virtuais em conjunto com a aplicação do usuário, o *kernel* e seus protocolos de comunicação.

4.3.1 API de Comunicação de Tempo-Real

A API (*Application User Interface*) de Comunicação de Tempo-Real tem como objetivo fornecer à aplicação do usuário um conjunto de interfaces de serviços de comunicação bem definidas assim como definir a configuração destes serviços. As interfaces fornecidas para a aplicação do usuário por este módulo camada são as seguintes:

- ```
  int VirtualChannel(int process_type, int opt, char
  proc_name[], char peer_name[], idaddr_t my_id, struct
  accesspts my_accpts , struct accesspts peer_accpts);
  ```
- ```
  vcConfig(struct config options);
  ```
- ```
  int vcWriteto (idaddr_t my_id, char *data, unsigned
  datalen);
  ```
  ```

- `int vcReadfrom(idaddr_t my_id, char **data, unsigned *datalen);`

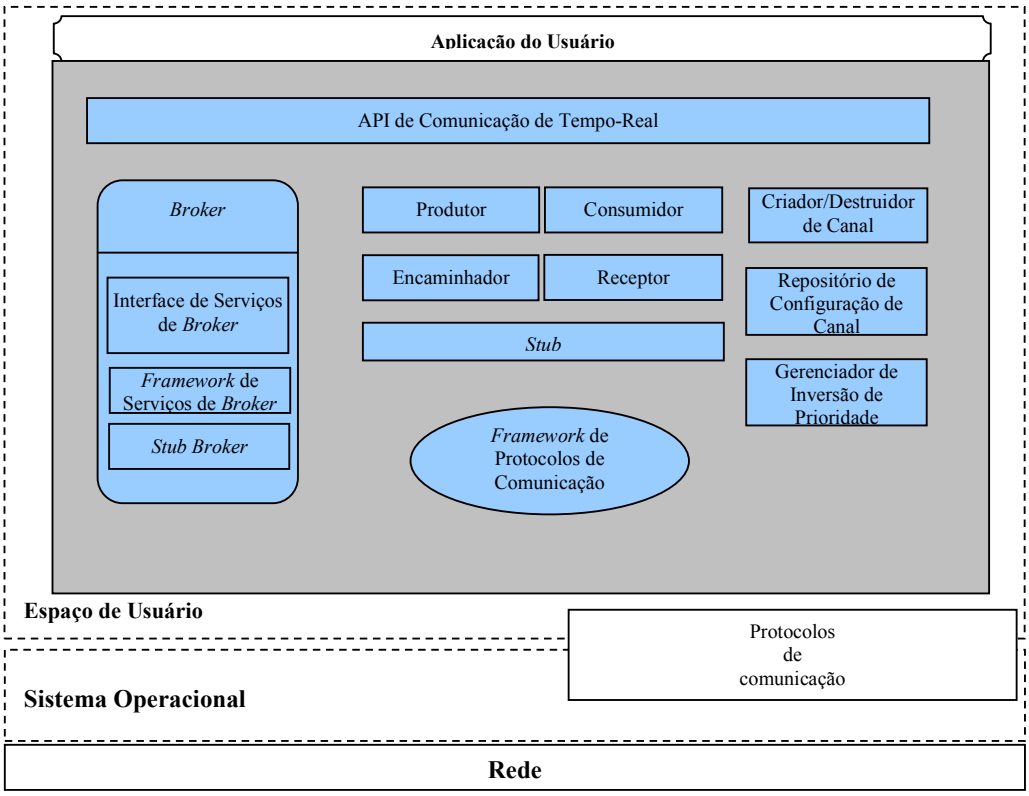

Figura 16: Arquitetura dos Canais Virtuais de Tempo-Real.

A primeira interface `VirtualChannel()` cria, inicia e destrói as estruturas internas do canal. Esta também dispara o Canal Produtor do *Broker* Interno para receber as mensagens enviadas pelo *Broker* Local (os *Brokers* Interno e Local serão descritos em detalhes na seção *Broker*). Para inicializar um canal, devemos passar os seguintes parâmetros para a função `VirtualChannel()`:

- **int process_type:** Tipo de canal. Deve-se receber o valor PRODUCER para canais do tipo produtor, CONSUMER para consumidores. Como os *Brokers* Local e Global também utilizarão das estruturas internas do Canal Virtual, estes deverão inicializar o canal com os valores *Broker* Local e *Broker* Global.
- **int opt:** Opção de inicialização do canal. Recebe o valor START para iniciar o canal e STOP para encerrar o mesmo.

- **char proc_name[]:** Nome fictício do canal. Utilizado pelos canais Consumidores no seu cadastro junto aos *Brokers*.
- **char peer_name[]:** Nome fictício do canal par. Utilizado pelos canais Produtores na tradução do nome em pontos de acesso do canal Produtor par.
- **idaddr_t my_id:** Número identificador do canal. Utilizado como um endereço virtual único para cada canal produtor e consumidor.
- **struct accesspts peer_accpts:** Pontos de acesso reais do canal Produtor/consumidor. Esta estrutura é composta por dois campos, um que aponta para estruturas descritoras de pontos de acesso reais para comunicação entre processos situados em uma mesma máquina, e outro para processos situados em máquinas distintas.

As Figuras 17, 18 e 19 mostram respectivamente as definições das estruturas *accesspt*, *udpaccpt* (utilizada na comunicação via protocolo UDP do protocolo TCP/IP) e *shmaccpt* (utilizada na comunicação via memória compartilhada), a ordem de execução das primitivas do Canal Virtual de Tempo-Real em uma aplicação, e um exemplo em liguagem C de inicialização de um canal do tipo Produtor.

```
struct accesspts{
    void *localaccesspt[NUMBER_OF_LOCAL_ACCESSPOINTS];
    void *rmtaccesspt[NUMBER_OF_REMOTE_ACCESSPOINTS];
};
```

```
typedef struct shmaccpt{
     unsigned int transprotocol;
    unsigned int nictech;
    key_t shmkey;
    int shmsize;
    key_t semakey;
    unsigned int sem_num;
    ...
};
```

```
typedef struct udpaccpt{
/*protocolo de transporte utilizado*/
    unsigned int transprotocol;
/*Tecnologia da interface de rede*/
    unsigned int nictech;
/* nome da interface*/
    char interface_name[6];
/* Endereço IP ligado a Interface de rede */
    char address[14];
/*Porta*/
   unsigned int port;    …
};
```

Figura 17: Definições das estruturas accesspt, udpaccpt e shmaccpt

A função `vcConfig()` tem por objetivo fazer ajustes no canal. A estrutura *config* pode ser composta por vários campos para ajustes finos ao canal. Entretanto, apenas um campo faz parte desta estrutura no momento:

- **int fail_police:** este campo recebe o tipo de política adotada no caso de falha de comunicação. Dois tipos de política são atualmente adotados:
  - *TRY_ALL_ROUTES*: redireciona o fluxo de dados para todas as rotas possíveis, ou seja, tenta enviar dos dados pelas rotas de todas as interfaces de redes origem para todas as interfaces destino.
  - *TRY_SPARE_NIC_FIRST*: redireciona o fluxo de dados diretamente para a interface de rede secundária, que envia as mensagens para o ponto de acesso ligado a interface de rede de mesma tecnologia que a origem.

A próxima função `vcWriteto()` têm a finalidade de transmitir os dados do canal Produtor para o canal Consumidor. Esta função é composta pelos seguintes parâmetros:

- **idaddr_t my_id:** Identificador do Produtor a ser utilizado na transmissão. Como vários produtores podem estar transmitindo em uma mesma aplicação (através de múltiplas *threads*) o id do canal produtor é utilizado para distinguir por qual produtor a mensagem será enviada.
- **char *data:** Ponteiro para os dados a serem enviados.
- **unsigned datalen:** Tamanho (em *bytes*) dos dados a serem enviados.

Já a função vcReadfrom() recebe os dados enviados pela ponta produtora da comunicação. Esta função é composta pelos seguintes parâmetros:

- **idaddr_t my_id:** Identificador do Consumidor a ser utilizado na recepção de mensagens. Como vários consumidores podem estar recebendo dados em uma mesma aplicação (através de múltiplas *threads*) o id do canal consumidor é utilizado para distinguir por qual consumidor a mensagem deve ser recebida.
- **char **data:** Armazena o dado recebido do Canal Produtor.
- **unsigned *datalen:** Retorna o tamanho do dado enviado pelo Canal Produtor

Para melhor compreensão do uso das primitivas descritas acima, o fluxograma da Figura 18 ilustra a ordem com que estas devem ser executadas para cada id de canal no uso do Canal Virtual de Tempo-Real.

Primeiramente, a primitiva `VirtualChannel()` é invocada passando o valor *START* no parâmetro *opt* e dos demais parâmetros descritos anteriormente para fazer a inicialização

do canal, indicando os pontos de acesso reais do canal, o tipo que o canal assumirá, o nome do canal e número identificador do canal. Em seguida, as funções `vcWriteto()` e `vcReadfrom()` são invocadas para fazer respectivamente a emissão ou recebimento das mensagens, dependendo do tipo de canal inicializado. Finalmente, a primitiva `VirtualChannel()` é invocada com o valor *STOP* no parâmetro *opt* para destruir o canal.

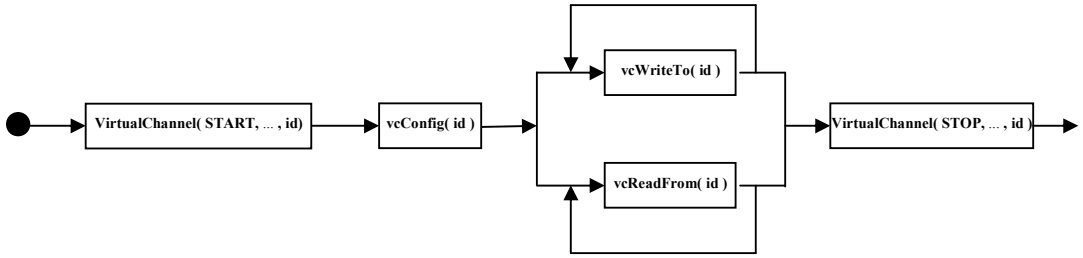

Figura 18: Ordem de execução das primitivas do Canal Virtual de Tempo-Real

4.3.2 Módulo Criador/Destruidor de Canal

Antes de efetuar a comunicação de dados, as estruturas internas do canal devem ser criadas e inicializadas, assim como destruídas após o termino do uso do Canal Virtual de Tempo-Real. Dessa forma, o propósito deste módulo é, basicamente, criar e destruir as estruturas internas do Canal. A diante, veremos algumas das mais importantes estruturas criadas pelo Criador/Iniciador de Canal:

- **Id Local:** Recebe um número que identifica o canal localmente. Este número deve ser único no sistema local.
- **Endereço de Máscara Local:** Número identificador da máquina local. Este endereço virtual diferencia as máquinas umas das outras.
- **Pontos de Acesso do canal:** Esta estrutura é formada por dois vetores de duas posições cada, onde o primeiro vetor contém informações de protocolos especialistas em comunicação local e o segundo para comunicação remota. A primeira posição de cada vetor contém o ponto de acesso primário, que será preferencialmente utilizado na comunicação. Já na segunda posição, conterá o ponto de acesso secundário, que será utilizado no caso de haver falha na comunicação.
- **Id da Mensagem:** Identificador único atribuído a cada mensagem.

- **Número Seqüencial da última mensagem:** Cada mensagem recebe um número seqüencial único, para evitar duplicidade de informação.

Na inicialização das estruturas internas do canal, primeiramente, o *Broker* Interno é disparado e a máscara de máquina requisitada para formar o identificador único do canal. O próximo passo dado pelo módulo varia conforme o tipo de canal a ser inicializado. Caso o canal for do tipo Consumidor, uma requisição de cadastro de nome é solicitada, e, dessa forma, seu nome, identificador e pontos de acesso reais são devidamente cadastrados junto aos *Brokers* Local e Global. Todavia, se este for Produtor, a técnica de *cache* e *prefetching* é utilizada, isto é, a requisição e armazenamento do identificador de máquina e da tradução do nome do canal Consumidor par são executados anteriormente à comunicação de fato. Isto contribui com a diminuição dos atrasos no início da comunicação entre os pares, devido ao fato de não ocorrer consultas aos *Brokers* por tradução de pontos de acesso do canal par durante a comunicação. A Figura 20 ilustra o comportamento do módulo Criador/Destruidor de Canal.

4.3.3 Repositório de Configuração de Canal

Este módulo trata do armazenamento e recuperação das estruturas internas criadas, inicializadas e modificadas pelos módulos Criador/Destruidor de Canal, Produtor e Consumidor. Cada vez que um novo canal produtor/consumidor é criado pela aplicação usuário, uma nova seção é aberta e inserida em uma lista, sendo cada qual identificada pelo endereço de canal. Assim, podemos ter vários canais abertos em uma mesma aplicação, cada qual com sua configuração.

```
#include "api/virtualchannelapi.h"
int main(void)
{
    char prod_name[6];
    idaddr_t  my_id = 1111;
    struct accesspts my_apts;
    udpaccpt *udpap[NUMBER_OF_REMOTE_ACCESSPOINTS];
    shmaccpt *shmap[NUMBER_OF_LOCAL_ACCESSPOINTS];
    memcpy(&prod_name, "boss", strlen ("boss"));

    /* Inicializa os pontos de acesso para comunicação remota*/
    for (int count = 0; count < NUMBER_OF_LOCAL_ACCESSPOINTS; count++)
        shmap[count] = (shmaccpt *)calloc(1,sizeof(shmaccpt));

    shmap[0]->conn_index = ConnIndex_Gen();
    shmap[0]->transprotocol = SHMEM;
    shmap[0]->shmkey  = 223;
    shmap[0]->shmsize = DEFAULT_SHM_MEMORY_SIZE;
    shmap[0]->semakey = 2;
    shmap[0]->sem_num = 0;
    /* Inicializa os pontos de acesso para comunicação remota*/
    for (int count = 0; count < NUMBER_OF_REMOTE_ACCESSPOINTS;count++)
       udpap[count] = (udpaccpt *)calloc(1, sizeof(udpaccpt));

    udpap[0]->conn_index = ConnIndex_Gen();
    memcpy(&udpap[0]->interface_name, "eth0", sizeof("eth0") );
    udpap[0]->transprotocol = UDP_BIT_ALT;
    udpap[0]->nictech       = FAST_ETHERNET;
    memcpy(udpap[0]->address,"200.18.98.171",14);
    udpap[0]->port             = 8001;

    my_apts.localaccesspt[0] = (void *)shmap[0];
    my_apts.localaccesspt[1] = (void *)shmap[1];
    my_apts.rmtaccesspt[0]   = (void *)udpap[0];
    my_apts.rmtaccesspt[1]   = (void *)udpap[1];
    ...
    VirtualChannel(PRODUCER, START, producer_name, my_id, my_apts);
    ...
    VirtualChannel(PRODUCER, STOP, producer_name, my_id, my_apts);
    ...
}
```

Figura 19: Exemplo de inicialização e destruição de um Canal Virtual de Tempo-Real do tipo produtor pela aplicação do usuário

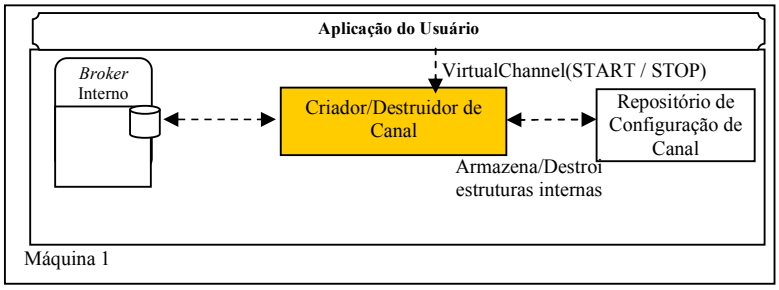

Figura 20: Módulo Criador/Destruidor de Canal

4.3.4 Módulo Consumidor

Este módulo é responsável por disparar várias tarefas receptoras, cada uma responsável pelo recebimento dos dados nos pontos de acesso cadastrados pela aplicação do usuário. O módulo Consumidor também é faz o desenfileiramento das mensagens recebidas pelas tarefas receptoras e descarte de mensagens duplicadas. A Figura 21 ilustra o diagrama de seqüência do módulo Consumidor, enquanto a Figura 22 o funcionamento deste módulo.

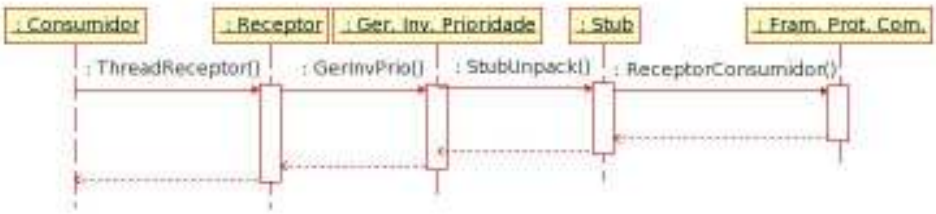

Figura 21: Diagrama de seqüência do módulo Consumidor

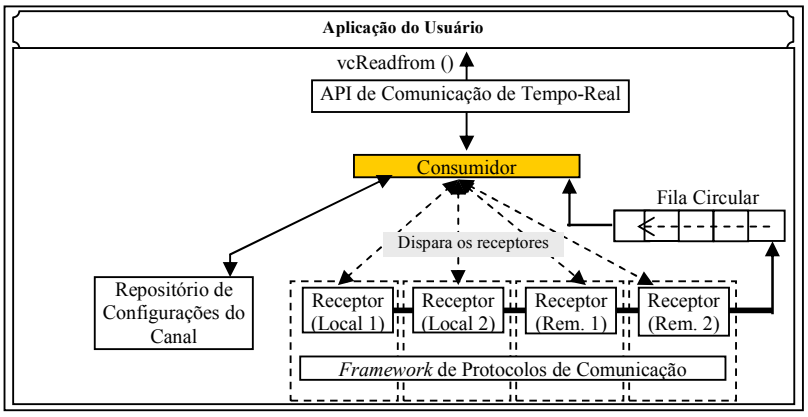

Figura 22: Funcionamento do módulo Consumidor

4.3.5 Módulo Produtor

O módulo Produtor gera e atualiza o campo Identificador de Mensagem conforme novas mensagens são produzidas pela aplicação usuário. As Figuras 23 e 24 ilustram respectivamente o funcionamento do módulo Produtor e o diagrama de seqüência do módulo Produtor no envio de dados.

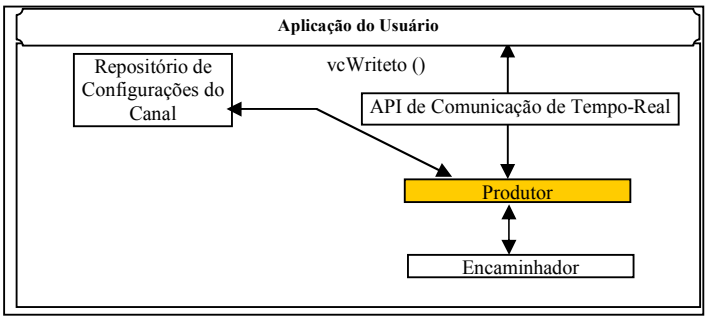

Figura 23: Funcionamento do módulo Produtor

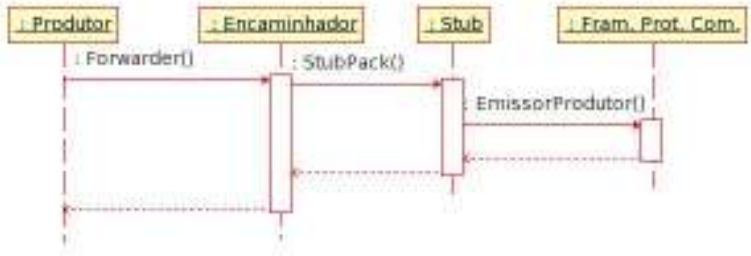

Figura 24: Diagrama de seqüência do módulo Produtor

4.3.6 Módulo Encaminhador

Este módulo é responsável pela numeração seqüencial dos pacotes e seleção apropriada dos protocolos e interfaces de comunicação para o encaminhamento das mensagens, tanto na transmissão normal dos dados quanto na presença de falhas na transmissão.

Com relação ao encaminhamento de mensagens, este é baseado na localização entre os comunicantes. Com o endereço virtual do canal Consumidor, a máscara é extraída e comparada com a máscara local. Dessa forma, caso as máscaras sejam iguais, o ponto de acesso para comunicação local e o endereço virtual do Canal Consumidor são recuperados. Caso contrário, o ponto de acesso para comunicação remota é selecionado.

As Figuras 25 e 26 ilustram respectivamente a comunicação entre processos em um mesmo sistema computacional e entre processos em sistemas computacionais distintos. Na Figura 27 é ilustrado o código em C para verificação de localização do canal Consumidor par pelo canal Produtor.

Na presença de falha durante a comunicação, o módulo Encaminhador entra em ação, utilizando a política de falha selecionada na configuração da inicialização do Canal (*TRY_ALL_ROUTES* / *TRY_SEC_ROUTE_FIRST*).

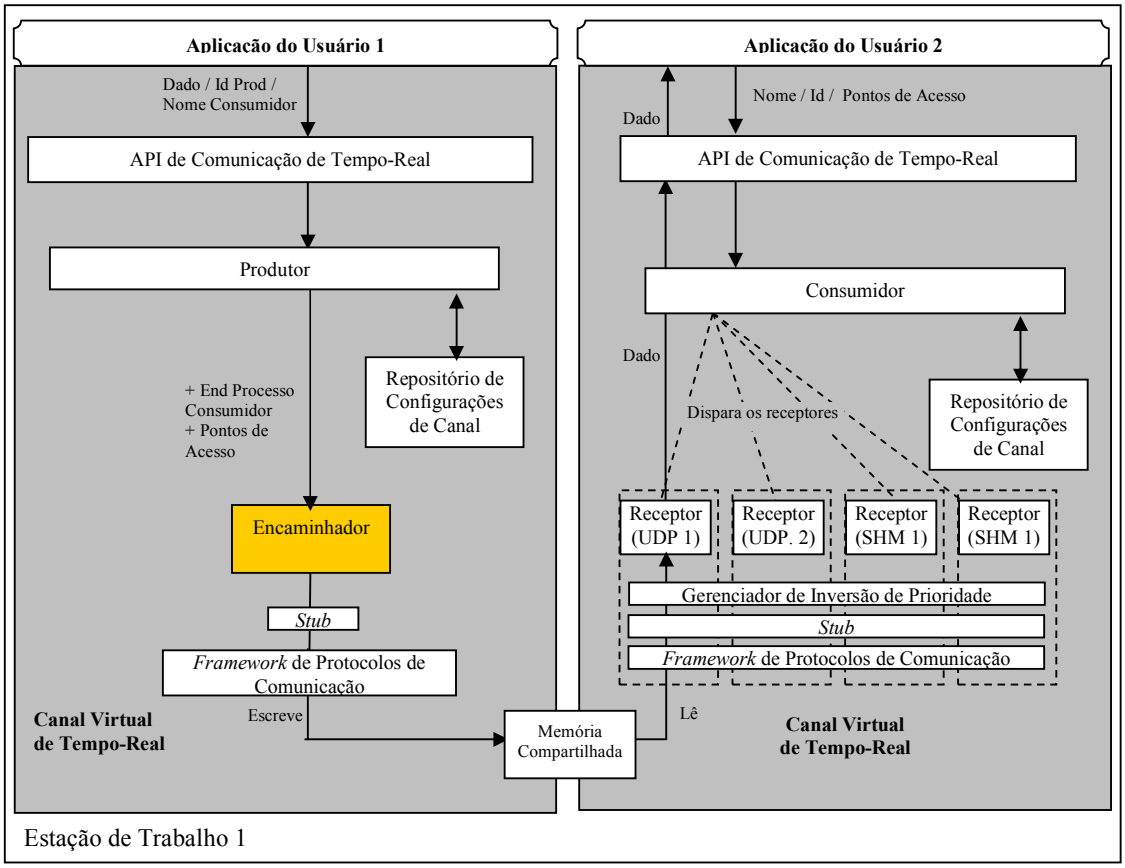

Figura 25: Módulo Encaminhador - Cópia local de dados

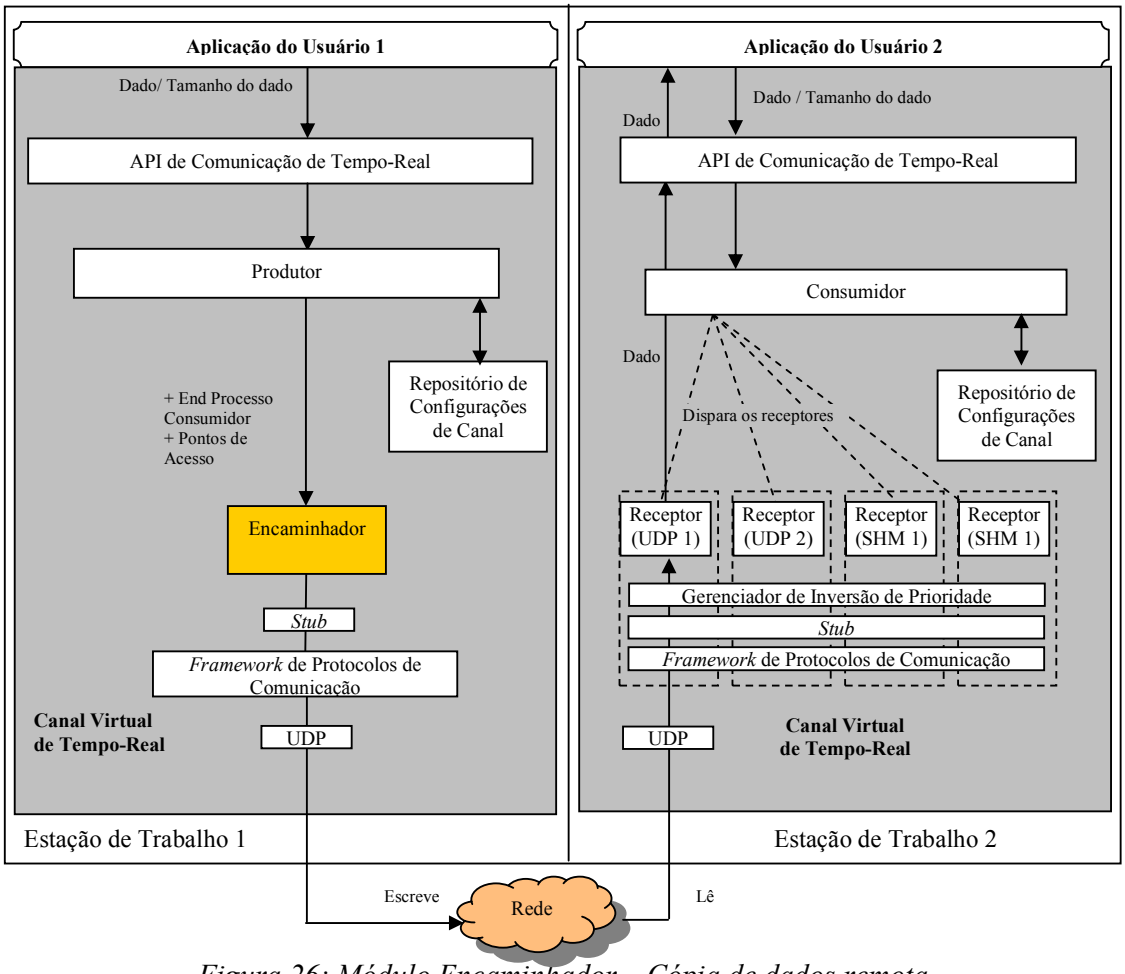

Figura 26: Módulo Encaminhador – Cópia de dados remota

```
#define MULT_FACTOR 1000000

...

idaddr_t peer_mask = cons_peer->idcons / MULT_FACTOR;

/*Exemplo: ⌊666000123/1000000⌋ = 666 */

if(peer_mask == **net_mask_addr) {
       //Canais pertencem a mesma máquina
       ...
}else{
       //Canais pertencem a máquinas diferentes
       ...
}
```

Figura 27: Código em C para verificação de localidade entre os canais Produtor e Consumidor

Para nomear as tecnologias das interfaces de rede, o Canal Virtual de Tempo-Real designa um número decimal único para cada tipo de tecnologia, onde a parte inteira identifica

as tecnologias compatíveis entre si, e a parte decimal a versão da tecnologia, como ilustra a Figura 28.

```
/*************************************************/
                /* NIC TECHNOLOGIES*/
/*************************************************/
#define MEMORY              1.0
#define BLUETOOTH_V1_1  2.0
#define BLUETOOTH_V2_0  2.1
#define WI_FI               3.0
#define FAST_ETHERNET   4.0
#define GIGA_ETHERNET   4.1
/*************************************************/
```

Figura 28: Identificação numérica das tecnologias de interface de rede

4.3.7 Módulo Receptor

O módulo Receptor desempenha a função de enfileirar os pacotes recebidos para serem consumidos pelo módulo Consumidor. Neste módulo é disparada uma *thread* receptora, que é associada a uma interface de rede e a um protocolo de comunicação em particular, que faz a redundância de pontos de acesso. O exemplo na Figura 29 ilustra quatro *threads* receptoras do módulo Receptor, cada qual utilizando um protocolo de comunicação em particular e fazendo o enfileiramento (*bufferização*) das mensagens recebidas.

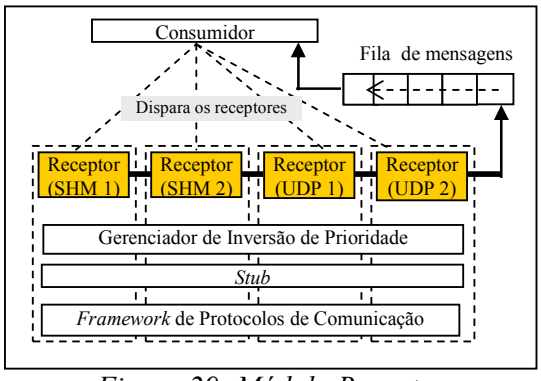

Figura 29: Módulo Receptor

4.3.8 Módulo *Stub*

A função do módulo *Stub* é adicionar e remover o cabeçalho de comunicação *Stub* do Canal Virtual de Tempo-Real em cada mensagem a ser transmitida. Para fazer o serviço, este

módulo oferece uma função para inserção e outra pra retirada do cabeçalho *Stub*, chamadas respectivamente de *Stubpack()* e *Stubunpack().* Assim sendo, toda vez que uma mensagem for enviada ou recebida por um canal Produtor ou Consumidor, esta é processada pelo *Stubpack()* e *StubUnpack()* para adicionar e remover respectivamente o cabeçalho *Stub*. Os campos que formam o cabeçalho *Stub* são os seguintes:

- Política: Identifica qual a política de escalonamento de tarefa é utilizada pelo processo comunicante de origem. Este campo é utilizado no tratamento de inversão de prioridade pelo canal Produtor.
- Prioridade: Identifica qual a prioridade utilizada pelo processo comunicante de origem. Da mesma forma que o campo Política, este é utilizado no tratamento de inversão de prioridade pelo canal Produtor.
- ID do canal Origem: Identifica o endereço virtual do canal origem da mensagem.
- ID da Mensagem: Identifica a qual mensagem o pacote pertence.
- Número de Seqüência do Pacote: O *Stub* Produtor gera um número seqüencial para cada pacote.
- Política do Canal Produtor: Indica a política de escalonamento utilizada pelo canal Produtor par. Este campo deve ser adicionado para auxiliar no tratamento da inversão de prioridade quando o pacote chegar ao destinatário.
- Prioridade do Produtor: Cada Mensagem recebe a prioridade da aplicação do usuário. Este campo deve ser adicionado para auxiliar no tratamento da inversão de prioridade quando o pacote chegar ao destinatário.

A Figura 30 ilustra o cabeçalho *Stub* e seu conjunto de campos em um pacote dos Canais Virtuais.

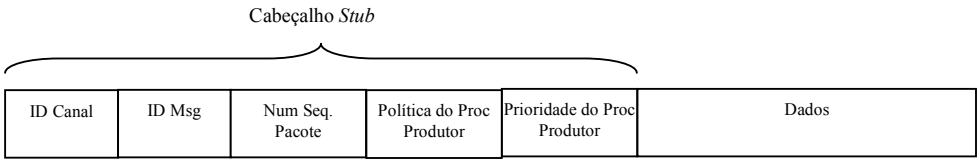

Figura 30: Pacote de dados do Canal Virtual

4.3.9 Gerenciador de Inversão de Prioridade

Para evitar a Inversão de prioridade, foi implementado o algoritmo *Immediate Priority Ceiling* [28]. Portanto, quando a mensagem é desempacotada, o Gerenciador de Inversão de Prioridade verifica a prioridade do canal Consumidor e a equipara ao nível da prioridade do

canal Produtor, caso seja menor do que a do seu par comunicante. Depois de receber toda a mensagem, o Gerenciador de Inversão de Prioridade restabelece a prioridade original do Canal Consumidor.

A Figura 31 mostra o trecho de código em linguagem C que eleva a prioridade do canal Consumidor no caso da prioridade do seu par ser maior.

```
if(polpri.policy == SCHED_RR &&
   polpri.priority > polpri_current.priority)
{
   SetSched(getppid()  , polpri.policy, polpri.priority);
   SetSched(Getpid()   , polpri.policy, polpri.priority);
}
```

Figura 31: Exemplo do código fonte da elevação de prioridade pelo Gerenciador de Inversão de Prioridade

4.3.9.1 *Framework* de Protocolos de Comunicação

Este módulo disponibiliza um conjunto de funções necessárias para integrar os mais variados protocolos de comunicação com o Canal Virtual de Tempo-Real. Dessa forma, cada protocolo apresenta a sua versão para a implementação de cada função, seguindo apenas algumas guias impostas pelos requisitos funcionais que a função do *framework* deve desempenhar.

Primeiramente, para integrar o novo protocolo de comunicação ao *framework*, uma nova estrutura de dados deve ser criada, contendo obrigatoriamente os campos *unsigned int transprotocol* e *unsigned int nictech*, que receberão respectivamente o código do novo protocolo de transporte, e o tipo de tecnologia a qual um ponto de acesso desta tecnologia será associado. Mais variáveis deverão ser inseridas nesta nova estrutura, que farão a identificação do ponto de acesso, como no caso dos campos IP e porta do protocolo de comunicação UDP, e a identificação do ponto de acesso depois que este tenha sido criado no sistema. Um exemplo deste campo é o descritor de *socket* do protocolo UDP. Um descritor de *socket* é um número único que representa o ponto de acesso criado no sistema operacional. Outras variáveis poderão ser inseridas a vontade nesta nova estrutura, de forma que a aplicação usuário possa fazer ajustes no novo protocolo, como tempo de *timeout* para cada pacote, número de retransmissões, entre outras.

O próximo passo para integração do novo protocolo ao *middleware* é a implementação de quatro funções do *framework*: *EmissorProdutor(), ReceptorConsumidor(), ProtocolStubPackBroker()* e *ProtocolStubUnpackBroker()*.

A função *EmissorProdutor()* tem como requisitos funcionais básicos inicializar o ponto de acesso do protocolo de comunicação (no caso deste não estar inicializado), enviar os dados para o ponto de acesso par e por fim destruir o ponto de acesso ao término do uso do Canal.

A segunda função, *ReceptorConsumidor()*, tem como funcionalidades básicas inicializar o ponto de acesso do protocolo de comunicação selecionado, receber os dados transmitidos pela outra ponta comunicante, e destruir o ponto de acesso na finalização do Canal.

Na função *ProtocolStubPackBroker()* os campos relevantes do ponto de acesso do protocolo a ser cadastrado e traduzido devem ser inseridos na posição reservada aos campos variáveis do cabeçalho *Stub Broker*. Uma abordagem mais detalhada sobre os campos variáveis e suas posições será discutida na subseção *Broker*.

Por último, na função *ProtocolStubUnpackBroker()*, os campos adicionados à mensagem do cabeçalho *Stub Broker* deverão ser extraídos na ordem em que foram inseridos.

4.4 *Broker*

O papel principal do *Broker* é traduzir o nome do canal Produtor em endereço de canal Produtor e seus respectivos pontos de acesso reais (*sockets*, memórias compartilhadas, entre outros). Além do papel de tradutor, o *Broker* oferece um serviço para aquisição de máscara de máquina, isto é, fornece um número único para cada máquina na rede.

4.4.1 Níveis hierárquicos de Broker

Um ponto importante a ressaltar sobre o *Broker* são os três níveis de hierarquia em que estes podem pertencer: *Broker* Interno, *Broker* Local e *Broker* Global.

O primeiro nível é o *Broker* Interno. Este atua como uma *thread* da aplicação usuário, recebendo diretamente as solicitações de serviços do Canal Virtual de Tempo-Real utilizado pela aplicação e armazenando os dados recebidos de resposta dos *Brokers* de maior hierarquia, fazendo o papel de um *cache* local à aplicação. O uso deste *Broker* Interno a

aplicação se justifica devido ao fato da comunicação entre *threads* ser mais rápida que entre processos distintos, conforme [29]. Assim, as informações armazenadas em *cache* no *Broker* Interno podem ser acessadas e recuperadas mais rapidamente pelos canais que executam em uma mesma aplicação.

O segundo nível é representado pelo *Broker* Local. Este *Broker* é executado como um processo em separado, atendendo todas as solicitações encaminhadas pelos *Brokers* Internos no sistema computacional local e armazenando as respostas das requisições para futuras consultas locais.

Já o terceiro nível é composto pelo *Broker* Global que têm um escopo de atendimento global ao sistema, atendendo a todas as requisições de todos os *Brokers* Locais, sendo a última instância para solicitação de serviços de *Broker*.

Assim que as solicitações vão sendo atendidas e as informações recuperadas, os *Brokers* armazenam os resultados em suas tabelas e os enviam para os níveis hierárquicos inferiores até que sejam copiados para as estruturas internas do Canal Virtual da aplicação usuário. Este armazenamento intermediário das respostas pelos vários níveis de *Broker* é baseado no princípio de localidade e referência, utilizado na arquitetura de memória dos sistemas computacionais modernos [30] para acelerar o processo de recuperação de dados nos casos em que se necessite novamente da tradução de endereço do canal Consumidor para uma futura transmissão, devido a velocidade na recuperação dos dados.

A Figura 32 ilustra os níveis hierárquicos de *Broker* do sistema, levando em consideração a velocidade de acesso de seus serviços e a sua visibilidade de atendimento para com os canais do sistema.

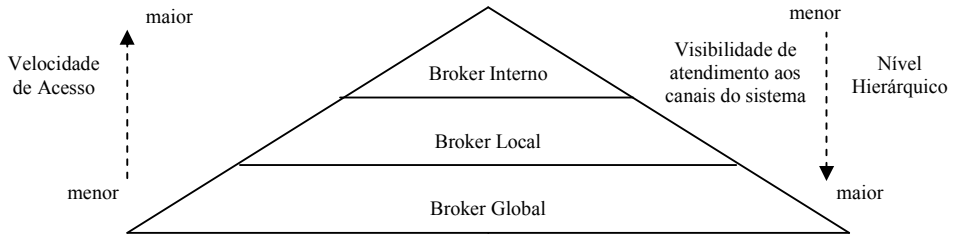

Figura 32: Velocidade de Acesso a um Broker por um Canal Produtor/Consumidor

4.4.2 Arquitetura dos *Brokers*

Os *Brokers* se utilizam os mesmos módulos da arquitetura do Canal Virtual de Tempo-Real com o acréscimo de três outros módulos: O módulo Montador de Mensagens *Broker*, *Framework* de Serviços de *Broker*, módulo *Stub Broker*. Na Figura 33 abaixo, a arquitetura de *Broker* é ilustrada juntamente com os canais Produtor e Consumidor. Estes são utilizados como os mecanismos de envio e recebimento na troca de mensagens entre os *Brokers*.

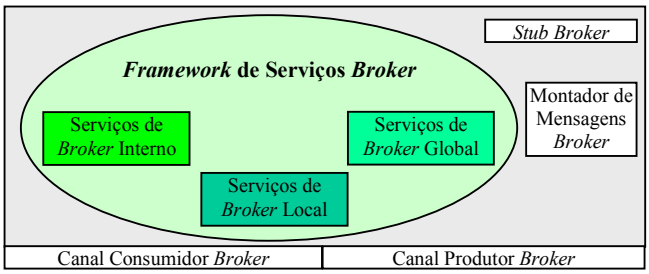

Figura 33: Arquitetura do Broker dos Canais Virtuais de Tempo-Real

4.4.2.1 Montador de Mensagens *Broker*

Este módulo é responsável por montar mensagens para solicitar diversos serviços oferecidos pelo *Framework* de Serviços *Broker*, assim como as mensagens de resposta dos serviços solicitados. As funcionalidades oferecidas por este módulo são *RequestMask(), RegisterConsumer(), TranslateConsumerName()*, *UnregisterConsumer()*, *Peerfound()*, *QueuedforTranslate()* e *RespMask()*. Estas funções são responsáveis respectivamente pelas montagens de mensagens para:

- Requisição de máscara de máquina.
- Requisição de registro de canal Produtor.
- Requisição de tradução do nome de consumidor.
- Requisição de remoção de registro de consumidor.
- Resposta com os dados da tradução do canal Produtor.

- Aviso de enfileiramento do canal Produtor na fila por tradução de um canal Produtor.
- Resposta contendo a máscara da máquina.

4.4.2.2 Canal Consumidor

O objetivo do canal Consumidor no *Broker* é permanecer a espera por requisições de serviços das aplicações de usuário. Apesar do funcionamento do canal ser feito conforme o descrito na seção Canal Virtual de Tempo-Real, a sua inicialização é ligeiramente diferente da inicialização utilizada nos canais Consumidor, não havendo o cadastro junto aos *Brokers* do canal Consumidor. Isto acontece pelo fato que todos que os *Brokers* são pré-configurados com os pontos de acesso reais dos canais Consumidores dos *Brokers*.

4.4.2.3 *Stub Broker*

Este módulo oferece os serviços de empacotamento e desempacotamentos das mensagens de requisição de serviço. Para estas mensagens, dois campos do cabeçalho são padrões para todos os tipos de mensagens. Entretanto, os demais campos são determinados conforme o valor do primeiro campo (*opcode*) da mensagem. Os campos padrões são:

- Opcode: Código da operação a ser efetuada. Cada mensagem tem um código único de identificação. Dependendo do valor deste campo, os demais campos não fixos do cabeçalho são definidos. Os valores que este campo pode assumir são estes:
  - REQ_REGISTER: Mensagem de registro de consumidores.
  - REQ_TRANSLATE: Mensagem de tradução de nome de consumidores.
  - REQ_UNREGISTER: Mensagem de remoção do registro de consumidores.
  - RESP_PEERFOUND: Usado na resposta por um *Broker* caso o endereço do canal Produtor par tenha sido encontrado.
  - RESP_QUEUED_FOR_TRANSLATE: Caso o nome do consumidor não tenha sido ainda registrado, uma mensagem indicando que o canal Produtor foi

enfileirado para uma futura tradução. Este valor de *opcode* que não necessita de campos extras.

- REQ_ADDR_MASK: Faz uma requisição de máscara de Máquina. Este valor de *opcode* que não necessita de campos extras.
- RESP_ADDR_MASK: Reposta à requisição de máscara.

- IdFrom: Identifica o endereço virtual do remetente da mensagem.

Como dito anteriormente, dependendo do valor assumido no campo *opcode*, os demais campos não fixos do cabeçalho são escolhidos. A seguir, serão listados os campos extras inseridos para cada valor de *opcode*:

a) REQ_REGISTER:

- cons_name: Apelido do canal Produtor a ser registrado.
- cons_id: endereço virtual do canal Produtor.
- localaccesspt[]: vetor de pontos de acesso para comunicação local. Este campo é subdividido em vários campos, onde cada campo varia conforme o protocolo de comunicação escolhido. Esta subdivisão é composta basicamente em três campos. Um campo indicando o protocolo de comunicação utilizado, outro indicando a tecnologia da interface de rede na qual o ponto de acesso estará associado (*binded*) e o terceiro sendo composto por múltiplos campos. Estes descrevem especificamente as características que identificam o ponto de acesso em questão (um campo para o endereço IP e outro para a porta, tomando como exemplo o protocolo UDP). Os dois primeiros campos devem constar em todos os pontos de acessos que forem acrescentados ao *framework* de protocolos de comunicação.
- rmtaccesspt[]: Idêntico ao item anterior. Contudo, este vetor identifica os pontos de acesso utilizados para comunicação remota.

b) REQ_PEERFOUND: São inseridos os mesmos campos de REQ_REGISTER. A diferença é que esta resposta é direcionada ao canal Produtor que fez a requisição de tradução de nome.

c) REQ_TRANSLATE:

- cons_name: Representa o apelido do canal Consumidor a ser traduzido.

d) REQ_UNREGISTER:

- cons_name: Apelido do canal Consumidor a ser removido da base de dados dos *Brokers*.

e) RESP_ADDR_MASK:

- net_mask_addr: identificador único da máquina.

As Figuras 34 e 35 mostram respectivamente os campos do cabeçalho *Stub Broker* para o valor de *opcode* REGISTER e os campos fixos e dependentes de *opcode* do cabeçalho *Stub Broker*.

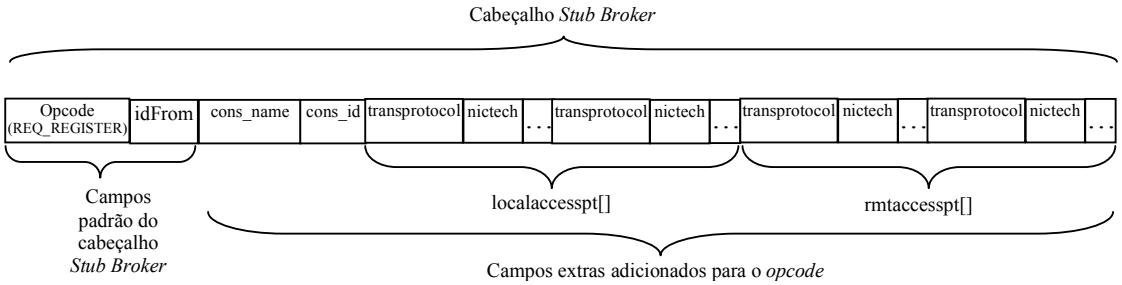

Figura 34: Campos do cabeçalho Stub Broker para o valor de opcode REQ_REGISTER

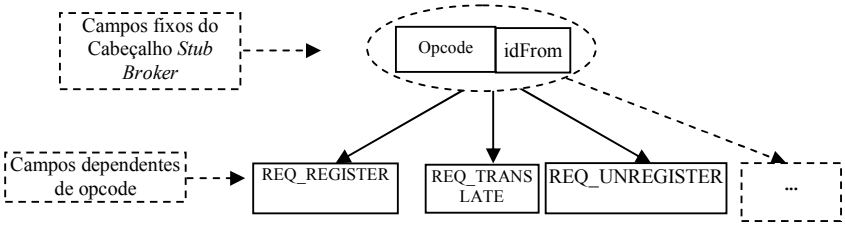

Figura 35: Campos fixos e dependentes de opcode do cabeçalho Stub Broker

4.4.2.4 *Framework* de Serviços de *Broker*

Este *Framework* oferece um conjunto de serviços para atendimento a requisições ao *Broker*. Cada serviço é identificado por um número único, chamado *optcode*. Estes conjuntos de serviços são implementados de forma semelhante para cada um dos três *Brokers*, sendo estes os serviços de *Broker* Interno, serviços de *Broker* Local e serviços de *Broker* Global.

Um exemplo dessa variação ocorre no serviço de requisição de máscara de máquina. Apesar de este serviço ser atendido por todos os *Brokers*, a máscara de máquina é gerada

somente pelo *Broker* Global. Isto acontece pelo fato do *Broker* Global ter uma visão geral do sistema, e assim poder centralizar a geração de números identificadores para cada máquina, garantindo dessa forma a unicidade de cada identificador. A seguir, serão descritos os serviços de *Broker* Interno, Local e Global:

- Registro de Consumidor: Recebida a requisição de registro, o cadastro é feito em uma lista de nomes de consumidores, e logo em seguida enviada ao próximo nível de *Broker*. Não havendo mais níveis de *Broker*, a requisição não é mais encaminhada. No caso de existir canais Produtores na lista de espera por uma tradução, mensagens de RESP_PEERFOUND são enviadas aos seus respectivos destinatários.
- Serviço de Tradução de Nomes: Na tradução de nomes de canais Consumidores, o *Broker* Interno recebe a primeira mensagem de tradução de nomes do canal Produtor. Caso não seja possível fazer a tradução nessa primeira tentativa, o pedido é colocado em uma fila de espera ordenada pela prioridade do canal Produtor, e encaminhada para o próximo nível de *Broker,* até chegar ao terceiro nível, o *Broker* Global, que apenas insere o pedido na fila de tradução pelo canal Consumidor. Um aperfeiçoamento no encaminhamento desta requisição é feito para diminuir o acesso ao meio de comunicação. Este consiste em colocar como remetente desta mensagem somente o identificador do *Broker* Local, além de não encaminhar mensagens de requisição de tradução que já haviam sido requisitadas por outros canais consumidores do sistema para o *Broker* Global. Dessa forma, a mensagem de tradução de nomes é transmitida pelo meio apenas uma vez. Dessa forma, o *Broker* Global também responde apenas uma vez a um *Broker* Local pela tradução de nomes.
- Serviço de Endereçamento de Máquina: Neste serviço, a mensagem de requisição de máscara de máquina é enviada na inicialização do canal Produtor para os *Brokers* Interno, e Local e por fim, encaminhada ao *Broker* Global, que faz o cálculo da máscara de máquina e retorna uma mensagem RESP_ADDR_MASK contendo a máscara.
- Serviço de Remoção de Registro: Mensagem enviada pelo canal Consumidor que percorre todos os níveis de *Broker* para retirar seu cadastro da lista de consulta dos *Brokers*.

Um ponto importante sobre os serviços de *Broker* é a possibilidade de criação de novos serviços para expandir as funcionalidades dos *Brokers*. Para isso, um novo código deve ser criado para identificar o novo serviço, assim como novas funções no módulo Montador de Mensagens para criar a nova requisição de mensagem e código inserido às funções de *Stub Broker* para empacotar e desempacotar as mensagens de requisição deste novo serviço. Para diferenciar as mensagens de requisição e resposta do serviço, o apelido dos identificadores das mensagens deve ser prefixado com “REQ” os *optcodes* de requisição e “RESP” para os de resposta às requisições.

5 ESTUDO DE CASO

Neste capitulo será apresentado um estudo de caso referente à comunicação entre dois processos utilizando o *middleware* de comunicação em um ambiente de *hardware* e rede, de forma a avaliar as características do canal descritas no primeiro capítulo, acrescentando outras características secundárias relevantes. Os pontos avaliados neste estudo serão os seguintes:

- Independência de protocolos de comunicação: para testar esta característica, será mostrado a dois protocolos de comunicação implementados e inseridos ao *Framework* de Protocolos de Comunicação. Um será o protocolo utilizando memória compartilhada (com semáforos), e o outro o protocolo bit alternante [31] via UDP.
- *Overhead* inserido pelo *middleware* de tempo-real: para mostrar a viabilidade do canal em termos de acréscimo de *overhead*, foram feitas medidas de tempo de uso de CPU para analisar o tempo da aplicação utilizando o *middleware*.
- Recuperação de comunicação na presença de falhas de transmissão: esta característica será testada através da simulação de falhas de comunicação entre dois processos utilizando o *middleware.* Vários prováveis cenários serão levados em consideração, onde serão avaliadas as políticas de configuração de rotas *TRY_ALL_ROUTES* e *TRY_SPARE_NIC_FIRST*.
- Transparência de localização na comunicação: esta questão é garantida pelos *Brokers* e pelo endereçamento virtual implementado pelo *middleware*. Como este item foi largamente discutido no capítulo anterior, não será necessário fazer uma nova abordagem desta questão.

Nas próximas seções serão apresentados, respectivamente, os cenários utilizados para os testes de transmissão de mensagens entre os dois processos, teste do *Framework* de Protocolos de Comunicação, quantização do *overhead* inserido pelo *middleware* e testes de recuperação do canal na presença de falhas de Comunicação. Este último item utilizou os dois protocolos de comunicação implementados no *Framework* de Protocolos assim como o cenário de testes lógico e físico, que será descrito na próxima seção.

5.1 Cenários de teste

Para criar um cenário de testes do Canal Virtual de Tempo-Real, primeiramente, uma implementação foi feita da arquitetura descrita no capítulo anterior, utilizando a linguagem de programação C sobre o sistema operacional Linux.

A escolha do sistema operacional Linux foi feita devido a vários fatores, onde os principais são: código fonte do sistema operacional e de suas inúmeras aplicações baseadas na licença GPL [32], características de tempo-real não-crítico nativas ao sistema, extensões do S.O. para tempo-real rígido, como RT-Linux [33] e RTAI [34], ferramental de programação e manuais de utilização inclusos nas distribuições Linux (Slackware [35] e Mandrake [36]), distribuição Linux voltadas para vários dispositivos embutidos (muLinux[37]), e uma grande portabilidade por suportar a maioria das funcionalidades definidas pelos padrões POSIX.1a (serviços essenciais), POSIX.1b (tempo-real) e POSIX.1c (*threads* POSIX) [38]. Com relação à linguagem C, sua escolha ocorreu devido a sua portabilidade, eficiência, e suporte nativo pelas distribuições do S.O. Linux.

Utilizando a implementação do *middleware*, outras duas aplicações foram implementadas, uma produtora e consumidora, e executadas em estações de trabalho distintas.

O cenário pretendido foi baseado em uma rede fictícia, onde haveria dois sistemas computacionais móveis (“A” e “B”) e um fixo (“C”), cada qual com uma interface de rede *Wi-Fi* e uma interface de rede *Bluetooth*, interfaces estas apelidadas de a1/a2 para a estação “A”, b1/b2 para B e c1/c2 para “C”. Neste cenário, haveria também um *Access Point Wi-Fi* para interconexão dos sistemas.

Avaliando este contexto, três prováveis situações poderiam ocorrer na comunicação de dados. Na primeira situação, ilustrada pela Figura 36, o nó intermediário “C” disponibilizaria uma rota alternativa de “A” para “B”, fazendo uma ponte entre as interfaces de tecnologias incompatíveis no caso de falha de transmissão. Assim, seria possível recuperar a comunicação de dados pelas rotas *a1→b1*, *a1→c1→c2→b2*, *a2→c2→c1→b1* e *a2→b2*, ou seja, poderia haver até *n* x *m* possibilidades de conexão entre as interfaces de rede estações “A” e “B”, sendo *n* o número de interfaces de “A” e *m* o número de interfaces de “B”. O segundo cenário, representado pela Figura 37, é uma variação do primeiro, onde as interfaces de rede a2 e b2 estão incomunicáveis.

No terceiro cenário, não haveria um nó intermediário como ponte, existindo somente as rotas diretas *a1→b1* e *a2→b2*, como ilustra a Figura 38.

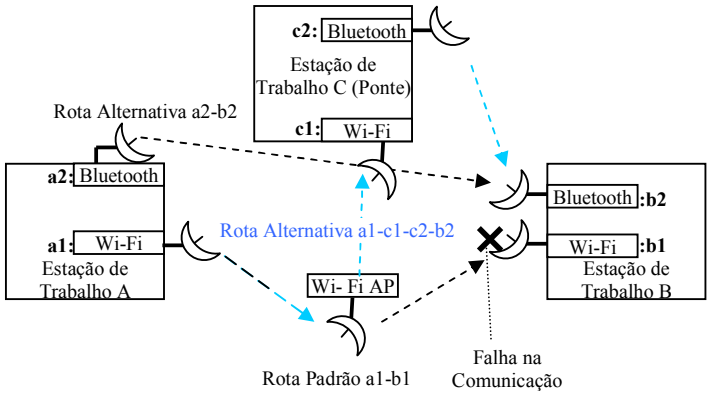

Figura 36: Cenário com comunicantes próximos composto por várias rotas alternativas de conexão

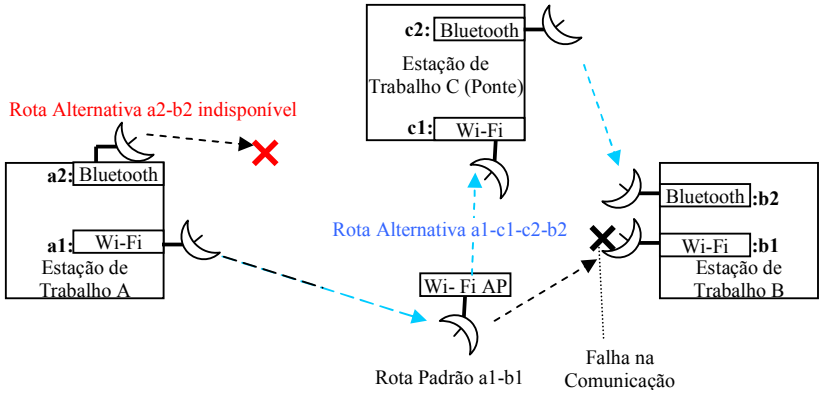

Figura 37: Variação do primeiro cenário com comunicantes fora de alcance entre as interfaces a2-b2

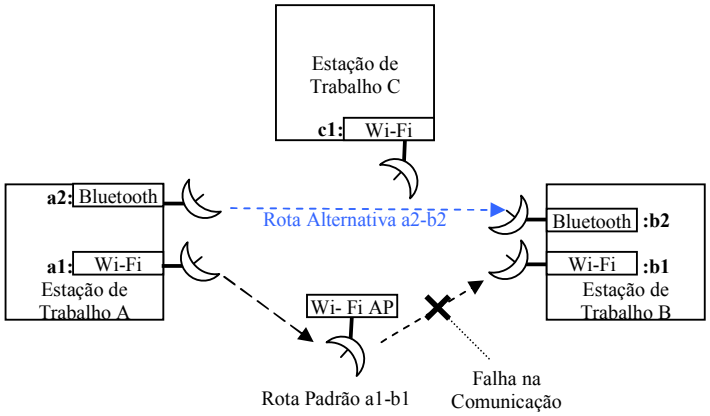

Figura 38: Cenário com apenas uma rota alternativa com comunicantes próximos

Para simular estes cenários hipotéticos de rede sem fio, foram utilizadas duas estações de trabalho em rede com as seguintes configurações:

- Estação de Trabalho A:
  - Processador: AMD Duron 900Mhz
  - Memória Principal: 128 MB RAM
  - S.O.: Linux
  - Versão de *Kernel*: 2.6.11.8
  - Interfaces de Rede: Duas interfaces *Fast Ethernet*
- Estação de Trabalho B:
  - Processador: AMD Athlon64 2.1GHz
  - Memória Principal: 512 MB RAM
  - S.O.: Linux
  - Versão de *Kernel*: 2.6.11.8
  - Interfaces de Rede: *Fast Ethernet* e *Gigabit Ethernet*

Para simular a primeira situação fictícia da rede sem fio, todas as interfaces de rede das estações de trabalho foram interligadas por um *switch*, fazendo com que as mesmas consigam ser alcançadas por rotas alternativas. Na segunda situação proposta, as quatro interfaces foram interligadas diretamente através de dois cabos *crossover*, para simular a impossibilidade de uso de rotas alternativas sob falhas na comunicação.

Tendo em vista que o propósito do nó "C" do cenário acima é apenas proporcionar um caminho alternativo entre as interfaces de rede dos sistemas "A" e "B", os caminhos de "A" para "B" que passam por "C" são considerados como equivalentes. Por isso, estes foram eliminados das simulações reais, ilustradas pelas Figuras 39, 40 e 41, que correspondem respectivamente aos cenários representados pelas Figuras 36, 37 e 38. A Tabela 1 faz a legenda das Figuras 39, 40 e 41.

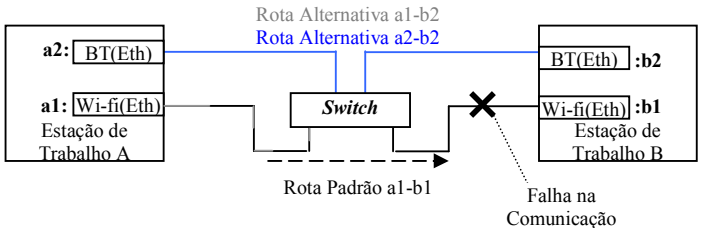

Figura 39: Cenário de simulação com várias rotas entre comunicantes próximos entre si.

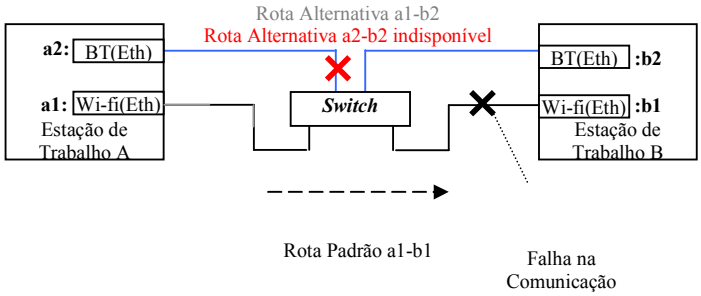

Figura 40: Cenário de simulação com várias rotas entre comunicantes distantes entre si.

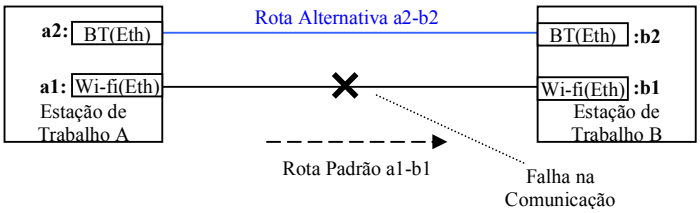

Figura 41: Cenário de simulação com apenas uma rota entre comunicantes próximos

Tabela 1: Legenda das Figuras 39, 40 e 41.

Interface de Rede	Tecnologia	Tecnologia Simulada
a1	Ethernet	Wi-Fi
a2	Ethernet	Bluetooth
b1	Ethernet	Wi-Fi
b2	Ethernet	Bluetooth

5.2 Teste do *Framework* de Protocolos de Comunicação

Para avaliar o *Framework* de Protocolos de Comunicação do Canal Virtual de Tempo-Real, foram implementados dois protocolos de comunicação interprocesso distintos. O primeiro protocolo, apelidado de Protocolo *Shmem* (proveniente da abreviação de memória compartilhada – *Shared Memory* – em inglês), faz a comunicação interprocesso via memória compartilhada utilizando semáforos para sincronização. O segundo protocolo faz a comunicação interprocesso utilizando o protocolo UDP, associado ao protocolo bit alternante [31] para efetuar retransmissão de dados no caso de mensagens na transmissão.

5.2.1 Protocolo de Comunicação via Memória Compartilhada - *Shmem*

O protocolo *Shmem* utiliza memória compartilhada com semáforos para fazer a transmissão de dados entre os comunicantes. Para haver sincronização no acesso a memória compartilhada, este protocolo utiliza uma solução do problema dos leitores e escritores [39] com suporte a múltiplos leitores simultâneos. A Figura 42 ilustra os algoritmos com uma solução do problema dos leitores e escritores com vários leitores simultâneos.

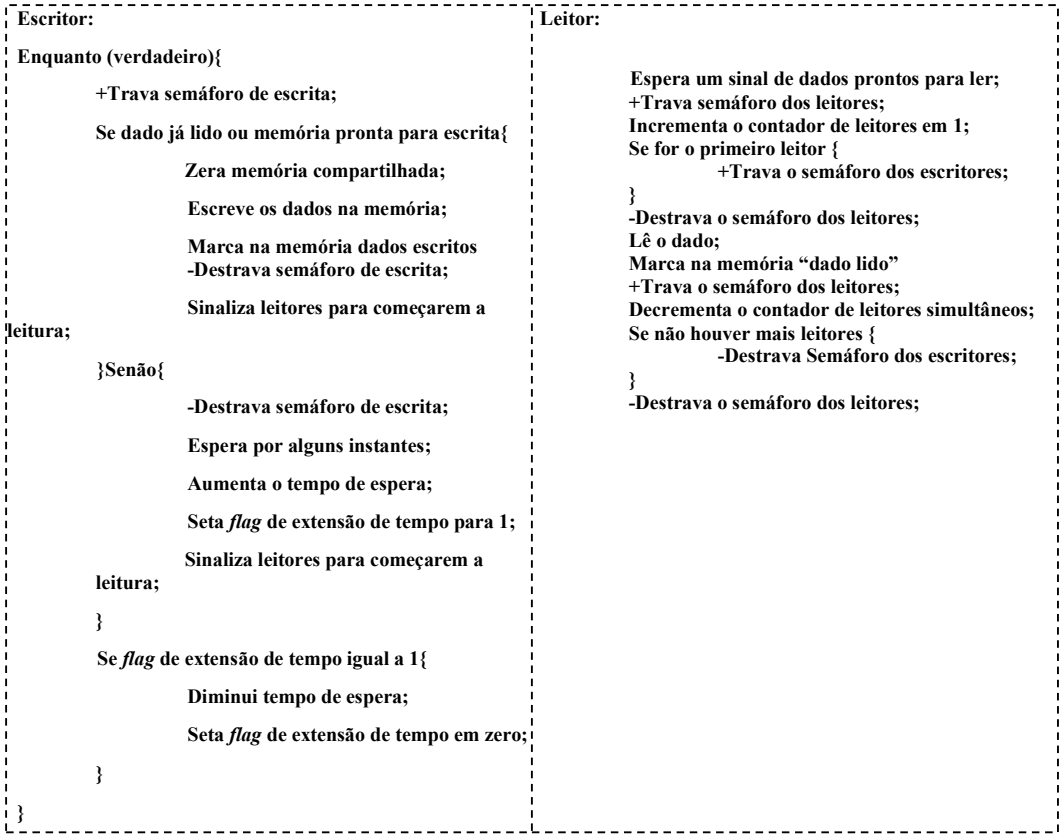

Figura 42: Algoritmo para o problema de Leitores e Escritores com suporte a múltiplos leitores simultâneos

Seguindo as diretrizes descritas na seção *Framework* de Protocolos de Comunicação, o primeiro passo feito para a integração deste protocolo foi a criação de uma nova estrutura chamada *shmaccpt* (*struct shmaccpt*) de dados, composta pelos seguintes campos:

- *unsigned int transprotocol*: Número identificador de protocolo de transporte. Para o protocolo *Shmem* foi definido o número zero.
- *unsigned int nictech:* Número identificador de tecnologia de interface de rede.

- *key_t shmkey:* Número identificador único do segmento de memória compartilhada.
- *int shmsize:* Tamanho em bytes deste segmento.
- *key_t semakey:* Número identificador do conjunto (vetor) de semáforo.
- *unsigned int sem_num:* Um semáforo do vetor de semáforos.
- *int shmid:* Número identificador de segmento de memória compartilhada inicializado.

O segundo passo foi concluído através da implementação das funções *EmissorProdutor(), ReceptorConsumidor(), ProtocolStubPackBroker()* e *ProtocolStubUnpackBroker()*, representadas respectivamente pelas Figuras 43, 44, 45 e 46.

```
int EmissorProdutor(char *pckt, int pcktsize, void **my_accesspt, void ** accesspt, idaddr_t id_to,
int closeconn, int restartconn ,struct timeout tout)
{
    ...
    switch(transprotocol)
    {
        case SHMEM:
            int shmid;
            shmap = (shmaccpt *)*accesspt;
            if (closeconn != CLOSE)
            {
                Semgrab(shmap->semakey, shmap->sem_num);
                shmid   = LocateSegment(shmap->shmkey, shmap->shmsize);
                if(shmid < 0)
                {
                    shmid = ShmCreate(shmap->shmkey, shmap->shmsize, shmap->semakey, shmap->sem_num);
                    if(shmid < 0){
                        return WRITE_ERR;
                    }
                }
            }
            else if(closeconn == CLOSE)
            {
                /*encerra o ponto de acesso.*/
                Delshm(shmid,shmap->semakey, shmap->sem_num);
                return 0;
            }
            err = Writetoshm(shmid, shmap->shmsize, shmap->semakey, shmap->sem_num,
                             pckt, pcktsize, id_to);
            if(err < 0)
                return WRITE_ERR;
        break;
        case UDP_BIT_ALT:
        ...
   }
```

Figura 43: Implementação da função EmissorProdutor() para o protocolo Shmem

```
int ReceptorConsumidor(void **accesspt, void **peer_ap, idaddr_t *my_id, stub_t **return_buffer, int
closeconn)
{
    ...
    switch(transprotocol)
    {
        case SHMEM:
            pthread_mutex_lock (&conn_mutex);
            conn = search_connlist(&headnode_connlist, conn_index);
            if (conn == NULL) {
                conn = insert_connlist(&headnode_connlist, conn_index);
                Semgrab(shmap->semakey, shmap->sem_num);
                conn->shmap_conn.shmid = LocateSegment(shmap->shmkey, shmap->shmsize);
                if ( conn->shmap_conn.shmid < 0){
                    conn->shmap_conn.shmid = ShmCreate(shmap->shmkey, shmap->shmsize,
                                                shmap->semakey, shmap->sem_num);
                    if ( conn->shmap_conn.shmid < 0 ){
                        pthread_mutex_unlock (&conn_mutex);
                        return -1;
                    }
                }
            }else if(closeconn == CLOSE) {
                Delshm(shmid,shmap->semakey,shmap->sem_num);
                pthread_mutex_unlock (&conn_mutex);
                return 0;
            }
            pthread_mutex_unlock (&conn_mutex);
            err = Readfromshm(&recvpckt, conn->shmap_conn.shmid, shmap->shmsize,
                              shmap->semakey, shmap->sem_num, my_id);
            if (err < 0)
                return READ_ERR;

        break;
        ...
```

Figura 45: Implementação da função ReceptorConsumidor() para o protocolo Shmem.

```
char *BrokerHdrPacking(brokerstub_t **hdrdata, int **hdrlenght)
{
    char brokerhdr[1024];
    int msgsize;
    unsigned lhdr;
    unsigned short shdr;
    unsigned short transprotocol;
    brokerstub_t *hdr;
    msgsize = 0;
    hdr = *hdrdata;
    ...
    switch (transprotocol)
    {
         ...
         case SHMEM:
                shmaccpt *shmap_tmp;
                 shmap_tmp = (shmaccpt *)hdr->localaccesspt[tmpcount];
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->transprotocol)),
                         sizeof(htonl(shmap_tmp->transprotocol)));
                 msgsize+=sizeof(htonl(shmap_tmp->transprotocol));
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->nictech)),
                         sizeof(htonl(shmap_tmp->nictech)));
                 msgsize+=sizeof(htonl(shmap_tmp->nictech));
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->shmkey)),
                         sizeof(htonl(shmap_tmp->shmkey)));
                 msgsize+=sizeof(htonl(shmap_tmp->shmkey));
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->shmsize)),
                         sizeof(htonl(shmap_tmp->shmsize)));
                 msgsize+=sizeof(htonl(shmap_tmp->shmsize));
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->semakey)),
                         sizeof(htonl(shmap_tmp->semakey)));
                 msgsize+=sizeof(htonl(shmap_tmp->semakey));
                 memcpy(&brokerhdr[msgsize], &(lhdr = htonl(shmap_tmp->sem_num)),
                         sizeof(htonl(shmap_tmp->sem_num)));
                 msgsize+=sizeof(htonl(shmap_tmp->sem_num));
         break;
         ...
    }
    ...
```

Figura 46: Implementação da função BrokerHdrPacking() para o protocolo Shmem.

```
brokerstub_t *BrokerHdrUnpack(char **brokerhdr)
{
    brokerstub_t *hdrdata;
    char *addrtemp, *hdrpointer;
    int i, sizeint, sizeshort, sizechar, sizeidaddr_t;
    struct in_addr address;
    int transprotocol;
    hdrpointer = *brokerhdr;
    sizeint     = sizeof(unsigned int);
    ...
    switch (transprotocol)
    {
        case SHMEM:
             shmaccpt *shmap;
             shmap = (shmaccpt *)calloc(1, sizeof(shmaccpt));
             shmap->transprotocol = ntohl(*(int      *)&hdrpointer[i]); i += sizeint;
             shmap->nictech       = ntohl(*(int      *)&hdrpointer[i]); i += sizeint;
             shmap->shmkey        = ntohl(*(key_t    *)&hdrpointer[i]); i += sizeint;
             shmap->shmsize       = ntohl(*(int      *)&hdrpointer[i]); i += sizeint;
             shmap->semakey       = ntohl(*(key_t    *)&hdrpointer[i]); i += sizeint;
             shmap->sem_num       = ntohl(*(unsigned *)&hdrpointer[i]); i += sizeint;
             hdrdata->localaccesspt[count] = (void *)shmap;
            count++;
         break;
        case ...
    }
  ...
```

Figura 47: Implementação da função BrokerHdrUnpack () para o protocolo Shmem.

5.2.2 Protocolo de Comunicação UDP com bit alternante

O protocolo UDP com bit alternante utiliza o protocolo UDP da pilha de protocolos TCP/IP juntamente com o protocolo bit alternante [31]. O protocolo bit alternante funciona exatamente como um protocolo de janela deslizante de transmissão e de recepção de tamanho um, ou seja, a cada mensagem enviada, uma confirmação de recebimento é retornada e a somente será enviada a próxima mensagem caso a anterior tenha sido confirmada. A cada mensagem enviada, um relógio é acionado para contabilizar o tempo máximo de retorno da mensagem de confirmação. Estourado este prazo de recebimento de confirmação, o dado é retransmitido. Todavia, depois de várias tentativas mal sucedidas de envio da mensagem, o protocolo retorna um código de erro.

Seguindo as diretrizes descritas na seção *Framework* de Protocolos de Comunicação, o primeiro passo feito para a integração deste protocolo foi a criação de uma nova estrutura de dados chamada *udpaccpt*, composta pelos seguintes campos:

- *unsigned int transprotocol*: Número identificador de protocolo de transporte. Para o protocolo *Udp_bit_alt* foi definido o número 1.
- *unsigned int nictech:* Número identificador de tecnologia de interface de rede.

- *char interface_name[]*: Nome associado ao dispositivo de rede (Ex.: eth0, eth1 para o sistema operacional Linux).
- *char address[]*: Endereço IP associado a interface de rede.
- *unsigned int port*: Número de Porta do *socket*.
- *int sd:* descritor de *socket*.

O segundo passo para inclusão do protocolo ao *Framework* foi concluído através da implementação das funções *EmissorProdutor(), ReceptorConsumidor(), ProtocolStubPackBroker()* e *ProtocolStubUnpackBroker()*, ilustradas respectivamente pelas Figuras 48, 49, 50 e 51.

```
int EmissorProdutor(char *pckt, int pcktsize, void **my_accesspt, void ** accesspt, idaddr_t id_to,
int closeconn, int restartconn ,struct timeout tout)
{
    ...
    switch(transprotocol)
    {
        case UDP_BIT_ALT:
            udpap       = (udpaccpt *)*accesspt;
            my_udpap = (udpaccpt *)*my_accesspt;
            conn_index = my_udpap->conn_index;
            pthread_mutex_lock (&conn_mutex);
            conn = search_connlist(&headnode_connlist, conn_index);
            if(conn != NULL)
                printf("conn->udpap_conn.sd %i\n", conn->udpap_conn.sd);
            if (conn == NULL || restartconn == RESTARTCONN){
                conn = insert_connlist(&headnode_connlist, conn_index);
                conn->udpap_conn.sd = Init_udp_Cli(&conn->udpap_conn.sock_addr,
                                        udpap->transprotocol, udpap->port,
                                        udpap->address, my_udpap->interface_name);
                if(conn->udpap_conn.sd == -1){
                    pthread_mutex_unlock (&conn_mutex);
                    return -1;
                }else
                    printf("Protocolo UDP iniciado com sucesso!!\n");
            }else if(closeconn == CLOSE){
                remove_connlist(&headnode_connlist, conn_index);
                pthread_mutex_unlock (&conn_mutex);
                return 0;
            }
            sd = conn->udpap_conn.sd;
            server = conn->udpap_conn.sock_addr;
            pthread_mutex_unlock (&conn_mutex);
            if ((err = SendToCli(sd, &server, pckt, pcktsize, 0, tout)) < 0) {
                return -1;
            }else{
                printf("Msg enviada com sucesso!\n");
            }
        break;
        ...
```

Figura 48:Implementação da função EmissorProdutor() para o protocolo UDP bit alternante.

```
int ReceptorConsumidor(void **accesspt, void **peer_ap, idaddr_t *my_id, stub_t **return_buffer, int
closeconn)
{
    ...
    switch(transprotocol)
    {
        ...
        case UDPUNICAST:
            my_udpap = (udpaccpt *)*accesspt;
            pthread_mutex_lock (&conn_mutex);
            conn = search_connlist(&headnode_connlist, conn_index);
            if (conn == NULL){
                conn = insert_connlist(&headnode_connlist, conn_index);
                conn->udpap_conn.sd = Init_udp_Serv(&conn->udpap_conn.sock_addr,
                                                    UDP_BIT_ALT, my_udpap->port, my_udpap->address,
                                    udpap->interface_name);
                conn->udpap_conn.peer_sock_addr = (sockaddr_in *)malloc(sizeof(sockaddr_in));
                if(conn->udpap_conn.sd < 0)
                    return -1;
            }else{
                if(closeconn == CLOSE){
                    free(conn->udpap_conn.sock_addr);
                    remove_connlist(&headnode_connlist, conn_index);
                    pthread_mutex_unlock (&conn_mutex);
                    return 0;
                }
            }
            sd = conn->udpap_conn.sd;
            peer_sock_addr = conn->udpap_conn.peer_sock_addr;
            pthread_mutex_unlock (&conn_mutex);
            err = RecvfromServ(sd, &peer_sock_addr, ACK_YES, &recvpckt, 0);
            if(err < 0)
                return -1;

            remote_peer->transprotocol =  UDP_BIT_ALT;
            memcpy(remote_peer->address, inet_ntoa(peer_sock_addr->sin_addr),14);
            remote_peer->port = peer_sock_addr->sin_port;
            *peer_ap = (void **)remote_peer;
            break;
        ...
```

Figura 49:Implementação da função ReceptorConsumidor () para o protocolo UDP bit alternante.

```
char *BrokerHdrPacking(brokerstub_t **hdrdata,int **hdrlenght)
{
    char brokerhdr[1024];
    int msgsize;
    unsigned lhdr;
    unsigned short shdr;
    unsigned short transprotocol;
    brokerstub_t *hdr;
    msgsize = 0;
    hdr = *hdrdata;
    ...
    switch (transprotocol)
    {
         ...
        case UDP_BIT_ALT:
            udpap = (udpaccpt *)calloc(1,sizeof(udpaccpt));
            udpap->transprotocol    = ntohl(*(unsigned int *)&hdrpointer[i]); i +=sizeint;
            udpap->nictech          = ntohl(*(int      *)&hdrpointer[i]); i += sizeint;
            address.s_addr          = ntohl(*(unsigned long int *)&hdrpointer[i]);
            addrtemp                = inet_ntoa(address);
            memcpy(&udpap->address, &*addrtemp, sizeof(udpap->address)); i +=sizeint;
            udpap->port             = ntohl(*(unsigned int *)&hdrpointer[i]);i +=sizeint;
            hdrdata->rmtaccesspt[count] = (void *)udpap;
            count++;
        break;
    }
    ...
```

Figura 50: Implementação da função BrokerHdrPacking() para o protocolo UDP bit alternante.

```
brokerstub_t *BrokerHdrUnpack(char **brokerhdr)
{
    brokerstub_t *hdrdata;
    char *addrtemp, *hdrpointer;
    int i, sizeint, sizeshort, sizechar, sizeidaddr_t;
    struct in_addr address;
    int transprotocol;
    hdrpointer  = *brokerhdr;
    sizeint     = sizeof(unsigned int);
    ...
    switch (transprotocol)
    {
        case UDP_BIT_ALT:
            udpap = (udpaccpt *)calloc(1,sizeof(udpaccpt));
            udpap->transprotocol    = ntohl(*(unsigned int *)&hdrpointer[i]); i +=sizeint;
            udpap->nictech          = ntohl(*(int      *)&hdrpointer[i]);      i += sizeint;
            address.s_addr          = ntohl(*(unsigned long int *)&hdrpointer[i]);
            addrtemp                = inet_ntoa(address);
            memcpy(&udpap->address, &*addrtemp, sizeof(udpap->address)); i +=sizeint;
            udpap->port           = ntohl(*(unsigned int *)&hdrpointer[i]);i +=sizeint;
            hdrdata->rmtaccesspt[count] = (void *)udpap;
            count++;
        break;
        case ...
    }
  ...
```

Figura 51: Implementação da função BrokerHdrUnpack () para o protocolo UDP bit alternante.

5.3 *Overhead* inserido pelo Canal Virtual de Tempo-Real

No Canal Virtual de Tempo-Real, duas fontes de *overhead* são encontradas. A primeira fonte se refere ao acréscimo de tempo para o início das comunicações devido ao uso de *Brokers* para localização dos pontos consumidores na rede. Como visto em [28], este acréscimo é minimizado pelo artifício de *cache* e *prefetching*, sendo tolerável para aplicações de tempo-real não-crítico (*soft real-time*). Também, como o Canal Virtual de Tempo-Real não se baseia em um protocolo e comunicação em especifico ou em tecnologias de dispositivos de rede, devido ao *Framework* de Protocolos de Comunicação, os atrasos causados por determinado protocolo de comunicação ou tecnologia de dispositivos de rede poderão ser contornados com a substituição destes por outros mais apropriados.

A segunda fonte de *overhead* é o tempo acrescido na comunicação pelo próprio *middleware* de tempo-real. Para medir este acréscimo, foram realizadas duas medições de tempo de transmissão de dados entre uma aplicação produtora e uma consumidora, onde foram enviadas três mil mensagens de 1kb cada utilizando a estação de trabalho B para o cenário de teste (vide seção Cenários de teste). Este processo de envio foi repetido por dez vezes para ambas as medições. A primeira medição foi efetuada nas aplicações produtora e consumidora implementadas com o Canal de Virtual de Tempo-Real utilizando para comunicação o protocolo *Shmem* do *Framework.* A segunda medição foi feita com aplicações

produtoras e consumidoras implementadas apenas com o protocolo *Shmem* (sem o *middleware*). As Tabelas 2 e 3 mostram respectivamente os tempos de envio e recebimento das mensagens utilizando o Canal Virtual de Tempo-Real com o protocolo *Shmem* e utilizando somente o protocolo *Shmem*.

Tabela 2: Tempo de transmissão e recepção de mensagens utilizando o Canal Virtual de Tempo-Real.

Tamanho da mensagem	No. de mensagens por transmissão	Comunicação	Tempo de transmissão proc. Produtor (em segundos)	Tempo de recepção do proc. Consumidor (em segundos)
1kb	3000	*Canal Virtual TR + Shmem*	6.039245681	6.024724783
1kb	3000	*Canal Virtual TR + Shmem*	6.034712145	6.041634784
1kb	3000	*Canal Virtual TR + Shmem*	6.008424567	6.00648789
1kb	3000	*Canal Virtual TR + Shmem*	6.031798219	6.034212545
1kb	3000	*Canal Virtual TR + Shmem*	6.021834915	6.021756321
1kb	3000	*Canal Virtual TR + Shmem*	6.036982415	6.275698521
1kb	3000	*Canal Virtual TR + Shmem*	6.029601403	6.061287501
1kb	3000	*Canal Virtual TR + Shmem*	6.024085478	6.022396971
1kb	3000	*Canal Virtual TR + Shmem*	6.037394566	6.036987721
1kb	3000	*Canal Virtual TR + Shmem*	6.035583946	6.035637801
Total de transmissões: 10			**Tempo médio**	
			6.029966	6.056082

Tabela 3: Tempo de transmissão e recepção de mensagens utilizando somente o Protocolo Shmem

Tamanho da mensagem	No. de mensagens por transmissão	Comunicação	Tempo de transmissão proc. Produtor (em segundos)	Tempo de recepção do proc. Consumidor (em segundos)
1kb	3000	*Somente Shmem*	6.023574238	6.008125136
1kb	3000	*Somente Shmem*	6.002465326	6.007671283
1kb	3000	*Somente Shmem*	6.020191522	6.020164559
1kb	3000	*Somente Shmem*	6.029081462	6.029424721
1kb	3000	*Somente Shmem*	6.005768412	6.005789621
1kb	3000	*Somente Shmem*	6.031027562	6.027217936

1kb	3000	*Somente Shmem*	6.030642378	6.024596357
1kb	3000	*Somente Shmem*	6.005192456	6.004427898
1kb	3000	*Somente Shmem*	6.026721688	6.034567891
1kb	3000	*Somente Shmem*	6.006519464	6.001459723
Total de transmissões: 10			**Tempo médio**	
			6.018118	6.016345

Fazendo a subtração das médias dos produtores e dos consumidores da primeira e segunda situação obtemos respectivamente um *overhead* de 0.011848s para a ponta produtora e 0.039737s para a consumidora. Estes resultados mostram que o Canal pode muito bem substituir o uso direto de um protocolo de comunicação por apresentar um *overhead* baixo, cerca de 11ms a mais para o produtor e 39ms para o consumidor. Comparativamente, com o tempo de *overhead* do produtor seria o tempo necessário para ocorrer um *quantum* do sistema operacional de propósito geral Linux, que dura cerca de 10ms.

5.4 Teste de recuperação do canal na presença de falhas de Comunicação

Para efetuar os testes de falhas de comunicação, dois programas com as respectivas funções produtora e consumidora de mensagens foram implementados utilizando o Canal de Comunicação de Tempo-Real. Estes programas foram situados no cenário posto na seção Cenários de teste, onde a aplicação produtora é configurada com as opções *TRY_ALL_ROUTES* e *TRY_SPARE_NIC_FIRST* em cada um dois três cenários descritos na seção.

Para efetuar comunicação local, o canal consumidor disponibilizou dois pontos de acesso de memória compartilhada, e, para comunicação remota, outros dois pontos de acesso utilizando o protocolo UDP com janela de repetição e transmissão de tamanho um (também conhecido como *bit* alternante), cada qual associado a uma interface de rede. Para a simulação de tecnologias distintas de *NICs*, o campo *nictec* dos pontos de acesso remotos foram configurados com os valores BLUETOOTH_V1_1 para simular interfaces de rede *Bluetooth* e WI_FI para as interfaces com tecnologia *Wi-fi*.

A Tabela 4 mostra os tempos médios de recuperação do canal utilizando o primeiro cenário descrito anteriormente utilizando as políticas de redirecionamento de dados *TRY_ALL_ROUTES* e *TRY_SPARE_NIC_FIRST*. Nas Tabelas 5 e 6 são mostrados

respectivamente os tempos médios de recuperação no segundo e terceiro cenários utilizando as duas políticas para tratamento de falhas na comunicação.

Tabela 4: Tempo médio de transmissão no cenário de simulação com vária rotas alternativas entre comunicantes próximos

Cenário: 1				**Política de redirecionamento**:	
				TRY_ALL_ROUTES	*TRY_SPARE_NIC_FIRST*
Tamanho da mensagem	No. de mensagens por transmissão	Mensagem Retransmitida	Comunicação	Tempo de transmissão proc. Produtor (em segundos)	
1kb	10	5ª	*UDP Bit Alternante*	0.321368905	0.322359000
1kb	10	5ª	*UDP Bit Alternante*	0.311681210	0.311371012
1kb	10	5ª	*UDP Bit Alternante*	0.318602009	0.310541009
1kb	10	5ª	*UDP Bit Alternante*	0.305146003	0.306471003
1kb	10	5ª	*UDP Bit Alternante*	0.320638041	0.320317251
1kb	10	5ª	*UDP Bit Alternante*	0.312281013	0.312281013
1kb	10	5ª	*UDP Bit Alternante*	0.310305078	0.310305078
1kb	10	5ª	*UDP Bit Alternante*	0.305659001	0.305659001
1kb	10	5ª	*UDP Bit Alternante*	0.321410909	0.321410909
1kb	10	5ª	*UDP Bit Alternante*	0.313481012	0.312900112
Total de transmissões: 10				**Tempo médio**	
				0.3140573	0.3133615

Tabela 5: Tempo médio de transmissão no cenário de simulação com vária rotas alternativas entre comunicantes distantes

Cenário: 2				**Política de redirecionamento**:	
				TRY_ALL_ROUTES	*TRY_SPARE_NIC_FIRST*
Tamanho da mensagem	No. de mensagens por transmissão	Mensagem Retransmitida	Comunicação	Tempo de transmissão proc. Produtor (em segundos)	
1kb	10	5ª	*UDP Bit Alternante*	0.320638041	0.463470012
1kb	10	5ª	*UDP Bit Alternante*	0.312281013	0.472719014
1kb	10	5ª	*UDP Bit Alternante*	0.318602009	0.465992004
1kb	10	5ª	*UDP Bit Alternante*	0.305146003	0.475143999
1kb	10	5ª	*UDP Bit*	0.32153041	0.466246992

			Alternante		
1kb	10	5ª	*UDP Bit Alternante*	0.312283513	0.468323002
1kb	10	5ª	*UDP Bit Alternante*	0.312920078	0.46356010
1kb	10	5ª	*UDP Bit Alternante*	0.321368905	0.464357005
1kb	10	5ª	*UDP Bit Alternante*	0.311678109	0.454473812
1kb	10	5ª	*UDP Bit Alternante*	0.318602109	0.472819231
Total de transmissões: 10				**Tempo médio**	
				0.315505	0.4667105

Como podemos observar na Tabela 4, o desempenho de ambas as políticas são idênticos, pois, independente da rota escolhida para o encaminhamento das mensagens esta conseguira atingir o ponto de acesso do destinatário. Em contrapartida, em cenários de rede onde os comunicantes raramente estão a uma distância mínima para que possa haver comunicação e onde há várias rotas alternativas entre os pontos de acesso dos dispositivos, representado pela Figura 37 a política *TRY_ALL_ROUTES* se mostra mais atraente, pois, esta política aumenta as chances de entrega das mensagens devido ao número de alternativas de rota oferecidas. Contudo, em cenários onde os dispositivos se encontram frequentemente próximos o bastante para se comunicarem (como ilustrado na Figura 38), a política *TRY_SPARE_NIC_FIRST* é mais adequada, por conseguir um redirecionamento de dados bem sucedido logo na primeira tentativa, conforme os tempos obtidos Tabela 6.

Tabela 6: Tempo médio de transmissão no cenário de simulação com apenas uma rota entre comunicantes próximos

Cenário: 3				**Política de redirecionamento**:	
				TRY_ALL_ROUTES	*TRY_SPARE_NIC_FIRST*
Tamanho da mensagem	No. de mensagens por transmissão	Mensagem Retransmitida	Comunicação	Tempo de transmissão proc. Produtor (segundos)	
1kb	10	5ª	*UDP Bit Alternante*	0.463470012	0.321359009
1kb	10	5ª	*UDP Bit Alternante*	0.472719014	0.311381012
1kb	10	5ª	*UDP Bit Alternante*	0.465992004	0.310602009
1kb	10	5ª	*UDP Bit Alternante*	0.475143999	0.306766003

1kb	10	5ª	*UDP Bit Alternante*	0.466246992	0.320319041
1kb	10	5ª	*UDP Bit Alternante*	0.468317002	0.312281013
1kb	10	5ª	*UDP Bit Alternante*	0.466576010	0.310305078
1kb	10	5ª	*UDP Bit Alternante*	0.474341005	0.305659001
1kb	10	5ª	*UDP Bit Alternante*	0.464470012	0.321410909
1kb	10	5ª	*UDP Bit Alternante*	0.472819011	0.312381112
Total de transmissões: 10				**Tempo médio**	
				0.4690095061	0.3132464

6 CONCLUSÕES E TRABALHOS FUTUROS

Baseado na evolução dos sistemas distribuídos de tempo real, no suporte às aplicações de tempo real pelos sistemas operacionais de propósito geral, e no aumento do número de tecnologias de interfaces de rede nos sistemas computacionais atuais, o Canal Virtual de Tempo-Real foi projetado e implementado, de forma a tirar vantagem das funcionalidades e características de tempo-real oferecidas pelos sistemas operacionais de propósito geral (em especial o S.O. Linux utilizado na implementação do *middleware*) e das alternativas de conectividade proporcionadas pelas múltiplas interfaces de rede e tecnologias de rede, contidas nos dispositivos computacionais, como *handhelds* e *laptops*.

Como estudo de caso, foi implementada a arquitetura descrita no capitulo 4 baseada no sistema operacional Linux, que utiliza dois protocolos para testar a funcionalidade do *Framework* de Protocolos de Comunicação. Foram feitas também medições de *overhead* de canal e testes de recuperação sob a presença de falha de comunicação utilizando os cenários lógicos e reais descritos na seção 5.1.

O teste de *Framework* foi bem sucedido, comprovando a capacidade de acrescentar outros protocolos de comunicação no Canal Virtual de Tempo-Real através dos roteiros para inclusão de protocolos no *framework*, descritas na seção 4.4, sem que haja modificações em partes da arquitetura diferentes das destinadas ao *framework*.

No teste de *overhead* inserido pela arquitetura do Canal, foram extraídos os tempos de transmissão entre duas aplicações, inicialmente utilizando o *middleware* de tempo-real com o protocolo *Shmem* para comunicação e finalmente sem a utilização do *middleware*, utilizando diretamente o protocolo *Shmem*. Dessa forma foi possível fazer a extração do *overhead* através da subtração dos tempos do cenário com uso do *middleware* e do cenário sem o uso do *middleware*. Como os *overheads* obtidos são mínimos, o uso do canal é tido como viável e pode substituir o uso direto de protocolos de comunicação.

Finalmente, o teste de recuperação de Canal sob falhas de Comunicação foi realizado através da medição das duas políticas de tratamento de falhas de comunicação nos três cenários de teste. A política *TRY_ALL_ROUTES*, conforme as medições de tempo tomadas nas duas situações descritas na seção 5.1, obteve desempenho semelhante à política

TRY_SPARE_NIC_FIRST no primeiro cenário devido às múltiplas rotas, que tornam qualquer caminho alternativo válido.

No segundo cenário, a política *TRY_ALL_ROUTES* mostrou-se mais efetiva devido à distância entre os comunicantes impossibilitar o acerto de rota na primeira tentativa pela política *TRY_SPARE_NIC_FIRST*. Finalmente, no último cenário a política *TRY_SPARE_NIC_FIRST* foi mais efetiva por escolher corretamente a rota logo na primeira tentativa, fato que não ocorre com a política *TRY_ALL_ROUTES* no cenário.

Com relação aos trabalhos futuros, pode-se estender a arquitetura para lidar com o controle da escolha de interface de rede com base no estado de energia da bateria do dispositivo computacional móvel e da quantidade de dados a serem enviados. Dessa forma, dependendo da quantidade de dados a serem enviados e do consumo de energia das interfaces de rede, o *middleware* de tempo real optaria pela interface que proporcionasse maior economia de energia. Assim, os canais teriam o potencial de aumentar a autonomia dos dispositivos móveis.

7 REFERÊNCIAS BIBLIOGRÁFICAS

[1] PALM. Palm Inc., *http://www.palm.com/us/*.

[2] BLUETOOTH. The official Bluetooth membership site, *https://www.bluetooth.org/*.

[3] IRDA. Infraread Data Association, *http://www.irda.org /*.

[4] WI-FI. Wi-Fi Alliance, *http://www.wi-fi.org/*.

[5] YOUNG, S., *Real-Time Languages: Design and Development*, Ellis Horwood, 1982.

[6] SOFTEC**,** Integrated SoftTech Solutions P Ltd*. Linux for Real Time Requirements*. 2004.

[7] LAUER H.; SATTEWWAITE, E., *The impact of Mesa on System Design,* In Proceeding of the 4th international Conference on Software engineering, pp 174-82. IEEE (1979).

[8] CORNHILL, D. et al, Limitations of Ada for Real Time scheduling. Proc International workshop on Real Time Ada Issues. ACM Ada Letters, pp. 33-9, 1987.

[9] METROWERKS, Metrowerks Inc., *Linux as a Real-Time Operating System*, 12 de Maio de 2003.

[10] MAN. Linux Man Pages, *http://unixhelp.ed.ac.uk/CGI/man-cgi?sched_setscheduler+2*, Consultado em 10/01/06.

[11] RIBEIRO, P., R., J., *Gerador de programas paralelos de tempo real para um ambiente visual*. Universidade Federal de São Carlos, Centro de Ciências Exatas e de Tecnologia, Dissertação de Mestrado PPG-CC, São Carlos, 1999.

[12] EONIC. *Virtuoso™ user guide: book 2 for Version 4.2***,** Eonic Systems, 2000.

[13] HOARE, C. A. R., *Communicating Sequential Proce*sses. Editora Prentice-Hall International, Junho 2004

[14] GAY, S.; SCHNEIDER, S., *Concurrent and Real-Time Systems.* <http://www.uoguelph.ca/~gardnerw/courses/cis6650/res/CRS.pdf>, Consultado em janeiro 2005.

[15] BAKKERS, A.; Hilderink, G.; Broenink, J., *A Distributed Real-Time Java System Based on CSP*, Control Laboratory, University of Twente, USA, 1999.

[16] HEIMFARTH; T. , GÖTZ, M.; RAMMING, J., F.; WAGNER, R., F., *RTC: A Real-time Communication Middleware on Top of RTAI-Linux*, Universidade Federal do Rio Grande do Sul – Departamento de Informática - Brasil, Universidade de Paderborn – Instituto Nixdorf Heinz – Alemanha, 2003.

[17] TANENBAUM, A., S. Redes de Computadores. Editora Campus, 4ª Edição.

[18] GERUM, P., *Real Time Application Interface*, URL: http://www.rtai.org/modules.php?name=Content&pa=showpage&pid=1, Consultado em 10/03/2005.

[19] PUGN, O.; COWHIG, J; CROWE, J. BLACKLOCK, E., *Scalable c-Coherent Interface (SCI)*, URL: http://ntrg.cs.tcd.ie/undergrad/4ba2.05/group12/, Consultado em 10/03/2005.

[20] IEEE, 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI)*, Piscataway, NJ: IEEEService Center, 1993.

[21] LANKER, S.;PFEIFFER, M.; BEMMERL, T., *Design and Implementation of a SCI-Based Real-Time CORBA*. Lehrstuhl für Betriebssysteme, RWTH Aachen, ISORC 2001.

[22] Compaq, Microsoft, e Intel, *Virtual Archiecture Specification Version 1.0*, Relatório técnico, Compaq, Microsoft, e Intel, Dezembro 1997.

[23] INFINIBAND. Infiniband Trade Association, *Infiniband Archiecture Specification, Release 1.0*, *http://www.infinibandta.org*. Consultado em 01/03/2005.

[24] RUBINI, A., *Linux device drivers*. O'Reilly & Associates Inc., 1998.

[25] YAGHMOUR, K. *The Real-Time Application Interface*,2001.

[26] ABDELZAHER, T., DAWSON, W., FENG, C., JAHANIAN, F., JOHNSON, S., MEHARA, A., MITTON, T., SHAIKH, A., SHIN, K., WANG, Z., ZOU, H., *ARMADA Middleware and Communication Services*. Laboratório de Computação de Tempo-Real, Departamento de Engenharia Elétrica e Ciência da Computação, Universidade de Michigan.

```
int
EmissorProdut
or(char
*pckt, int
pcktsize,
void
**my_accesspt
, void **
```


[27] ZELENOVSKY, R., MENDONÇA, A., *Introdução aos Sistemas Embutidos*, URL: http://www.mzeditora.com.br/artigos/embut.htm, Consultado em 20/02/2005.

[28] ROSSLER, A., D., *Integração dos Requisitos Temporais de um Kernel de Tempo-Real e de sua Comunicação em Redes*, São Carlos, 2004, p. 153, Dissertação Mestrado PPG-CC, Universidade Federal de São Carlos.

[29] KELLER, R. Threads vs process. *http://www.cs.hmc.edu/courses/2001/spring/cs156/htmlthreads/*, February 2001

[30] TANENBAUM, A. S., *Modern Operating System*. PrenciteHall, 1992.

[31] The Alternating Bit Protocol. *Bhttp://staff.science.uva.nl/~psf/specifications/abp.html*. Consultado em 15/01/06.

[32] WIKPEDIA. GNU General Public license, *http://en.wikipedia.org/wiki/GNU_General_Public_License*. Consultado em 10/12/2005.

[33] RTLINUX. FSMLabs RT-LINUX, *http://www.fsmlabs.com/*.

[34] RTAI. RTAI - The RealTime Application Interface for Linux from DIAPM, *https://www.rtai.org/*.

[35] SLACKWARE. The Slackware Linux Project, *http://www.slackware.com/*.

[36] MANDRAKE . Linux Mandrake, *http://www.linux-mandrake.org/*.

[37] MULINUX. , *http://mulinux.dotsrc.org/*.

[38] POSIX. What is POSIX, *http://www.lynuxworks.com/products/posix/posix.php3*. Consultado em 01/07/2005.

[39] CHRISTOPHER, T. W., Animation of multiple readers/writers Algorithms, *http://www.tools-of-computing.com/tc/CS/Monitors/multiplereaderswriters.htm*. Consultado em 10/01/2006.