View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Adamô Dal Berto

Projeto e Implementação de um Protocolo de Comunicação para o Ambiente RTAI-RTNET.

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Luís Fernando Friedrich, Dr.
Orientador

Florianópolis, Dezembro de 2003

Projeto e Implementação de um Protocolo de Comunicação para o Ambiente RTAI-RTNET.

Adamô Dal Berto

Esta Dissertação foi aprovado em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação

____

Prof. Raul Sidnei Wazlawick, Dr.

Coordenador do Curso

Banca Examinadora

____

Prof. Luís Fernando Friedrich, Dr.

Orientador

____

Prof. Alexandre Moraes Ramos, Dr.

____

Prof. José Mazzucco Jr., Dr.

____

Prof. Rômulo Silva de Oliveira, Dr.

... e a história desse bochincho faz parte do meu passado.
"Jayme Caetano Braun"
(1924 - 1999)

Agradecimentos

Como não poderia ser diferente, os meus primeiros pensamentos de gratidão são dirigidos, a meus pais Arlete e Nelson, pois a eles devo, além de minha existência, o enorme sacrifício por eles feito para minha educação.

A Deus, pelo dom supremo da vida, por tudo que tenho e sou.

A todos os meus familiares, pelas suas presenças e apoio incondicional em todas as etapas de minha vida. A meus tios Alcidor e Docinha. A meus primos Aldo e Adriana.

A Marcelle, pelo incentivo, carinho, dedicação, apoio em todos os momentos. E também pela sua impaciência para ver este trabalho concluído.

Ao meu grande amigo e mestre Prof. Dr. Simão Sirineo Toscani, pelo apoio e motivação, fundamental para o início desta jornada.

A todos os meus amigos, pois este trabalho também não poderia ser realizado sem o apoio moral fornecido pelos grandes amigos pessoais que fiz durante o curso e pelos que trago de longa data. Desta forma agradeço aos meus novos amigos André, Beto, Ricardo Zago, Renato Noal e ao pessoal da antiga, Régis, Luciano, Gerson, Tiago, Edison, Luis Carlos, Marco Antônio, Rafael Bertei, Kiko, Ricardo (HP) e Cassiano. Em especial agradeço a Rafael Speroni, Fernando Barreto, e Neves pelos conselhos e auxílios prestados em vários momentos desta caminhada.

Aos novos amigos que fiz agradeço pelo companheirismo, sua amizade e pelos momentos agradáveis que juntos passamos. Aos velhos companheiros por compreenderem minha ausência em certos momentos.

Ao meu orientador, Prof. Dr. Luís Fernando Friedrich, pela dedicação prestada no decorrer de todo o trabalho, por sua atenção e amizade.

A todos os demais professores e servidores do Departamento de Informática e Estatística da UFSC. Destes em especial, cabe meu mais sincero agradecimento a Verinha

pela sua sempre pronta atenção e disponibilidade na resolução dos problemas.

Agradeço também a todos aqueles que fizeram parte desta caminhada e por ventura os tenha esquecido durante a escrita deste singelo agradecimento.

Sumário

Lista de Figuras ix

Lista de Tabelas xi

Lista de Siglas xii

Resumo xiii

Abstract xiv

1 Introdução 1

1.1 Motivação . . . 1

1.2 Justificativa . . . 2

1.3 Objetivos . . . 3

1.4 Organização do Trabalho . . . 3

2 Sistemas de Tempo Real: Uma Revisão 5

2.1 O ambiente de Tempo Real . . . 5

2.2 Classificação dos Sistemas de Tempo Real . . . 6

2.3 Escalonamento de Tarefas . . . 8

2.4 Escalonamento de Tarefas de Tempo Real . . . 9

2.4.1 Abordagens de Escalonamento . . . 9

2.5 Escalonamento de Tarefas Periódicas . . . 12

2.5.1 RMA (*Rate Monotonic Algorithm*) . . . 13

2.5.2 EDF (*Earliest Deadline First*) . . . 14

2.5.3 DM (*Deadline Monotonic*) . . . 14

2.6 Análise de Escalonabilidade 15
2.6.1 Testes de Escalonabilidade 16
2.7 Inversão de Prioridades 17
2.7.1 Protocolo de Herança de Prioridade (*Priority Inheritance Protocol*) 17
2.7.2 Protocolo de Prioridade Teto (*Priority Ceiling Protocol*) 18

3 Modelo de Referência OSI e as Comunicações em Tempo Real 19
3.1 Modelo de Referência OSI (*RM OSI*) 19
3.2 As camadas do modelo OSI 20
3.3 Classificação dos Protocolos de Controle de Acesso ao Meio (MAC) . . . 22
3.3.1 Classificação dos Protocolos MAC quanto à Alocação do Meio . . 22
3.3.2 Classificação dos Protocolos MAC quanto ao comportamento temporal 23
3.4 Tecnologia ATM e as Aplicações de Tempo Real 30
3.5 Abordagens para a comunicação de Tempo Real 31

4 O Sistema Operacional Linux e o Tempo Real 33
4.1 Núcleo do Linux 33
4.2 O Linux como um SO de Tempo Real 36
4.3 Linux X Tempo Real Crítico 37
4.4 Modificando o Linux 39
4.5 Os benefícios do Linux como Sistema de Tempo Real Brando 39
4.6 Extensões do Linux para Tempo Real 40
4.6.1 RED LINUX 40
4.6.2 KURT LINUX 42
4.6.3 RTLINUX 43

5 O Ambiente de Tempo Real RTAI e RTNET 49
5.1 RTAI LINUX 49
5.2 Princípios de Implementação 50
5.3 Camada de Abstração de Hardware (*Hardware Abstraction Layer*) 51
5.4 Escalonamento no RTAI 52

5.5 Características do RTAI 53

5.5.1 Comunicação Interprocessos (IPC) 53

5.5.2 Gerenciamento de Memória 56

5.6 LXRT: Interface de Tempo Real para processos em modo usuário 56

5.7 RTNET 58

5.7.1 Histórico 58

5.7.2 Fundamentos 59

5.7.3 Princípios de Implementação 60

5.8 RTMAC 61

5.9 RTNETPROXY 63

6 Projeto e Implementação do Protocolo SRTP 65

6.1 Especificação de Protocolos 65

6.2 Definição do Protocolo SRTP 66

6.3 Implementação do SRTP 69

6.4 Visão Geral do Protocolo 69

6.5 Componentes do protocolo SRTP 72

6.5.1 Transmissão de Pacotes no SRTP 72

6.5.2 Recepção de Pacotes no SRTP 75

6.6 Controle de Erros no SRTP 77

6.7 Condições de Corrida no módulo 78

7 Análise de Resultados 81

8 Conclusão 85

Referências Bibliográficas 88

Lista de Figuras

2.1 Mérito Temporal a) Serviço de tempo real brando b) Serviço de tempo real crítico [14] . . . 7
2.2 Taxonomia de abordagens de escalonamento de tempo real . . . 10
2.3 Testes de Escalonabilidade [19] . . . 16

3.1 As sete camadas do Modelo de Referência OSI . . . 20

4.1 Núcleo do Linux [58] . . . 34
4.2 Princípio de Implementação do RTLINUX . . . 44
4.3 Controle de Interrupções no RTLINUX . . . 47

5.1 Visão geral da arquitetura RTAI Linux [37]. . . . 50
5.2 Tratamento de Interrupções no RTAI [37]. . . . 51
5.3 Estrutura básica do RTNET [60]. . . . 59
5.4 Rede RTNET [59]. . . . 60
5.5 RTNET com o módulo RTMAC carregado [60]. . . . 61
5.6 Ciclo TDMA no RTMAC [60]. . . . 62
5.7 A Atribuição de *offsets* no ciclo RTMAC [60]. . . . 63
5.8 Integração com a pilha Linux TCP/IP [59]. . . . 64

6.1 Sistemas SRTP. . . . 67
6.2 Função de transição para o subsistema transmissor δ_T. . . . 67
6.3 Função de transição para o subsistema receptor δ_R. . . . 68
6.4 Comparativo entre as pilhas de protocolo . . . 71
6.5 Formato dos pacotes SRTP. . . . 73

6.6 Fluxo normal do Protocolo SRTP. 78
6.7 Fluxo com ocorrências de erros no Protocolo SRTP. 79

7.1 Função para a obtenção de tempos em nanossegundos [42]. 82
7.2 Valores médios obtidos. 83

Lista de Tabelas

3.1 Resumo das principais abordagens para a problemática de tempo real. . . 31

5.1 Requisitos para a utilização do RTNET [60] 59

6.1 Estrutura de controle de mensagens SRTP 70

6.2 Funções RTAI-RTNET utilizadas na implementação 71

Lista de Siglas

ATM	Asynchronous Transfer Mode
CPU	Central Processing Unit
CSMA	Carrier Sense Multiple Access
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSMA/DCR	CSMA with Deterministic Collision Resolution
DM	Deadline Monotonic
DOS	Disk Operating System
EDF	Earliest Deadline First
FIFO	First-In First Out
IDT	Interrupt Descriptor Table
IP	Internet Protocol
IPC	InterProcess Communication
LAN	Local Area Network
LLC	Logical Link Control
MAC	Medium Access Control
MUP	MultiUniPprocessors
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
RM	Rate Monotonic
RMA	Rate Monotonic Algorithm
RTHAL	Real-Time Hardware Abstraction Layer
SMP	SymmetricMultiProcessors
SO	Sistema Operacional
SOTR	Sistema Operacional de Tempo Real
SRTP	Simple Real-Time Protocol
TCP	User Datagram Protocol
TDMA	Time Division Multiple Acces
TR	Tempo Real
UDP	User Datagram Protocol
UP	UniProcessor
WAN	Wide Area Network

Resumo

A utilização de sistemas computacionais vem obtendo grande abrangência nas mais diversas áreas de conhecimento. Dentre as atividades a que estes podem estar relacionados, existem aquelas que possuem alguns requisitos temporais. Os sistemas de tempo real são aqueles cujo correto funcionamento está associado, não apenas às respostas corretas a determinados estímulos, mas também às restrições temporais impostas pelo ambiente. Muitas pesquisas vêm sendo desenvolvidas com o objetivo de dar suporte a sistemas de tempo real através de soluções proprietárias ou extensões para sistemas operacionais de propósito geral. O RTAI é uma extensão do Linux para tempo real, cuja pilha de protocolo de rede é chamada RTNET. Neste trabalho é apresentada a implementação de um protocolo alternativo de rede, na forma de um módulo, para comunicação de tarefas em modo usuário que desejem fazer uso da pilha RTNET. São apresentados, ainda, os resultados dos testes de forma comparativa aos protocolos UDP/IP.

Abstract

Computational systems utilization have broadening towards various knowledge areas. Among the activities that these systems may be related to, are the ones with temporal constraints. In real-time systems the right functioning is related not only to the right responses to determined stimulus, but also to the temporal constraints imposed by the environment. Many researches has been developed aiming to support real-time systems through proprietary solutions or general-purpose operating systems extensions. RTAI is a Linux extension for real-time, which uses the network protocol stack RTNET. In this work is presented an alternative network protocol, implemented as a module, for user mode tasks communication through RTNET stack. It's also presented the tests results, for comparative purposes to the UDP/IP protocols.

Capítulo 1

Introdução

1.1 Motivação

A utilização de sistemas computacionais vem obtendo grande abrangência nas mais diversas áreas de conhecimento. Dentre as atividades a que tais sistemas podem estar relacionados, existem aquelas cuja correta execução não está vinculada somente aos resultados lógicos obtidos, mas também nos seus requisitos temporais. A estes sistemas dá-se o nome de sistemas de tempo real.

Nesta classe de sistemas podem estar contidos desde simples controladores de dispositivos inteligentes até sistemas mais complexos, como é o caso de sistemas de controle de tráfego aéreo. Independentemente da complexidade e tamanho, o que caracteriza estes sistemas é a preocupação com a garantia no cumprimento dos prazos.

De acordo com as possíveis conseqüências do não atendimento dos prazos impostos, os sistemas de tempo real são classificados em brandos ou rígidos. No primeiro caso, o resultado pode ter alguma utilidade, ainda que sua produção ocorra após o término do prazo. Já nos sistemas rígidos, uma vez ultrapassado o prazo, os resultados não têm mais valor.

Em virtude do atendimento de requisitos temporais não estar entre os objetivos dos sistemas operacionais de propósito geral, muitas pesquisas vêm sendo realizadas com o objetivo de dar suporte a sistemas de tempo real, através do desenvolvimento de soluções proprietárias ou extensões para sistemas operacionais de propósito geral.

O RTAI é uma extensão do Linux para tempo real, cuja pilha de protocolo de rede

chamada RTNET foi implementada posteriormente, com o objetivo de prover suporte de tempo real para aplicações que utilizem rede, através do *hardware Ethernet*, cuja utilização é amplamente difundida.

O RTNET implementa comunicação através dos protocolos UDP/IP, que não se mostra adequado para todos os tipos de aplicações de tempo real, devido a dois problemas:

- **Desempenho:** No contexto de redes locais, a utilização do UDP/IP ocasiona um custo adicional desnecessário, tanto em função do tamanho das mensagens quanto no processamento dos cabeçalhos. Em ambientes de tempo real, onde as comunicações são geralmente leves, este custo pode vir a ser muito elevado em relação ao tamanho das mensagens que trafegam pela rede;

- **Confiabilidade:** Mesmo com avanço da tecnologia de transmissão de dados alguns ambientes hostis ainda podem produzir uma quantidade de ruído capaz de causar erros na comunicação. Estes erros não podem ser tratados pelo UDP/IP, uma vez que este não garante a entrega das mensagens enviadas por ele.

Este trabalho apresenta o protocolo SRTP (*Simple Real-Time Protocol*), que, baseado no sistema RTNET, implementa funcionalidades leves de controle de erros em apenas três camadas (ao contrário das 5 do UDP/IP). O protocolo proposto visa oferecer uma alternativa para a comunicação de tarefas de tempo real, em espaço usuário (tarefas LXRT).

Apesar do protocolo ser implementado em um sistema de tempo real e, portanto, utilizar as características destes, não figura entre os objetivos do SRTP, prover garantias temporais. Deixando estas garantias a cargo de protocolos de controle de acesso ao meio que podem ser implementados em nível de enlace ou aplicação.

1.2 Justificativa

O funcionamento do RTNET baseia-se nos protocolos UDP/IP, que são adequados para redes de larga escala, como a Internet. Em redes locais, entretanto, a utilização deste tipo de protocolo ocasiona um custo adicional desnecessário em cada camada, tanto em função do tamanho das mensagens quanto no processamento adicional no tratamento destas. Em

ambientes de tempo real, onde as comunicações são, geralmente, leves, este custo pode vir a ser muito elevado em relação ao tamanho das mensagens que trafegam pela rede.

Além disso, a utilização da pilha UDP/IP, implementados no RTNET, não pode ser considerada confiável, uma vez que estes protocolos não fazem tratamento de erros, deixando esta tarefa ao encargo das aplicações.

Desta forma faz-se necessário um protocolo que seja mais leve e realize algum tipo de tratamento de erros, como o apresentado neste trabalho.

1.3 Objetivos

O objetivo principal deste trabalho é implementar um protocolo leve e confiável para o ambiente de tempo real RTAI-RTNET. Dentre os objetivos específicos do trabalho podemos citar:

- Fazer um estudo do sistema operacional aberto RTAI e de suas capacidades na implementação de sistemas de tempo real;
- Estudar o sistema de comunicação RTNET, suas capacidades e limitações como framework para a implementação de novos protocolos;
- Projetar formalmente um protocolo de comunicação simples e leve, que atenda requisitos de confiabilidade de maneira eficiente;
- Implementar tal protocolo como um módulo do núcleo, baseando-se nas funcionalidades do RTAI-RTNET;
- Validar e testar tal protocolo através de comparativos do mesmo em relação a soluções já existentes.

1.4 Organização do Trabalho

O conteúdo deste trabalho está organizado em oito capítulos, conforme descritos a seguir:

O capítulo 2 apresenta a conceituação básica dos sistemas de tempo real apresentando as definições e terminologias mais utilizadas no ambiente de tempo real.

No capítulo 3 são apresentadas as definições relativas à comunicação de tempo real, bem como suas classificações em relação ao comportamento temporal. São aprofundadas as classificações relativas aos protocolos de enlace, mais especificamente em sua subcamada de controle de acesso ao meio (MAC);

O capítulo 4 aborda a utilização do sistema operacional Linux no que se refere ao tratamento de tarefas com requisitos temporais e algumas de suas extensões para tempo real;

No capítulo 5 são apresentados o ambiente de tempo real RTAI-RTNET em que se baseia a implementação deste trabalho;

O capítulo 6 apresenta as considerações relativas ao projeto e a implementação do protocolo SRTP;

No capítulo 7 estão descritos os testes realizados, bem como a análise dos resultados dos mesmos através da comparação entre o protocolo SRTP e os protocolos UDP/IP implementados no Linux;

O capítulo 8 apresenta as conclusões e as considerações finais a respeito do trabalho realizado, bem como sugestões para trabalhos futuros.

Capítulo 2

Sistemas de Tempo Real: Uma Revisão

Os sistemas computacionais têm se tornado muito presentes em todas as áreas das atividades humanas. Devido à vasta gama de campos de atuação a que atendem tais sistemas abrangem muitas atividades que possuem requisitos temporais.

As aplicações podem variar muito com relação à sua complexidade e necessidades de garantia em atender restrições temporais. Entre os sistemas mais simples pode-se destacar a grande variedade de equipamentos eletrodomésticos dotados de controladores inteligentes. Já no outro extremo encontram-se sistemas que possuem alto grau de complexidade, tais como sistemas militares, sistema de controle de plantas industriais, alguns sistemas hospitalares e sistemas de controle de trafego aéreo. Assim, pode-se constatar que algumas aplicações possuem restrições temporais muito mais rígidas do que outras.

2.1 O ambiente de Tempo Real

Um sistema computadorizado dito de tempo real é aquele cujo correto desempenho não está somente associado a uma resposta adequada a um dado estímulo (*correctness*), mas também a limites de tempo de execução (*timeliness*), isto é, enquanto ela ainda se faz necessária [51].

Existem muitos mal entendimentos sobre computação de tempo real. É comum que este conceito esteja associado à velocidade de processamento. Entretanto, um sistema de tempo real não possui como requisito básico o fato de ser um sistema veloz. Sua característica fundamental deve ser a previsibilidade temporal [50].

Mesmo que o termo "tempo real" seja utilizado em muitas aplicações, ele pode estar sujeito a diferentes interpretações. Diz-se que um sistema de controle opera em tempo real se ele é capaz de responder rapidamente a eventos externos. Pode-se interpretar isto da seguinte forma: um sistema é considerado de tempo real se ele é rápido. Entretanto, o termo rápido tem um significado relativo e não captura propriedades principais que caracterizam este tipo de sistemas.

O conceito de tempo é uma propriedade intrínseca dos sistemas de controle. Não faz sentido projetar um sistema de computação tempo real para controle de vôos sem considerar as características de tempo das aeronaves. Desta forma, o ambiente é sempre um componente essencial para qualquer sistema tempo real.

Enquanto o objetivo da computação rápida é minimizar o tempo de resposta (média) de um conjunto de tarefas determinado, o objetivo da computação de tempo real é atender requisitos de tempo individuais de cada tarefa [19].

Os sistemas computadorizados de tempo real, normalmente, fazem parte de sistemas muito mais amplos, composto da seguinte forma [29]:

- Interface
- Sistema computadorizado
- Interface de instrumentação (Sensores e Atuadores)
- Objeto a controlar

2.2 Classificação dos Sistemas de Tempo Real

Os sistemas de tempo real estão divididos basicamente em duas classes: os sistemas ditos críticos (*Hard Real-Time*) e os brandos (*Soft Real-Time*) [47], [19]. Esta classificação se dá em virtude dos requisitos temporais, prazos (*deadlines*) a que estes estão sujeitos.

Um deadline representa o prazo ao fim do qual uma determinada tarefa deverá ser concluída. Em sistemas de tempo real, uma tarefa pode estar sujeita a prazos críticos (*hard deadlines*) ou a prazos brandos (*soft deadlines*), a não conclusão da tarefa dentro de seu prazo máximo de execução terá respectivamente, conseqüências catastróficas ou mais brandas.

Um sistema dito de tempo real crítico é aquele que apresenta pelo menos um prazo crítico a ser cumprido. O resultado do não cumprimento deste limite de tempo, normalmente gera uma situação de catástrofe (exemplo: controladores de um reator nuclear).

No contexto de sistemas de tempo real brando supõe-se que, se um prazo não é cumprido, as possíveis avarias temporais podem ser ditas benignas, uma vez que não acarretam forma alguma de catástrofe, mas apenas uma pequena deterioração do serviço prestado [41]. A figura 2.1, sintetiza a classificação das tarefas, levando em consideração o mérito temporal das mesmas.

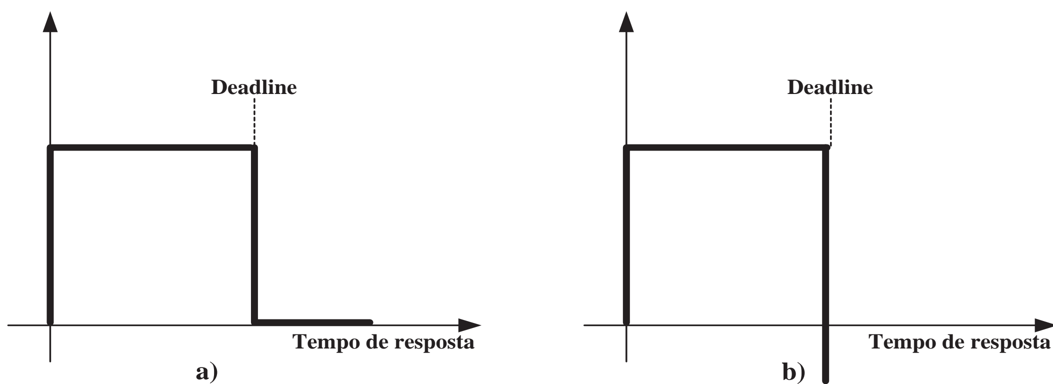

Figura 2.1: Mérito Temporal a) Serviço de tempo real brando b) Serviço de tempo real crítico [14]

O que diferencia de forma marcante esses dois tipos de sistemas de tempo real é que em uma situação anômala, os sistemas críticos devem continuar a cumprir seus prazos. Nestes sistemas, essas situações devem ser antecipadamente previstas durante a fase de projeto do sistema. Para Koeptz e Veríssimo [30], os sistemas de tempo real críticos devem ser equacionados com base em um modelo de carga máxima (*load hypothesis*) e em um modelo de falhas (*fault hypothesis*).

Segundo Farines *et al.* [19], os sistemas de tempo real crítico ainda podem subdividir-se em duas classes: seguros e operacionais em caso de falhas.

Seguros em caso de falhas (*Hard Real-Time Fail Safe*): nesta classe de sistemas, está prevista a existência de um ou mais estados ditos seguros que podem ser atingidos em caso de falhas[27];

Operacionais em caso de falha (*Hard Real-Time Fail Operacional*): esta classe de

sistemas trabalha com a idéia de que ao apresentar falhas parciais, este possa se degradar fornecendo alguma forma de serviço mínimo, mas ainda seguro.

As aplicações de tempo real críticas possuem como característica a necessidade de detectar os erros dentro de um curto espaço de tempo e com um grande grau de probabilidade. O tempo para a detecção do erro deve ser da mesma ordem de grandeza do período de amostragem do sistema de controle mais rápido, tornando assim, possível do ponto de vista temporal, que alguma ação corretiva sobre o sistema possa ser tomada em caso de falha.

Em Farines *et al.*[19] encontra-se, ainda, outra forma de distinção entre os sistemas de tempo real. A visão apresentada na literatura por alguns autores, leva em conta o ponto de vista da implementação. Tal forma de distinguir os sistemas de tempo real faz com que os mesmos sejam classificados das seguintes formas: os sistemas de resposta garantida e os sistemas de melhor esforço.

Os sistemas de resposta garantida (*guaranteed response system*) são aqueles onde existem recursos suficientes para suportar a carga de pico e cenários de falha definidos. Os sistemas do melhor esforço (*best effort system*) são aqueles que fundamentam-se na estratégia de alocação dinâmica de recursos, baseando-se em estudos probabilistas sobre a carga e os cenários de falhas aceitáveis.

2.3 Escalonamento de Tarefas

O “mundo real” é inerentemente concorrente, uma vez que existem inúmeros recursos que por serem escassos precisam ser compartilhados por diversas entidades da mesma forma que são incontáveis as entidades que, ao longo do tempo, rivalizam entre si para usufruir um recurso escasso.

Em um sistema computacional, um processador, ou qualquer outro componente do sistema torna-se um recurso escasso quando a ele é confiada a execução de duas ou mais tarefas que, idealmente, deveriam desenvolver-se de forma simultânea. Assim, pode-se dizer que qualquer recurso disponível em um sistema torna-se escasso quando em quantidade inferior ao número de processos que em um dado instante solicitarem acesso ao mesmo.

Na ausência de uma política de moderação, tal concorrência tem, inevitavelmente, conseqüências caóticas. Desta forma, torna-se necessário buscar um meio que permita disciplinar de forma eficaz a disputa pelos recursos escassos.

Um algoritmo de escalonamento é um conjunto de regras que uma vez detectada a existência de concorrência, define o modo de partilhar um recurso ou um conjunto destes pelas diversas entidades que pretendem usufruí-lo, possibilitando que este intento seja alcançado [34].

Em sistemas computacionais, os processadores não são os únicos recursos compartilhados. Largura de banda de um barramento, interfaces de rede, entre outros componentes, figuram como recursos escassos. No escopo deste trabalho, entretanto, o recurso que está sendo considerado é apenas o sistema de comunicação, e as tarefas são as mensagens que trafegam por meio deste. Pretende-se, então, apresentar algoritmos clássicos de escalonamento de mensagens em tempo real.

2.4 Escalonamento de Tarefas de Tempo Real

As necessidades temporais condicionais inerentes aos processos de tempo real são as principais responsáveis pela dificuldade no desenvolvimento de algoritmos de escalonamento de um recurso. Abordagens de escalonamento consideradas justas em um determinado contexto podem não ser eficazes quando analisadas à luz das necessidades temporais expressas em tarefas de tempo real [19].

Um algoritmo de escalonamento para tarefas de tempo real crítico somente se torna válido se garantir, dentro de alguns pressupostos, que todas as tarefas serão executadas e concluídas dentro de seus respectivos prazos.

2.4.1 Abordagens de Escalonamento

Em Farines *et al.* [19], é demonstrado que, levando-se em consideração o tipo de carga a ser tratada e as etapas de escalonamento, pode ser definida uma taxonomia identificando a existência de três grupos principais de abordagens de escalonamento de tempo real: as abordagens com garantia em tempo de projeto, as com garantia em tempo de execução e

as baseadas na estratégia de melhor esforço [43]. A figura 2.2 representa esta classificação das diferentes abordagens de escalonamento de tempo real.

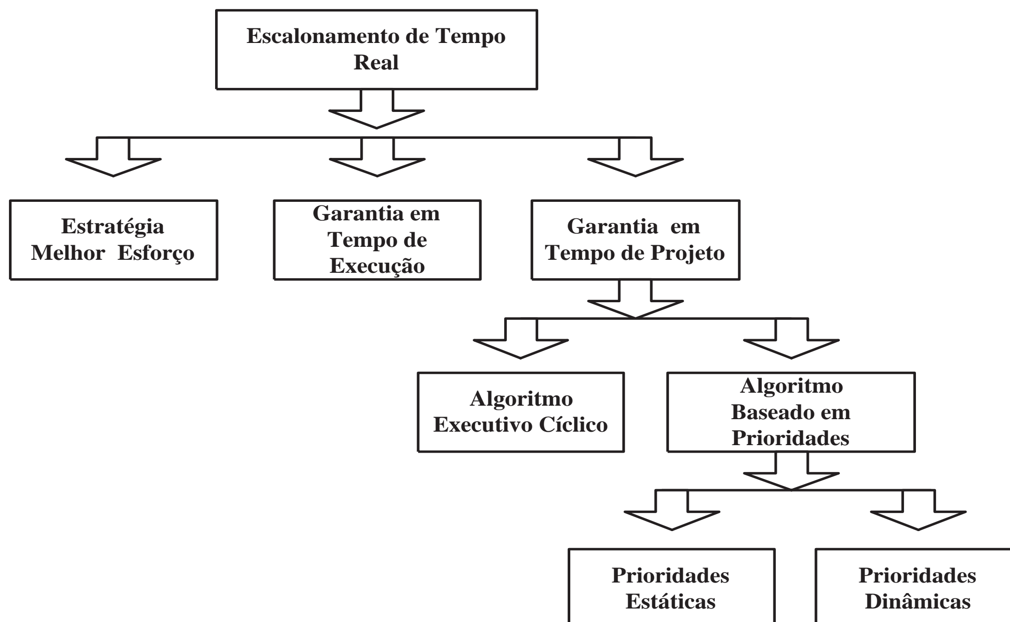

Figura 2.2: Taxonomia de abordagens de escalonamento de tempo real

2.4.1.1 Abordagens com Garantias em Tempo de Projeto (*Off-line guarantee*)

Este grupo de garantias tem por objetivo a previsibilidade determinista, que somente é alcançada a partir de um conjunto de premissas:

- A carga é estática, sendo conhecida em tempo de projeto, bem como os instantes de chegada das tarefas. Sua modelagem é feita em função de tarefas periódicas e esporádicas;

- Os recursos são suficientes para o cumprimento das tarefas, respeitando suas restrições temporais em condição de carga máxima (pior caso).

Existem dois tipos de abordagens com garantia em tempo de projeto: o executivo cíclico e o escalonamento dirigido a prioridades.

No caso **Executivo Cíclico**, tanto o teste de escalonabilidade quanto à produção da escala são realizados em tempo de projeto. Esta abordagem gera a sua escala visando

refletir o pior caso (pior situação de chegada de tarefas, piores tempos de execução e tarefas esporádicas com a máxima ocorrência).

Apesar de o sistema garantir a execução das tarefas com base nas suas restrições temporais, desperdiça recursos que são sempre reservados para o pior caso, uma vez que este está distante do caso médio [19].

O **Escalonamento Dirigido a Prioridade** mostra-se mais dinâmico, uma vez que apenas o teste de escalonabilidade é realizado durante o tempo de projeto, sendo a escala produzida em tempo de execução por um escalonador dirigido a prioridades. Em virtude do pior caso ser considerado apenas durante o teste de escalonabilidade, esta abordagem é mais flexível, mas ainda assim garante recursos para o pior caso.

2.4.1.2 Abordagens com Garantia Dinâmica e de Melhor Esforço

As abordagens com garantia dinâmica e de melhor esforço, ao contrário do grupo anterior, tratam a carga computacional como sendo dinâmica, e, portanto, não previsível [19].

Os tempos de chegada das tarefas não são conhecidos previamente. Uma vez que o pior caso não pode ser antecipado em tempo de projeto, não é possível que sejam previstos os recursos necessários para todas as situações de carga. Assim, não existe como garantir antecipadamente que todas as tarefas, independente da situação de carga, irão ter seus prazos respeitados.

Nos casos em que se trata com carga dinâmica podem ocorrer situações onde os recursos computacionais são insuficientes para os cenários de tarefas (situações de sobrecarga). Por isso, tanto os testes de escalonabilidade, quanto a escala devem ser realizadas em tempo de execução.

As abordagens com garantia dinâmica utilizam um teste de aceitação para verificar a escalonabilidade do conjunto de tarefas a serem executadas, realizado cada vez que uma nova tarefa chega ao grupo (fila de prontos). Tais abordagens baseiam-se em análises realizadas com hipóteses de pior caso sobre alguns parâmetros temporais. Se o seu resultado indicar o conjunto como não escalonável, a nova tarefa que chegou será então descartada preservando o conjunto de tarefas que, previamente, foi garantido como escalonável [19].

Estas classes de abordagem que oferecem garantias dinâmicas são próprias para aplicações que possuem restrições críticas, mas que não operam em ambientes determi-

nistas.

As abordagens baseadas no melhor esforço são fundamentadas nos algoritmos que tentam encontrar uma escala que possa suprir as necessidades em tempo de execução, sem realizar teste ou realizando testes mais fracos.

Não existe a garantia do envio de tarefas atendendo suas restrições temporais. Essas abordagens são adequadas para aplicações não críticas, envolvendo prazos que não sejam rígidos, onde a perda destes prazos não representa custos além da diminuição do desempenho das aplicações, como é o caso de sistemas multimídia.

O desempenho de sistemas baseados no melhor esforço quando considerados apenas os casos médios, é muito melhor que os baseados em garantia dinâmica. As hipóteses pessimistas feitas em abordagens com garantia dinâmica podem desnecessariamente descartar tarefas. Nas abordagens de melhor esforço tarefas são abortadas somente em condições reais de sobrecarga, quando ocorrem falhas temporais.

2.5 Escalonamento de Tarefas Periódicas

Nas aplicações de tempo real, as atividades envolvidas apresentam como característica o comportamento periódico de suas ações. Em um conjunto de tarefas periódicas, as características que determinam a priori o conhecimento dos tempos de chegada e da carga computacional do sistema permitem a obtenção de garantias, em tempo de projeto, a respeito da escalonabilidade.

Em esquemas de escalonamento dirigidos a prioridades, as tarefas do conjunto têm suas prioridades derivadas de suas restrições temporais. Escalonamentos deste tipo apresentam melhor desempenho e flexibilidade, em comparação com abordagens como o executivo cíclico.

Pretende-se, aqui, apresentar os algoritmos clássicos de prioridade fixa ("*Rate Monotonic*", "*Deadline Monotonic*" e "*Earliest Deadline First*"), tidos como ótimos para suas respectivas classes de problemas [19], e que são caracterizados por modelos de tarefas simplificados.

2.5.1 RMA (*Rate Monotonic Algorithm*)

O algoritmo de escalonamento RM (*Rate Monotonic*) foi uma das primeiras técnicas de análise de escalonabilidade apresentada em [34]. A principal regra que o caracteriza é a atribuição de prioridades às tarefas periódicas de forma inversamente proporcional a seus períodos.

A análise de escalonabilidade no RM é realizada em tempo de projeto e baseia-se no cálculo de utilização. Nestas condições, um sistema é dito escalonável se cumprir a seguinte condição (2.1):

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.1}$$

Onde *N* é o número total de processos no sistema, *Ci* e *Ti* o pior tempo de computação e a taxa de chegada do processo *i*, respectivamente.

O RM utiliza-se de um esquema de prioridades fixas, atribuídas antes da execução, não mudando no tempo (escalonamento estático). O escalonador utilizado na execução da escala produzida é preemptivo.

O RM é dito ótimo entre os algoritmos que utilizam prioridade fixa, pois nenhum outro pode escalonar um conjunto de tarefas que não pode ser escalonado pelo RM.

O modelo de tarefas é estabelecido com base em algumas premissas [19]:

- As tarefas são periódicas, não existindo dependência entre estas.
- O deadline de cada tarefa é igual ao seu período (*Di* = *Pi*).
- Cada tarefa necessita do mesmo tempo de computação (*Ci*) a cada período.
- O tempo de chaveamento entre tarefas não é considerado.

Neste caso, a utilização alcançada se aproxima do máximo teórico, coincidindo o teste abaixo com uma condição necessária e suficiente [34]:

$$U = \sum Ci/Pi \leq 1 \tag{2.2}$$

sendo U a utilização do recurso, Ci o tempo de computação de cada tarefa, e Pi o período.

O RM garante a escalonabilidade de um conjunto de tarefas periódicas quando a utilização máxima por todos os processos a um recurso seja inferior a 69,3%. Neste patamar todo o conjunto de tarefas é escalonável.

2.5.2 EDF (*Earliest Deadline First*)

O EDF (*Earliest Deadline First*) consiste na análise de cada processo de acordo com seu prazo máximo de execução, tornando-se assim dinâmico [16]. Neste algoritmo a escala de execução é produzida por um escalonador preemptivo dirigido a prioridades, durante a execução do sistema.

Diferente do RM, o EDF atribui maior prioridade à tarefa com prazo mais perto de findar. O processo de escalonamento é simples. Todo o processo deve informar sua presença e seu prazo máximo, sendo tais informações adicionadas a uma lista, ordenada de maneira que as tarefas com os prazos mais próximos do vencimento tenham prioridade mais elevada.

O modelo de tarefas possui premissas idênticas as utilizadas na abordagem RM, conforme visto na seção 2.5.1.

No EDF, toda a vez que uma tarefa chegar no sistema, a fila de tarefas prontas pode sofrer uma reordenação. Isto ocorre em virtude da nova distribuição das prioridades em relação as seus prazos. A cada ativação de uma *Ti*, um novo valor de *deadline* absoluto é determinado.

O teste de escalonabilidade no EDF é feito na fase do projeto e a base é a utilização do processador (recurso compartilhado). Um conjunto de tarefas é dito escalonável baseando-se na equação 2.3:

$$U = \sum Ci/Pi \leq 1 \tag{2.3}$$

sendo U a utilização do recurso, Ci o tempo de computação de cada tarefa, e Pi o período.

2.5.3 DM (*Deadline Monotonic*)

O algoritmo DM, introduzido em Leung e Whitehead [32], atribui de forma fixa maior prioridade às tarefas periódicas, cujos prazos são considerados sempre menores ou iguais

aos períodos das mesmas. O RMA é um caso particular deste algoritmo [4].

Na abordagem utilizada pelos algoritmos RM e EDF, onde o prazo é igual ao período de uma tarefa, é possível que uma instância de determinada tarefa possa ser executada em qualquer instante dentro de seu período. Contudo, isto pode nem sempre ser desejado por ser demasiadamente restritivo, quando se trata de aplicações de tempo real.

O modelo de tarefas do DM segue as seguintes premissas [19]:

- Tarefas periódicas e independentes entre si;
- Tempo de processamento (pior caso);
- Prazos Relativos menores ou iguais aos períodos das tarefas;
- Tempo de chaveamento entre tarefas não é considerado.

O DM considera os prazos relativos menores ou iguais aos períodos das tarefas, desta forma as prioridades são atribuídas estaticamente, na ordem inversa aos valores de seus prazos relativos. Sendo assim, em qualquer instante, a tarefa com o menor prazo relativo é executada.

O escalonador utilizado no DM é um escalonador preemptivo que se baseia em prioridades, e a escala utilizada pelo mesmo é criada em tempo de execução.

2.6 Análise de Escalonabilidade

Em se tratando de sistemas de tempo real crítico, torna-se necessário provar, de modo formal, que uma dada configuração cumpre sempre suas metas. Contudo, esta tarefa está muito longe de ser algo trivial devido ao grande número de possibilidades existentes, mesmo em sistemas simples.

Ao ato de verificar se uma dada configuração das várias tarefas que compõem um sistema, com uma certa ordem de escalonamento, cumpre ou não as metas dá-se o nome de teste de escalonabilidade.

2.6.1 Testes de Escalonabilidade

No processo de escalonamento de tarefas de tempo real, a realização de testes de escalabilidade tem uma grande importância, uma vez que através destes é possível que se determine se um conjunto de tarefas é escalonável, ou seja, se existe uma escala realizável para este conjunto.

A forma de realização de tais testes varia de acordo com os modelos de tarefas e políticas definidas em um problema de escalonamento. Para isto é necessário que sejam definidos parâmetros que reflitam variáveis, como o nível de ocupação de determinado recurso ou mesmo o tempo de resposta do sistema.

Segundo Farines *et al.*[19], os testes de escalonabilidade são classificados em: Testes Suficientes, Testes Exatos e Testes Necessários.

A figura 2.3 representa a maneira como estão divididos os testes de escalonabilidade.

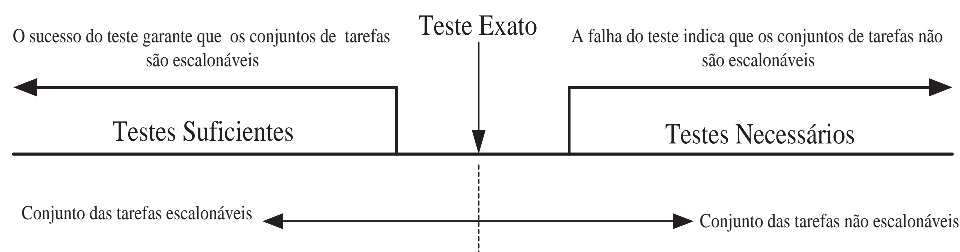

Figura 2.3: Testes de Escalonabilidade [19]

Testes Suficientes: Conjunto de testes capaz de apontar se um conjunto de tarefas é escalonável. A aplicação destes testes é de relativa facilidade, mas é necessário que se leve em conta o fato de que entre os conjuntos de tarefas descartados podem existir alguns grupos de tarefas escalonáveis. Todo o conjunto de tarefas aceitas por estes testes são escalonáveis de fato.

Testes exatos: São testes capazes de apontar com precisão se um conjunto de tarefas é ou não escalonável, contudo apesar de tal precisão este tipo de testes são impraticáveis [19].

Testes necessários: conjunto de testes menos restritivo baseando-se em uma análise simples do conjunto de tarefas existentes no sistema. Os conjuntos de tarefas descartados por esta classe de testes não são escalonáveis, contudo, os conjuntos aceitos não podem

ser garantidos como escalonáveis[19].

2.7 Inversão de Prioridades

Em sistemas multitarefa com escalonamento por preempção e mecanismos de controle de acesso a estruturas compartilhadas é possível que uma tarefa fique bloqueada por uma outra tarefa de menor prioridade que detenha o direito de acesso a uma estrutura compartilhada. Entretanto, se uma outra tarefa com prioridade intermediária interrompe a tarefa de menor prioridade e passa a executar continuando a tarefa de maior prioridade bloqueada, a este problema dá-se o nome de inversão de prioridade [26]. Quando isso não é devidamente acautelado pode dar origem a cenários de bloqueio circular por parte de várias tarefas.

Em Sha *et al.* [45] é apresentada a descrição de dois algoritmos que visam limitar o tempo pelo qual um processo pode experimentar uma situação de inversão de prioridade.

2.7.1 Protocolo de Herança de Prioridade (*Priority Inheritance Protocol*)

O protocolo PIP tem por objetivo limitar o tempo pelo qual um processo pode estar sujeito a um regime de inversão de prioridade, através da elevação da prioridade do processo que a motiva.

Quando um processo *Pi* bloqueia a execução de um processo *Pj* de prioridade superior à sua *Pi* passa a ser executado com a prioridade de *Pj*.

Utilizando-se o protocolo PIP em um processo genérico *Pi*, composto por *m* seções críticas, a execução de *Pi* pode vir a ser bloqueada *m* vezes. No pior dos casos, um processo *Pi* pode ser bloqueado sempre que tentar entrar em uma seção crítica.

Este protocolo impõe desta forma um limite de tempo pelo qual um processo pode sofrer uma inversão de prioridade, o que não garante, entretanto, que este limite não seja bastante elevado. Além disso, este protocolo não tem a possibilidade de prevenir situações de impasse (*deadlock*).

2.7.2 Protocolo de Prioridade Teto (*Priority Ceiling Protocol*)

Na abordagem do protocolo de prioridade teto (PCP) a idéia central é limitar o problema de inversão de prioridade e evitar a formação de situações de impasse e cadeias de bloqueios. O PCP é dirigido para escalonamento de prioridade fixa.

Este protocolo é uma extensão do PIP, adicionando uma regra de controle sobre os pedidos de entrada em exclusão mútua. Para evitar o bloqueio múltiplo, a regra não permite que uma tarefa entre em uma seção crítica, caso existam semáforos bloqueados que possam bloqueá-la. Uma vez que a tarefa entra na sua primeira seção crítica, ela não pode ser bloqueada por tarefas de prioridade mais baixas.

Na prática, cada semáforo tem uma prioridade teto associada que é igual à prioridade da tarefa de maior prioridade que pode pegar o semáforo. Assim, uma tarefa T tem permissão de entrar em uma seção crítica, somente se sua prioridade for maior do que todas as prioridades teto dos semáforos pertencentes a outras tarefas diferentes de T. Ao sair da seção crítica, a tarefa passa a executar com a sua prioridade estática.

No capitulo seguinte, serão apresentados conceitos básicos sobre o modelo de referência OSI, onde serão abordados aspectos relativos às comunicações em tempo real, concentrando-se na camada de enlace e na subcamada MAC.

Capítulo 3

Modelo de Referência OSI e as Comunicações em Tempo Real

3.1 Modelo de Referência OSI (*RM OSI*)

No final de 1979, a ISO (*International Standard Organization*) convergiu para a definição de um modelo de sistemas de comunicação estruturado em camadas, para especificar a comunicação entre sistemas abertos. Após alguns estágios intermediários, em 1983 foi aprovado o documento definitivo ISO/EIC 7894. Este modelo de referência é até nossos dias utilizado para reger o desenvolvimento de arquiteturas de comunicações entre sistemas heterogêneos.

O RM OSI foi desenvolvido com a finalidade de identificar e estabelecer uma taxonomia das diferentes funções dos sistemas de comunicação, sendo apenas utilizado como uma referência para o desenvolvimento de sistemas de comunicação.

Este modelo define sete camadas ou funções, não sendo obrigatório que um determinado sistema implemente todas as camadas de forma distinta.

A figura 3.1 mostra como se relacionam as camadas e como elas tratam as informações neste modelo.

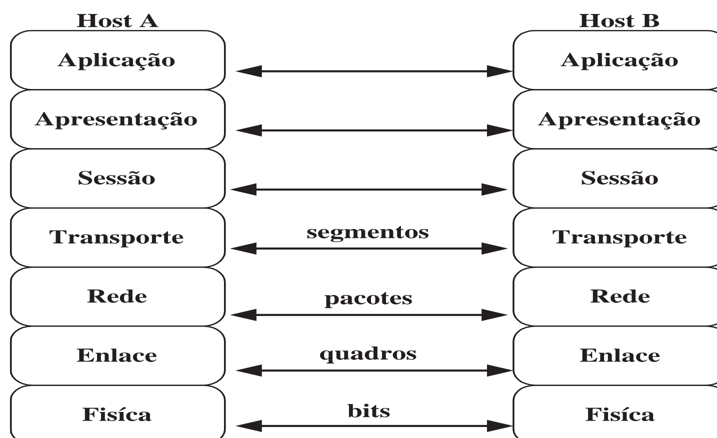

Figura 3.1: As sete camadas do Modelo de Referência OSI

3.2 As camadas do modelo OSI

As camadas que compõem o Modelo de Referência OSI são especificadas na norma ISO/EIC 7894 [24]. Pretende-se, aqui, apresentar uma breve descrição de cada uma destas camadas e suas principais funcionalidades.

Camada Física (Camada 1): A camada física define as especificações elétricas, mecânicas, funcionais e de procedimentos para ativar, manter e desativar o link físico entre sistemas finais. São definidas nesta camada características tais como, níveis de voltagem, temporização de alterações de voltagem, taxas de dados físicos, distâncias máximas de transmissão, conectores físicos e outros atributos similares.

Camada de Enlace (Camada 2): Esta camada define o formato dos quadros, a sua codificação lógica, fornece trânsito seguro de dados através de um link físico. Além disso, é nesta camada que são tratados o endereçamento físico (em oposição ao endereçamento lógico), a topologia de rede, o acesso à rede, a notificação de erro, a entrega ordenada de quadros e o controle de fluxo. Em se tratando de redes locais, a camada de enlace é subdividida em duas camadas: A subcamada MAC (*Medium Access Control*) e subcamada LLC (*Logical Link Control*)[25].

A subcamada MAC que tem como objetivo proporcionar mecanismos de controle de acesso ao meio físico de transmissão.

A subcamada LLC localiza-se na parte superior da camada de enlace, podendo prestar serviços como: serviço de datagrama não-confiável, serviço de datagrama com

confirmação e serviço orientado à conexão confiável.

Camada de Rede (Camada 3): É neste nível que são descritas as características da rede, de forma a garantir que os pacotes cheguem ao seu destino através do caminho escolhido na fonte. Este nível é imprescindível quando diferentes caminhos são possíveis, pois é neste ponto que se faz o chamado encaminhamento de pacotes.

Camada de Transporte (Camada 4): A camada de transporte recebe os dados da camada de seção e os divide, se necessário, em unidades menores e os passa para a camada de rede garantindo que todas essas unidades cheguem corretamente à outra máquina.

Nesta camada, define-se a maneira como serão estabelecidas as ligações, bem como será realizada a comunicação entre dois nós da rede. A camada de transporte estabelece, mantém e termina corretamente circuitos virtuais. Pode fornecer um serviço confiável, por utilizar o controle do fluxo de informações e a detecção e recuperação de erros de transporte.

Camada de Sessão (Camada 5): Após o estabelecimento lógico de um canal de comunicação entre dois pontos feita pela camada de transporte, a camada de sessão fornece uma metodologia para controle do diálogo, sendo responsável por sincronizar as aplicações e iniciar uma sessão de trabalho.

Camada de Apresentação (Camada 6): Esta camada realiza transformações necessárias aos dados, antes do envio à camada de sessão. Transformações típicas dizem respeito à compreensão de texto, criptografia, conversão de padrões de terminais e arquivos para padrões de rede.

Camada de Aplicação (Camada 7): Esta é a camada mais próxima do usuário, fornecendo serviços de rede aos aplicativos do usuário. Ela não fornece serviços a nenhuma outra camada, mas apenas a aplicativos fora do modelo OSI.

O modelo em camadas do Protocolo OSI, apesar de ser algo conceitual, representa de um modo geral a forma como os sistemas são implementados na maneira real. Cada camada traduz uma interface entre as duas camadas contíguas utilizando um protocolo baseado em par. Isto significa que, ao longo das camadas, o *jitter*[1] de latência introduzido será grande e a eficiência irá se tornar reduzida.

Deste modo, implementações de protocolos baseadas em muitas camadas podem tor-

[1]Jitter: é a inconstância na razão do atraso (Variação no Atraso)

nar o tamanho do *jitter* abusivo para sistemas com restrições temporais. Assim, estes protocolos não são aplicáveis a sistemas de tempo real [20]. Visando garantir uma melhor performance temporal, na maioria dos sistemas de tempo real utiliza-se uma arquitetura de softwares de comunicação baseada em apenas três camadas: camada física, camada de enlace (subdividida em LLC e MAC) e camada de aplicação.

No contexto deste trabalho serão analisados apenas protocolos da camada de enlace e na subcamada MAC do RM OSI, onde esta enquadrado o projeto.

3.3 Classificação dos Protocolos de Controle de Acesso ao Meio (MAC)

Os protocolos de controle de acesso ao meio podem ser classificados, tanto à luz da forma como alocam o meio de transmissão, quanto em função de seus respectivos comportamentos temporais.

3.3.1 Classificação dos Protocolos MAC quanto à Alocação do Meio

Em Stemmer [52] encontra-se uma classificação dos protocolos de acesso ao meio, em 5 categorias;

Protocolos de Alocação Fixa: Os protocolos desta categoria alocam o meio aos nós por intervalos de tempos determinados, independentemente da real necessidade de acesso.

Protocolos de Alocação Controlada: Neste tipo de protocolo, um determinado nó (estação) apenas recebe o direito de acessar o meio quando toma posse de um quadro de permissão. Este quadro é entregue aos nós através de uma ordem predefinida.

Protocolos de Alocação por Reserva: Esta classe de protocolo caracteriza-se pelo fato de os nós necessitarem reservar banda com antecedência, através de uma solicitação feita a estação controladora, dentro de um intervalo de tempo anteriormente reservado para esta finalidade.

Protocolos de Alocação Aleatória (Protocolos de contenção): Estes protocolos permitem que o acesso ao meio, por parte dos nós, ocorra de forma aleatória, mas respeitando um conjunto de regras.

Protocolos Híbridos: São ditos híbridos, todos aqueles protocolos que se enquadram em duas ou mais das categorias anteriores.

3.3.2 Classificação dos Protocolos MAC quanto ao comportamento temporal

Os protocolos de acesso ao meio podem ser classificados, basicamente, de duas formas quanto ao seu comportamento temporal:

Protocolos Determínisticos: Esta categoria de protocolo é caracterizada pela sua possibilidade de definir um prazo limite para a entrega de uma mensagem especifica, mesmo que isto ocorra no pior caso. Alguns destes protocolos concedem acesso ao nó independente da necessidade deste utilizar o seu intervalo de transmissão.

Protocolos Não Determínisticos: Esta classe de protocolos é, freqüentemente, caracterizada pela competição entre estações pelo direito de acessar o meio de transmissão.

3.3.2.1 Protocolos MAC Não Determínisticos

Dentre os protocolos MAC não determinísticos, pretende-se apresentar, nesta seção, o CSMA e suas variações.

O protocolo **CSMA** ("*Carrier Sense Multiple Access*") permite a partilha do meio de transmissão, baseando-se na detecção da existência de uma transmissão em curso ("*Carrier Sense*").

Quando um nó pretende enviar dados, primeiro será verificado se o meio de transmissão está livre para que a transmissão possa ser iniciada. Contudo, se o meio de transmissão estiver ocupado existem várias abordagens baseadas neste algoritmo para a resolução desta situação.

CSMA não persistente: Nesta abordagem, o comportamento da estação que deseja transmitir é menos afoito que nas demais abordagens, pois, se o meio de transmissão estiver ocupado, a estação irá esperar um período de tempo aleatório, para voltar a fazer nova tentativa de transmissão [54].

Como principal desvantagem, esta abordagem apresenta o fato de que o meio de transmissão pode vir a ser desocupado neste meio tempo, enquanto existem dados para ser

transmitidos. Isto introduz um maior atraso de emissão ao nível das estações em relação aos protocolos persistentes [48].

CSMA 1 persistente: Nos protocolos baseados nesta abordagem, quando uma estação estiver pronta para enviar dados, esta deve primeiro escutar o meio de transmissão. Estando este livre a estação passará, a enviar suas mensagens. Caso contrário, a estação deve aguardar na escuta até que o meio esteja livre para transmissão [48].

No caso de mais de uma estação, ao escutar o meio de acesso, detectarem-no como disponível para transmissão, ambas passarão a transmitir. Nestas condições ocorre uma colisão, e os nós envolvidos devem esperar um período de tempo aleatório para que voltem a escutar o barramento novamente, e possam fazer uma nova tentativa de transmissão.

Esta abordagem sofre influência do tempo de propagação dos pacotes, pois quanto maior este tempo pior o desempenho do protocolo, em virtude da ocorrência de colisões. O tempo de propagação depende principalmente da taxa de transmissão(bits/s) e do comprimento do meio de transmissão (cabo) [54].

CSMA p persistente: Este algoritmo tenta diminuir as colisões evitando que o meio de transmissão seja subutilizado. Nesta abordagem, se uma estação possui uma mensagem para enviar, primeiro irá escutar o meio de transmissão, verificando a disponibilidade do mesmo e, só então, passando a transmitir com uma probabilidade p. Porém, se o meio de transmissão estiver ocupado será necessário que a estação aguarde um período de tempo, equivalente ao atraso máximo de propagação no meio de transmissão para voltar a tentar a transmitir. Este tempo é especifico e será sempre igual a q=1-p, onde q é o intervalo de tempo e p a probabilidade.

Mesmo a técnica p-persistente não resolve totalmente os problemas, por um lado continuam a existir colisões e por outro continuam a existir instantes em que o meio não é utilizado e existem dados à espera para ser emitidos [54].

CSMA/CD (*Collision Detection*): O método de acesso CSMA/CD foi especificado pela IEEE e, posteriormente, pela ISO, através de normas IEEE 802.3 e ISO 8802-3 [22]. Tipicamente, a técnica CSMA/CD utiliza um algoritmo 1-persistente, que é o mais eficiente sob o ponto de vista da utilização do meio de transmissão. Em lugar de minimizar o número de colisões, este tipo de protocolo tenta-se reduzir as suas conseqüências [49].

Nas implementações CSMA anteriores, quando ocorre uma colisão, o meio de trans-

missão fica inutilizado por um período de tempo igual ao tempo de transmissão de cada um dos quadros.

O mecanismo CD obriga que os nós escutem a rede enquanto emitem dados, razão pela qual o CSMA/CD é também conhecido por "*Listen While Talk*" (LWT) [49].

Como o nó emissor também escuta a rede ele pode detectar a colisão. Nesse caso, a emissão do pacote é cessada imediatamente, sendo emitido um sinal ("*jam*") de 48 bits que notifica todas as estações da ocorrência da colisão. Neste caso, o nó espera um período de tempo aleatório e volta a tentar a fazer a transmissão. Para evitar colisões sucessivas, utiliza-se uma técnica conhecida por "*binary exponential backoff*" em que os tempos aleatórios de espera são sempre duplicados por cada colisão que ocorre [54].

Existe um aspecto importante a considerar para que as colisões sejam detectadas com sucesso, o tamanho mínimo dos pacotes deve ser tal que o seu tempo de transmissão seja superior ao dobro do atraso de propagação. Se isto não acontecer, uma estação pode completar a emissão do pacote, sem que o sinal produzido pela colisão chegue a tempo [54].

As redes baseadas nos métodos de acesso CSMA possuem, como característica, o fato de que quanto maior o número de estações e mais elevado for o trafego na rede, a probabilidade da ocorrência de colisões aumenta.

Os protocolos baseados nas filosofias CSMA são altamente não determínisticos, não sendo possível prever com exatidão o seu comportamento. Esta característica os torna não interessantes para redes que trabalhem com restrições temporais rígidas.

3.3.2.2 Protocolos MAC determínisticos

Os protocolos desta classe têm como característica o fato de possuírem um tempo de resposta que pode ser determinado, mesmo no pior caso. Os protocolos MAC determínisticos podem ser classificados de duas formas: os métodos com comando centralizado e os de comando distribuído.

O protocolo **Mestre-Escravo**, apresenta dois conceitos importantes que são relativos à estação denominada mestre e à denominada escrava. Em qualquer interligação na qual esteja sendo utilizado um protocolo baseado nesta técnica, necessariamente deverá existir uma estação mestre e pelo menos uma estação escrava.

Define-se como mestre, a estação responsável por toda a iniciativa do processo de comunicação. A troca de dados entre as estações, em qualquer dos dois sentidos, é iniciado e finalizado pelo mestre. A estação mestre realiza uma varredura cíclica de cada uma das estações escravas, solicitando dados ou verificando se elas possuem dados a serem enviados. A estação escrava é definida como aquela que, no processo de comunicação, só realiza alguma função sob requisição e controle do mestre.

Na comunicação entre mestre e escravo existem duas situações possíveis:

- mestre deseja enviar/receber dados para/do escravo.
- escravo deseja enviar/receber dados para/do mestre.

No primeiro caso, em virtude de o mestre possuir o poder para iniciar o processo de comunicação, pode fazer isso a qualquer instante, enviando uma mensagem requisitando ao escravo a realização de uma determinada função. Isto pode ser feito de acordo com suas necessidades e independente do estado do escravo. Ao receber a mensagem enviada pelo mestre, o escravo executa a função requisitada e envia uma resposta contendo o resultado desta.

No segundo caso, como o escravo não pode tomar a iniciativa de começar o processo de comunicação, ele deve aguardar até que o mestre lhe pergunte se ele deseja enviar/receber alguma mensagem, e somente quando isto ocorrer, o escravo envia sua mensagem requisitando ao mestre a realização de determinada função. Ao receber a mensagem enviada pelo escravo, o mestre realiza a função solicitada e envia uma resposta contendo seu resultado. Esse processo recebe o nome de "*polling*".

Este tipo de configuração deixa o sistema totalmente dependente desta estação central. Esta é uma das mais costumeiras configurações utilizadas quando se trata de sistemas de controles [52].

Os **Métodos de Acesso Token-Passing** são aqueles onde existe a possibilidade da definição de mais de uma estação com o direito de acesso ao meio físico. Nesta técnica, um padrão especial de bits, denominado *token*, circula entre as estações da rede. A estação que obtiver este pacote é a que possui o direito para efetivar suas transmissões.

Nos protocolos baseados nesta filosofia, é necessário assegurar que apenas uma estação possua o *token* em cada instante, e que, por outro lado, o *token* circule de forma cíclica

entre todas as estações afim de que todas possam transmitir seus dados. Com o uso desta técnica, as diversas estações que compõem um sistema podem trocar dados livremente entre si sem a necessidade da intromissão de uma estação concentradora ou outro intermediário qualquer.

Nestas abordagens, cada nó possui um número de ordem, o *token* circula de nó em nó segundo esta ordem, o nó de número de ordem mais elevado envia o *token* para o de ordem mais baixa. Devido a esta circulação do *token*, em termos lógicos, os nós (estações) estão sempre dispostos em anel, embora a rede física possa ter uma topologia em formato de árvore ou barramento [48].

As duas configurações mais usuais deste método de acesso são as redes baseadas no padrões *Token Ring* e *Token Bus*.

O método de acesso ***Token Ring*** foi especificado pela IEEE e, posteriormente, pela ISO, através de normas IEEE 802.5 e ISO 8802-5 [23]. Sua topologia física é em formato de um anel. Um anel não é, de fato, um meio de difusão, mas um conjunto de ligações ou interfaces ponto a ponto individuais que formam um círculo.

O método para deter o acesso ao meio de transmissão é a obtenção de um quadro especial de bits, denominado *token*. O quadro de permissão circula em torno do anel sempre que todas as estações estão ociosas.

Quando uma determinada estação deseja transmitir uma mensagem, ela executa os seguintes passos:

- Remove o *token* do anel;
- Executa a transmissão de sua mensagem;
- Conforme os bits mensagem vão retornando, após terem dado a volta, a estação deve retirá-los do anel;
- Recoloca o *token* no anel com a prioridade adequada.

Como somente a estação que possui o *token* tem permissão de executar a transmissão, não existem colisões nesse tipo de rede.

Também importante, nesse tipo de rede, é a existência de prioridades que servem para permitir que aplicações que necessitam uma transmissão mais freqüente (como o controle

de uma linha de montagem) sejam executadas antes de outras que podem aguardar para serem atendidas (como edição de texto).

O método de acesso ***Token Bus*** é especificado através das normas técnicas ISO 8802-4 e IEEE 802.4 [21] e foi projetado com adoção da relação de prioridade na transmissão dos *frames* (quadros) e do estabelecimento de um pior caso conhecido no tempo de transmissão dos quadros.

Fisicamente, o *token bus* é um cabo em forma de uma árvore ou um barramento linear no qual as estações são conectadas. As estações estão organizadas logicamente em um anel virtual que define a ordem na qual os quadros serão transmitidos, pois cada estação que compõem este anel tem conhecimento próprio e das estações com endereço anterior e posterior ao seu.

Uma vez que o anel tenha sido inicializado, a estação de maior endereço envia seus quadros e, ao término desta atividade envia um quadro de controle para seu sucessor. Ao adquirir o *token*, a estação pode transmitir seus quadros por um certo período de tempo, ao final do qual deve liberar e repassar o *token*.

Na ausência de dados para enviar, uma estação que recebe o *token* passa-o imediatamente para a próxima estação com prioridade inferior. O *token* viaja pelo anel, dando permissão somente às estações que o possuem para enviar quadros. Sendo assim, as colisões deixam de existir.

Outro padrão de redes que pode ser destacado é o **100 VG-AnyLAN** [48], que se constitui uma tecnologia de rede, que provê uma taxa de dados de 100 Mbit/s usando um método de acesso de controle centralizado, demoninado "*Demand Priority*". Este método de acesso, é um método de requisição simples e determinístico que maximiza a eficiência da rede pela eliminação das colisões que ocorrem no método CSMA/CD.

O protocolo "Demand Priority" baseia-se um HUBS inteligentes que recebem dos nós, ligados em estrela, pedidos para acesso ao meio de transmissão, podendo indicar a prioridade do pedido. Este método evita as colisões do CSMA/CD e evita o tempo de circulação do token.

O *HUB* 100VG tem a missão de coordenar todo o acesso ao meio, gerindo a lista de pedidos pendentes. Existem pedidos de prioridade normal e prioridade elevada. Os pedidos de prioridade normal são atendidos porta a porta, mas quando chegam pedidos

de prioridade elevada estes serão atendidos imediatamente. Para evitar que os pedidos de prioridade normal fiquem retidos, é definido um tempo de residência máximo, depois deste ser ultrapassado estes pedidos passam a prioridade elevada.

Outra vantagem deste padrão é oferecer compatibilidade com as redes *Ethernet*) ou (*Token-Ring*).

Além deste protocolos vistos anteriormente, destacam-se ainda, alguns protocolos desenvolvidos através de pesquisas realizadas no intuito de definir protocolos determínisticos baseados no CSMA. Tais pesquisas surgem em virtude de que este modelo prove um grau de liberdade bastante desejável, propiciando autonomia aos nós da rede. O objetivo básico reside na resolução dos problemas decorrentes das colisões.

Dentre as abordagens baseadas no protocolo CSMA, pode-se citar a dos "Cabeçalhos Forçantes", a de "Comprimento de Preâmbulo" e o "CSMA/DCR".

Em Stemmer [52] encontra-se uma descrição do método de acesso dos **Cabeçalhos Forçantes** (*Forcing Headers*), mostrando seu funcionamento e a maneira como trata a ocorrência de colisões de forma determínistica.

Neste método, cada mensagem é iniciada por um cabeçalho, composto por uma serie de bits, que definem sua prioridade, sendo vetada a existência de duas mensagens com prioridades idênticas em uma mesma aplicação. O inicio da transmissão ocorre com o envio dos cabeçalhos, que são enviados bit a bit, em baixa velocidade. Após o envio de cada bit, o nível de sinal do barramento é lido.

Os bits são codificados de forma que uma colisão tenha o efeito de uma operação lógica AND sobre os bits enviados ao barramento. A transmissão em uma estação é interrompida no momento em que esta verifica que o barramento, ao receber um bit com o valor 1, retorna um bit com valor 0. Se o cabeçalho for transmitido até o fim sem nenhuma colisão é porque a mensagem é a prioritária dentre as envolvidas na colisão. Sendo assim a mensagem mais prioritária é enviada.

O método do **Comprimento do Preâmbulo** (*Preamble Length*) possui como característica o fato de que a cada mensagem é associada a um preâmbulo com comprimento diferente, sendo este transmitido com a detecção de colisão desativada. Logo após a transmissão do preâmbulo da mensagem, a estação passa a reativar a detecção de colisão.

A partir disto, se uma colisão for detectada, é sinal da existência de outra mensagem

mais prioritária sendo enviada (com preâmbulo maior), e a estação interrompe imediatamente a sua transmissão. A mensagem que possui o preâmbulo mais longo é a mais prioritária do conjunto envolvido na colisão.

O protocolo **CSMA/DCR** (*CSMA with Deterministic Collision Resolution*) apresenta um modo de funcionamento similar ao do *Ethernet*. Quando ocorre uma colisão, entretanto, o determinismo é garantido através da busca em uma árvore binária balanceada. As prioridades (índices) são atribuídas a cada estação e não às mensagens [31].

Para operar de forma correta, cada estação deve conhecer o status do barramento, seu próprio índice, além do número total de índices consecutivos alocados às fontes de mensagens (Q). O Tamanho da árvore binária é a menor potência de 2, maior ou igual a Q.

O modo de operação deste método de acesso é similar ao do CSMA/CD até a ocorrência de colisões. Ao ocorrer uma colisão, inicia-se um período de resolução por busca em árvore binária, denominado época.

Todas as estações envolvidas na colisão se autoclassificam em dois grupos: os vencedores ("*Winners*") e os perdedores ("*Losers*"). As estações do grupo das vencedoras tentam uma nova transmissão. Em caso de uma nova colisão, as estações são novamente divididas em dois grupos. Se não ocorrer nova colisão, a estação vencedora transmite seus dados.

As estações do grupo das perdedoras desistem e aguardam o término da transmissão. Caso o grupo dos vencedores esteja vazio, a busca é revertida, sendo feita uma nova divisão a partir do grupo dos perdedores.

As épocas são encerradas quando todas as estações envolvidas na colisão original conseguirem transmitir seus dados sem colisão. O tempo de duração de uma época pode ser calculado. Assim, consegue-se a obtenção de um resultado determinista.

3.4 Tecnologia ATM e as Aplicações de Tempo Real

O ATM (*Asynchronous Transfer Mode*) é uma tecnologia atual e em expansão, com altas taxas de transmissão, capaz de suportar mídias diversas, inclusive as que exigem tempo real.

A tecnologia ATM está baseada no transporte e comutação de células de tamanho fixo e reduzido, apresentando um *jitter* de transmissão bastante pequeno, uma vez que as mensagens são encapsuladas em pacotes de tamanho fixo, denominados células, que são enviadas periodicamente [2].

A inserção de um protocolo de detecção de erro, sendo executado sobre esta tecnologia, irá introduzir um *jitter* de tamanho determinado [61]. Além disso, esta tecnologia permite a implementação de sistemas de tempo real distribuídos em redes WAN.

3.5 Abordagens para a comunicação de Tempo Real

Em se tratando de sistemas de tempo real, a resolução dos problemas relativos à comunicação não está, simplesmente, baseada na definição de um protocolo determinístico para acesso ao meio. A definição da forma como será efetuado o escalonamento das mensagens também é necessária para a solução do problema [52].

A exemplo do que se faz em sistemas multitarefa, onde tarefas concorrentes são escalonadas em virtude de critérios definidos, de forma a determinar qual delas terá acesso ao processador em um dado momento, em sistemas de comunicação de tempo real esta premissa também se faz necessária, uma vez que deve existir algum critério para o escalonamento das mensagens, visando definir qual delas terá acesso ao meio de comunicação em um dado instante.

A tabela 3.1 apresenta, de forma reduzida, algumas soluções apresentadas para a problemática da comunicação em tempo real.

Tabela 3.1: Resumo das principais abordagens para a problemática de tempo real.

Abordagem	**Requisitos**	**Exemplos de Protocolos**
Atribuição de Prioridades com teste de escalonabilidade off-line	MAC com resolução de Prioridades	Token Ring com Prioridade, Comprimento de Preâmbulo, Forcing Headers (CSMA/CA)
Circuitos Virtuais TR com escalonamento on-line de mensagens	MAC com tempo de Acesso ao Meio limitado	Token Passing (Token Bus, Token Ring), TDMA, CSMA/DCR
Reserva com escalonamento Global	Requer cópias locais de todas as filas de mensagens, difundidas em Slots times de reserva.	PODA

Na seção seguinte pretende-se apresentar o sistema operacional Linux e suas características para a utilização no tratamento de tarefas de tempo real. Além disso, são des-

critos alguns dos projetos realizados com o objetivo de melhorar, através de extensões ao Linux, a forma como este atende as tarefas com requisitos temporais rígidos.

Capítulo 4

O Sistema Operacional Linux e o Tempo Real

O Linux é um Sistema Operacional (SO) baseado na filosofia UNIX, gratuito e com código fonte aberto e apresenta todos os requisitos fundamentais esperados de um sistema operacional moderno [36]. A saber, multitarefa, multi-usuário, memória virtual, bibliotecas compartilhadas, proteção de memória, suporte a máquinas com SMP (multiprocessamento simétrico), flexibilidade do Posix e uma pilha de rede.

O Linux é um SO de 32bits (64 bits em CPU's de 64 bits), originalmente concebido por Linus Torvalds, na época estudante no Departamento de Ciência de Computação da Universidade de Helsinki. Sua implementação foi realizada nas linguagens C e *Assembly*, sendo esta última utilizada apenas em áreas estritamente dependentes da arquitetura ou em regiões de código onde existia a necessidade de customizar a velocidade de processamento [12].

4.1 Núcleo do Linux

O núcleo (*kernel*) é o âmago de um sistema operacional. Ele é responsável pela gerência do *hardware*, pela alocação dos recursos computacionais do sistema entre processos e pelo escalonamento dos processos ativos com o objetivo de que estes sejam executados de forma concorrente [12].

O núcleo do sistema operacional tem, entre outros, o objetivo de evitar que as

aplicações mantenham qualquer acesso direto ao *hardware*, viabilizando este acesso através de serviços por ele disponibilizado. Isto possibilita que seja mantida a integridade entre os processos dos usuários e os do próprio sistema.

Como forma de externar suas funcionalidades são implementadas as chamadas de sistema, que têm por finalidade fornecer acesso a algumas áreas sob o controle do núcleo às aplicações usuárias.

Dentre os componentes importantes que constituem o núcleo do Linux estão: gerenciamento de memória, gerenciamento de processos, gerenciadores de dispositivos, sistema de arquivos, tratamento de rede. A figura 4.1 demonstra um diagrama simplificado do sistema operacional Linux, nela estão representados os subsistemas que compõem seu núcleo e a forma que os mesmos se relacionam [58].

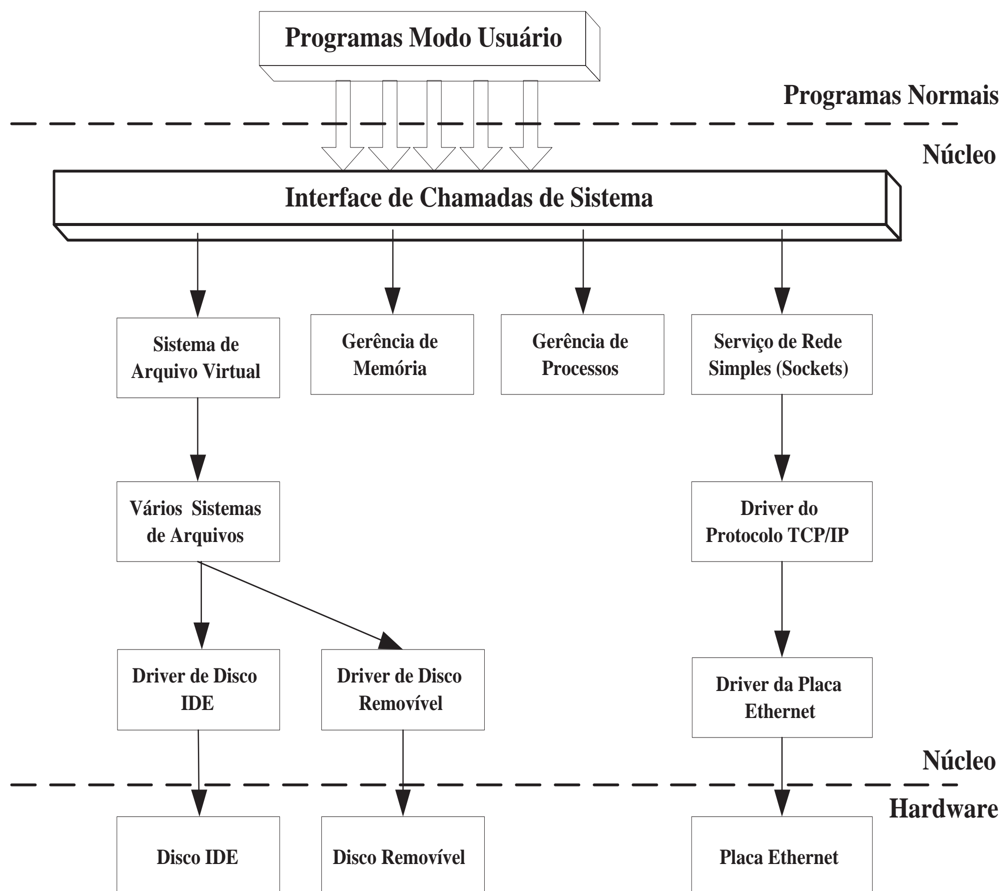

Figura 4.1: Núcleo do Linux [58]

No Linux, todos os subsistemas que compõem o núcleo são executados em modo supervisor e compartilhando, desta forma o mesmo espaço de endereçamento. A

comunicação entre os subsistemas ocorre através de chamadas de função em C [1].

Convencionalmente existem duas formas principais de desenvolvimento do núcleo de um sistema Operacional, a saber, na forma monolítica ou de micronúcleo.

Um sistema monolítico é aquele em que todos os componentes do sistema operacional fazem parte de um código único [53]. Esse código único pode ser visto como o "processo sistema operacional" que executa em modo supervisor, tendo acesso ao conjunto total de instruções do processador. A maioria dos sistemas operacionais UNIX tem seu núcleo organizado desta forma [39].

Os sistemas operacionais que possuem sua organização baseada em micronúcleo possuem um núcleo pequeno, com um número reduzido de funcionalidades, tais como primitivas de sincronização, um escalonador simples e um mecanismo de comunicação entre processos [5, 39].

Este diminuto núcleo possui dependência intrínseca com a plataforma de *hardware* para a qual foi implementado. As demais funcionalidades do sistema operacional são implementadas como processos (processo escalonador, processo gerente de memória, processo gerente de disco, etc.) e a troca de informações entre estes processos e com o micronúcleo, ocorre através da troca de mensagens [5, 39].

O Linux implementa um meio termo entre o núcleo monolítico e o micronúcleo, utilizando o conceito de módulos [12]. Os Módulos são "fragmentos do sistema operacional" que podem ser carregados ou descarregados dinamicamente em tempo de execução sem que exista a necessidade de se recompilar o núcleo ou até mesmo reiniciar a máquina. Uma vez carregado um módulo ele passa a ser parte do núcleo, sem nenhuma forma especial de restrições de permissão.

Essa característica possibilita flexibilidade ao sistema operacional, além de otimizar a utilização de memória por parte do núcleo. Isto proporciona que novas funcionalidades possam ser inseridas ao sistema operacional, sem a necessidade do mesmo ter seu núcleo recompilado, ou até mesmo que a máquina tenha de ser reiniciada. Sendo somente necessário que a nova funcionalidade tenha de ser compilada na forma de módulo e carregada ou descarregada quando isto se fizer necessário [46].

Na realidade, o que ocorre é uma união do desempenho de um sistema monolítico, com a modularidade e portabilidade de um micronúcleo. Desta forma, apenas são car-

regadas para a memória as partes estritamente necessárias para que o núcleo possa ser executado, as demais partes (módulos) são carregadas apenas quando realmente se fizerem necessárias [12].

Para o desenvolvimento de módulos abertos a toda a comunidade de usuários, o Linux adota um conjunto de regras que definem interfaces para os seus subsistemas e estruturas de dados que serão utilizadas no carregamento e descarregamento dos módulos. Para o gerenciamento dos módulos, também foram definidas no Linux chamadas de sistema que possibilitam a carga e a remoção dos mesmos em tempo de execução [36].

A grande vantagem dos módulos, em nível de programação, é a possibilidade de que seu código fonte possa ser alterado e compilado, sendo necessário apenas carregá-lo novamente para que as alterações tenham efeito. Esse processo pode ser repetido tantas vezes quanto for necessário, sem ser preciso reiniciar o sistema [40, 46].

As distribuições do Linux utilizam-se de módulos de maneira extensiva. Durante o procedimento de instalação do sistema é utilizado um núcleo básico e muitos módulos possibilitando assim, que uma instalação de determinada distribuição funcione corretamente em qualquer máquina, sendo apenas necessário agregar a este núcleo os módulos necessários para o pleno funcionamento do *hardware* [40].

4.2 O Linux como um SO de Tempo Real

O Linux é um Sistema operacional que se baseia no princípio de compartilhamento de tempo, tendo por objetivo fornecer um bom desempenho na média, mas não pode garantir um sincronismo temporal exato [6]. Em virtude de seu núcleo seguir o modelo do Unix tradicional, isto não o torna apropriado para tratar aplicações com requisitos temporais críticos [19]. Existem alguns outros problemas, que impedem que o Linux seja utilizado para o tratamento de atividades com requisitos de tempo real crítico.

No Linux existe uma classe de processos denominados de processos de tempo real. Contudo, isto não significa que estes processos sejam realmente de tempo real, pois um processo desta classe é pura e simplesmente uma tarefa de prioridade elevada no Linux [19].

Os processos de tempo real quando prontos para a execução sempre serão escalonados

antes de outras tarefas com prioridades mais baixas. Contudo, muitas vezes, estes têm de ficar aguardando um longo período de tempo até que um processo de prioridade mais baixa termine seu trabalho e assim venha executar. Esta classe de processos de tempo real pode ser muito útil em algumas aplicações, mas isto não é o bastante para um sistema requisitos temporais críticos.

O suporte do Linux para TR pode ser chamado de ”brando”. Usar a classe de processos de TR do Linux pode naturalmente ser útil e eficaz em sistemas com requisitos temporais que não sejam rígidos.

Em Tatibana [55] encontra-se um estudo aprofundado da utilização do Linux para tarefas de tempo real brando. Pode-se constatar que, apesar deste sistema não possuir preocupações temporais rígidas em seu projeto demonstra um bom comportamento temporal na realização de tarefas de tempo real brando.

4.3 Linux X Tempo Real Crítico

O Linux foi projetado com a finalidade de assegurar uma distribuição justa, do tempo de utilização de CPU para cada processo no sistema [12]. No Linux cada processo possui uma prioridade, a qual não depende somente da importância do processo. A prioridade pode ser alterada dependendo do comportamento do processo, com a finalidade de que o sistema inteiro venha a ter um bom desempenho médio. Isto é conveniente, por exemplo, a um usuário *desktop* normal. Todavia este tipo de comportamento, não é próprio de um sistema com rígidos requisitos temporais, pois a execução de um processo dependerá da carga do sistema e do comportamento dos outros processos. Desta maneira pode-se constatar que o comportamento do sistema é imprevisível, o que é impróprio aos sistemas com requisitos temporais críticos. Um dos principais motivos que faz com que o Linux não seja vocacionado para tempo real crítico é o fato de não ser preemptivo [12].

Um sistema operacional é dito preemptivo quando toda a tarefa pode ser interrompida por uma outra de prioridade mais elevada, sempre que esta necessitar executar. No Linux, embora as tarefas sejam interrompidas continuamente, a medida que este compartilha seus recursos entre todos os processos, as interrupções ainda não podem ocorrer a qualquer momento.

Além disto, o núcleo do Linux pode desabilitar as interrupções por um longo período, não importando se estas se fazem necessárias ou não [6]. Quando um processo faz uma chamada de sistema, todas as interrupções são desabilitadas. Um processo, por si mesmo, pode também desabilitar as interrupções, sempre que necessário. Isto impede que tarefas de prioridade mais alta obtenham o controle sobre o sistema a qualquer momento que necessitem. Se uma tarefa de prioridade mais elevada necessitar ser executada pode ter de aguardar até que uma tarefa de mais baixa prioridade volte a habilitar as interrupções, ou que esta termine sua execução, o que é inaceitável em um sistema TR, visto que impede que o sistema responda prontamente aos eventos de tempo real.

Outro problema encontrado no Linux é o fato da resolução do tempo ser baixa, para tratar tarefas com restrições temporais críticas. Os sistemas de tempo real, tipicamente, utilizam-se de interrupções periódicas de relógio para invocar o mecanismo escalonador, com a finalidade de verificar se existe a necessidade da troca de tarefas a serem executadas. O Linux também trabalha desta maneira [36].

Nas arquiteturas compatíveis com a do PC da IBM o Linux programa o temporizador de *hardware* para gerar interrupções a uma taxa de 100Hz, possibilitando uma definição de 10 milissegundos [9]. Isto significa que o núcleo não irá verificar se existe alguma tarefa ou evento pendente com uma periodicidade menor do que 10 milissegundos [7].

As tarefas que são iniciadas de acordo com este temporizador (e não de acordo com uma fonte externa), não podem ser inicializadas de forma mais precisa do que esta definição de 10ms. Para muitas aplicações de tempo real isto não é o suficiente, pois exigências de precisão de ordem menor a de 1 milisegundos não são incomuns. Naturalmente é possível usar uma taxa mais elevada de interrupção, mas isto pode causar um grande *overhead* [1].

A utilização de uma taxa muita elevada de interrupções consome desnecessariamente tempo de CPU, pois estas passarão a ocorrer em intervalos específicos independentemente da necessidade de um interruptor de tarefa ser executado.

[1] *Overhead*: Custo adicional em processamento ou armazenamento que, como conseqüência piora o desempenho de um programa ou de um dispositivo de processamento. Usado normalmente para se referir a custos adicionais indesejáveis, que deveriam ou poderiam ser evitados.

4.4 Modificando o Linux

Modificar o núcleo do Linux para adicionar o suporte a tarefas de TR não é algo tão trivial, devido à sua complexidade e tamanho. Outro problema reside no fato do Linux ser alvo de freqüentes modificações, sendo assim, estas mudanças teriam de ser refeitas a cada nova versão do núcleo.

4.5 Os benefícios do Linux como Sistema de Tempo Real Brando

A Utilização do Linux com o sistema para suporte a tarefas de tempo real brando justifica-se por uma série de características que o tornam uma alternativa viável. Dentre essas pode-se citar:

- Baixo custo (Não existe a necessidade de aquisição de Licenças);
- Código aberto (Possibilita que modificações e correções possam ser feitas sob demanda);
- Suporte (SO extensamente utilizado, existe um vasto conhecimento a seu respeito);
- Estabilidade;
- Confiabilidade (sistema consolidado e provou ser estável e confiável);
- Modularidade;
- Portabilidade (disponível em diversas plataformas de *Hardware*, portá-lo para outras plataformas é algo relativamente fácil);
- Pode ser utilizado em sistemas embarcados.

Em virtude das características acima citadas e com a necessidade de satisfazer as aplicações com requisitos temporais de críticas, algumas pesquisas e implementações foram realizadas com o objetivo de tornar o Linux um SO adequado para tratar as tarefas de tempo real de maneira eficaz e ainda propiciar uma a vasta gama de facilidades existentes neste sistema.

4.6 Extensões do Linux para Tempo Real

Em virtude de o Linux possuir recursos que facilitam sua adaptação para o contexto de tempo real, muitas pesquisas e adaptações foram realizadas sobre o seu núcleo, visando tornar o sistema eficaz no tratamento de tarefas com requisitos temporais ou simplesmente melhorar seu desempenho neste caso. Algumas das implementações que se destacam são: RED LINUX [57], KURT LINUX [38], RTLINUX [7], RTAI [35]. Destes sistemas, a solução mais conhecida é o RTLINUX que durante muito tempo foi usado como base para o desenvolvimento de outros sistemas operacionais.

Em http://www.realtimelinuxfoundation.org/variants/variants.html encontra-se uma lista de projetos relacionados ao Linux, inclusive adaptações para o cumprimento de tarefas com requisitos de tempo real.

4.6.1 RED LINUX

O RED LINUXé um projeto de pesquisa da Universidade da Califórnia em Irvine. O objetivo deste projeto é adicionar potencialidades de tempo real ao núcleo do Linux, com foco em aplicações que misturam requisitos temporais e sem tais requisitos.

O suporte a tarefas de tempo real é adicionado modificando o núcleo do Linux, sendo o oposto do que é proposto pelo RTAI e RTLINUX, onde um micronúcleo de tempo real coexiste com um núcleo Linux de propósito geral.

O RED LINUX pode ser utilizado em projetos de pesquisa onde algoritmos de escalonamento diferentes são comparados. Este sistema inclui diversos mecanismos de escalonamento que podem ser utilizados. A troca destes se dá através de alguns ajustes nos atributos de sincronização.

4.6.1.1 Princípios de Implementação

Para tornar o Linux um sistema voltado para tempo real, três novos componentes foram adicionados ao SO: Um microtemporizador, um escalonador *time-driven* e um software emulador de interrupções [57].

O microtemporizador e o software emulador de interrupção foram portados do RTLINUX. O escalonador foi adicionado ao núcleo do Linux, em conjunto com os escalonado-

res existentes no Linux.

O projeto do RED LINUX não visou tornar o núcleo original do Linux totalmente preemptivo, pois isto demandaria muito esforço. Em lugar disto, foram introduzidos pequenos pontos de preempção em seu código. Em cada um destes pontos o núcleo verifica se existe alguma tarefa de tempo real pendente para ser executada. Tal dispositivo resulta na diminuição dos tempos de bloqueio, prevenindo que as tarefas de tempo real sejam iniciadas a qualquer momento.

4.6.1.2 Escalonamento

O RED LINUX fornece três diferentes algoritmos de escalonamento [57]:

Time-driven - este escalonador utiliza o tempo atual e uma programação predefinida para decidir que tarefa será executada em seguida. Esta programação é pré-estabelecida antes da execução e não pode ser alterada em tempo de execução. Este escalonador não faz parte do núcleo original do Linux, tendo sido adicionado pelo RED LINUX.

Priority-driven - Neste mecanismo a tarefa com a prioridade mais elevada pronta para funcionar será sempre a próxima ser executada. Este mecanismo de escalonamento esta presente no núcleo do Linux.

Share-driven - esta política de escalonamento visa o compartilhamento dos recursos de uma maneira uniforme e razoável entre todas as tarefas. Este é o modo de escalonamento normal do Linux.

Cada um destes algoritmos de escalonamento possui benefícios e desvantagens. O objetivo de RED LINUX é combinar os benefícios de todos os três usando de forma simultânea, pois em tempo de execução o escalonador associa estes três algoritmos de escalonamento. A maneira em que são mesclados é definida pela política de escalonamento utilizada e definida de acordo com as necessidades de uma aplicação específica.

Para cada tarefa no sistema, os seguintes atributos de escalonamento são definidos [57]:

- *Priority* - define a importância da tarefa em relação às outras no sistema
- *Start time* - define o tempo no qual a tarefa pode ser executada (tarefas não podem ser executada antes de serem originadas).

- *Finnish time* - define o prazo final da tarefa (deadline). A tarefa deve sempre ser terminada antes de exceder seu prazo final.
- *Budget* - define a quantidade de recursos de CPU que estão reservados para a tarefa.

A política de escalonamento é definida então ajustando a importância relativa destes atributos e dos valores destes atributos em cada tarefa. A política de escalonamento pode também ser modificada em tempo de execução.

4.6.2 KURT LINUX

O KURT LINUX foi desenvolvido na Universidade de Kansas, tendo como objetivo o desenvolvimento de um SO de tempo real baseado no Linux. O KURT LINUX é um sistema de tempo real "*Firm*", uma mescla entre um sistema de tempo real brando e crítico, possuindo diferentes maneiras para tratar tarefas de TR ou sem requisitos temporais. Para tornar o Linux um sistema voltado a aplicações de tempo real, foi modificado o mecanismo de sincronismo e adicionado ao núcleo do Linux, um escalonador de tempo real [38].

4.6.2.1 Princípios de Implementação

O princípio básico por trás do KURT LINUX, é a operação do núcleo em 3 modos [17]:

Modo normal (*normal mode*) - o sistema opera de forma similar a um sistema Linux convencional.

Modo concentrado (*focused*) - neste modo o SO executa somente processos da classe de tempo real.

Modo misto (*mixed*) - o sistema executa tanto processos de tempo real ou tarefas sem restrições temporais.

O KURT LINUX alterna entre esses modos de operação através da utilização de uma chamada de sistema. Todos os eventos de tempo real devem ser predefinidos em uma escala quando o sistema é configurado.

A política de escalonamento é implementada através de um escalonador cíclico. Este tipo de escalonador se baseia no uso de uma tabela, onde são anotadas todas as ações de planificação: instante de ativação, tarefa a executar, duração destas.

Durante a execução cabe ao escalonador apenas o trabalho da leitura seqüencial e execução da escala de trabalho. O escalonador entra, no modo concentrado para manipular os eventos de tempo real. Quando não existe nenhum trabalho com restrições temporais a ser realizado o escalonador passa a operar no modo normal. O KURT LINUX é preemptivo somente quando no modo concentrado [17].

Os eventos de tempo real podem ser atrasados quando o sistema estiver operando no modo normal e as interrupções estiverem desabilitadas no momento em que a troca de modo de execução deveria ocorrer. Desta forma, pode-se afirmar que isto limita a usabilidade do KURT LINUX em muitas aplicações de tempo real crítico.

4.6.2.2 O Mecanismo Temporizador

Com a finalidade de aumentar a resolução do tempo, o mecanismo temporizador do Linux foi modificado. O princípio básico é similar ao utilizado pelo mesmo componente do RTLINUX. Ao invés das interrupções serem geradas periodicamente em um intervalo determinado de tempo, um temporizador é programado para gerar interrupções. O tempo para próximo evento é lido em uma escala, desta forma o temporizador é programado para interromper neste momento [38].

Esta implementação propicia um sincronismo preciso, com definição na ordem de microssegundo, sem ter a necessidade de que o sistema tenha de ser interrompido na mesma proporção. Além deste temporizador programável, o Linux requer ainda um temporizador periódico normal que gera interrupções em um intervalo de tempo fixo.

A fim de satisfazer tal exigência, a simples introdução de uma interrupção periódica extra nesta escala, irá gerar esta interrupção.

4.6.3 RTLINUX

Originalmente, o RTLINUX teve início como um projeto de pesquisa no Instituto de Tecnologia do Novo México e tinha como um dos seus objetivos o desenvolvimento um SO de tempo real, não comercial, para controle de instrumentos e de robôs. Alem disto, outro objetivo era utilizar um SO para pesquisa em projeto de sistemas com ou sem restrições temporais.

Atualmente o RTLINUX é mantido pelo Finite State Machine Labs Inc., uma companhia fundada pelos criadores do RTLINUX. Este SO foi portado para diversas plataformas de *hardware* e é utilizado para a aquisição de dados, controle e comunicações em tempo real, dentre outras aplicações.

4.6.3.1 Princípios de Implementação

Os princípios de execução do RTLINUX são os mesmo de qualquer SO voltado para execução de tarefas de tempo real crítico. A idéia básica por trás do RTLINUX é adicionar junto ao núcleo de propósito geral (núcleo do Linux) um novo núcleo de tempo real, mas fora desse. Isto significa que o SO possui desta forma dois núcleos, um padrão do Linux e o outro para as atividades de tempo real crítico.

O núcleo de TR é a base do sistema, tomando conta de todas as tarefas de tempo real e rodando o núcleo do Linux sobre si como se fosse apenas uma tarefa de baixa prioridade. Desta forma, o Linux irá funcionar sob o controle de um núcleo voltado para atividades de tempo real crítico. Esse núcleo toma para si todo o trabalho de tempo real como suas próprias tarefas, e sempre que não há nenhum trabalho de tempo real a ser executado as tarefas Linux são escalonadas para rodar. A figura 4.2 demonstra o princípio de implementação do RTLINUX.

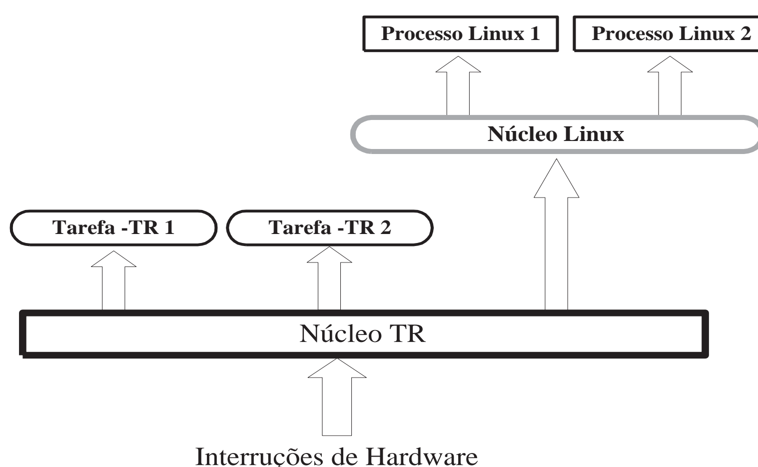

Figura 4.2: Princípio de Implementação do RTLINUX

O benefício da utilização deste modelo de projeto de um sistema com dois núcleos, um voltado para atividades de tempo real e outro para atividades gerais, é que somente

algumas pequenas modificações necessitam ser feitas no núcleo do Linux. Desta forma este núcleo de tempo real pode ser adicionado ao sistema como um módulo separado. Isto torna o RTLINUX facilmente portável às novas versões do Linux [6].

O núcleo RTLINUX fornece somente os mecanismos para criação e escalonamento de tarefas, serviços de interrupção e uma comunicação de baixo nível entre tarefas. Desta forma, o fornecimento de todos os demais serviços necessários ficam ao encargo do Linux.

Uma aplicação rodando no RTLINUX deve ser dividida em duas partes, uma de tempo real e a outra não. A primeira, voltada à satisfação dos requisitos de tempo real é mantida tão pequena e simples quanto possível, com a finalidade de garantir os requisitos temporais. A outra parte não deve ter nenhuma exigência de tempo real, possuindo todos os recursos do Linux disponíveis para utilização.

A comunicação entre estas duas partes da aplicação ocorre através das funcionalidades fornecidas pelo RTLINUX. Com a finalidade de fornecer esta facilidade, o RTLINUX implementa alguns dispositivos, tais como as filas FIFO e memória compartilhada [6].

4.6.3.2 O Escalonador de Tempo Real (*Real-time scheduler*)

O Escalonador de tempo real do RTLINUX é um módulo do núcleo, podendo, desta forma, ser facilmente alterado (carregado e descarregado), a fim de melhorar o suporte a uma dada aplicação possibilitando que o algoritmo mais apropriado ao escalonamento desta tarefa possa ser utilizado. Como padrão, um simples escalonador preemptivo é utilizado, onde a tarefa de mais alta prioridade e pronta para a execução sempre será programada para ser a próxima a ser realizada. Contudo, sempre que uma tarefa estiver no estado de pronta para execução, pode solicitar o uso dos recursos a outra de mais baixa prioridade, e esta tarefa, de prioridade inferior, que esta executando deve liberar imediatamente o recurso.

O Escalonador do RTLINUX possui suporte a tarefas periódicas, incluindo um algoritmo de escalonamento para as mesmas.

4.6.3.3 Comunicação Entre Processos (IPC)

No RTLINUX não é possível que uma tarefa de tempo real chame diretamente nenhuma rotina do núcleo do Linux, em virtude de que uma outra tarefa de tempo Real pode interromper o núcleo a qualquer tempo [7]. Todavia, todos os serviços do núcleo do Linux necessários aos processos de tempo real podem ser alcançados através de rotinas convencionais do Linux. Para isto, o RTLINUX inclui um suporte a comunicação entre o núcleo do Linux e/ou de seus processos e o núcleo de tempo real e suas tarefas. Esta comunicação ocorre através de um mecanismo denominado de filas FIFO.

O funcionamento destas filas é muito similar ao que acontece em *PIPE's* do Unix, onde a comunicação é por fluxo de dados sem estrutura. Uma fila FIFO é um *buffer* de *bytes* de tamanho fixo, sobre a qual pode-se fazer operações de leitura e escrita.

O RTLINUX implementa ainda, um outro dispositivo de comunicação entre as tarefas de tempo real e as convencionais, e esta comunicação pode ocorrer através de memória compartilhada.

4.6.3.4 Manipulador de Interrupções

Ao invés de modificar o Linux, com a finalidade deste se tornar preemptivo, o RTLINUX utiliza-se de um mecanismo de *software* que emula o manipulador de interrupções para o Linux. O RTLINUX controla, então, todas as interrupções emitidas ao Linux habilitando ou desabilitando-as, quando julgar possível.

As interrupções são divididas em dois grupos: as controladas pelo Linux (interrupções não de tempo real) e aquelas controladas pelo núcleo de tempo real (interrupções de tempo real). As interrupções geradas pelo núcleo de tempo real ou por tarefas desta classe são controladas diretamente pelo núcleo de tempo real. Contudo, as interrupções que pertencem à classe das tarefas que não possuem requisitos temporais são manipuladas de uma forma especial, pelo emulador de interrupções [7].

O emulador de interrupções é uma camada de software que esta localizada entre o núcleo do Linux e o *hardware* controlador de Interrupções. Este *software* captura todas as interrupções de *hardware* convencionais e, através disso, controla as interrupções enviadas ao Linux. Quando esse invoca uma rotina que pode desabilitar ou habilitar

interrupções, a chamada é capturada pelo emulador de interrupções. O princípio de funcionamento deste manipulador de interrupções no RTLINUX é demonstrado na figura 4.3.

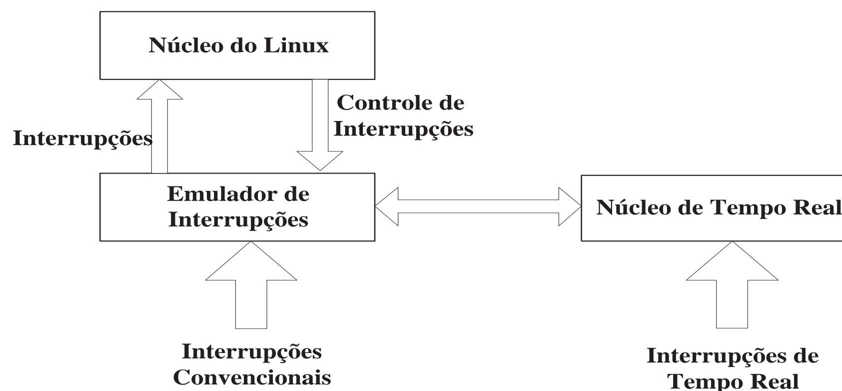

Figura 4.3: Controle de Interrupções no RTLINUX

O objetivo do software emulador é fazer com que o ato de manipular interrupções pareça inalterado para o Linux. Desta forma, quando esse tenta desabilitar uma interrupção, o que ocorre realmente é que elas não são desabilitadas, ao invés disso, apenas um *flag* é marcado no emulador.

Quando o emulador captura uma interrupção, primeiro verifica o estado do flag correspondente à mesma. Se este não estiver marcado (interrupções habilitadas), a interrupção é repassada diretamente ao manipulador de interrupções do Linux. Caso contrário (interrupções desabilitadas), o emulador espera até que a interrupção seja habilitada e, somente depois disto, ela é repassada. As interrupções de tempo real não são tratadas por este emulador, sendo diretamente manipuladas pelo núcleo de tempo real. O benefício da utilização deste mecanismo de manipulação de interrupções, é o fato de que apenas pequena modificações são necessárias no núcleo.

A implementação deste mecanismo é o que possibilita que as interrupções sejam apenas controladas (habilitadas e desabilitadas) pelo núcleo de tempo real. Contudo, do ponto de vista do núcleo do Linux, o ato de manipular as interrupções parece permanecer inalterado, o sistema permanece com a visão de estar gerenciando as interrupções diretamente. O grande beneficio disto é o fato de que apenas pequenas modificações são necessárias no núcleo de propósito geral.

4.6.3.5 Sincronismo

O Escalonador de Tarefas de tempo real utiliza-se de um temporizador de intervalo programável, chamado micro-temporizador. O objetivo deste é interromper os eventos de tempo real. Seu funcionamento é o oposto ao de um temporizador periódico, pois as interrupções ocorrerão somente quando necessárias, possibilitando a economia da capacidade da CPU, propiciando uma exatidão muito boa do tempo. Além do micro-temporizador, o Linux ainda requer existência de um temporizador periódico. O emulador de interrupções é que faz o papel deste temporizador periódico, gerando interrupções periódicas.

Na próxima seção serão apresentados os conceitos e características de funcionamento do RTAI, uma outra extensão do Linux para tempo real, e sua pilha de protocolos de rede de tempo real RTNET, que figuram como alvo deste trabalho.

Capítulo 5

O Ambiente de Tempo Real RTAI e RTNET

5.1 RTAI LINUX

O RTAI (*Real-Time Application Interface*) teve seu início no Departamento de Engenharia Aeroespacial, Politécnico de Milão. O projeto visava a utilização de computadores do padrão PC para controle de sistemas com requisitos temporais rígidos.

A primeira versão do RTAI foi criada para rodar sobre o DOS, uma plataforma de 16 bits. Em meados de 1997, pesquisadores do DIAMP iniciaram a considerar a migração do RTAI para uma plataforma de 32 bits. O sistema cotado para ser o alvo do projeto foi o Linux. Um dos principais motivos para a escolha deste sistema como plataforma alvo foi o fato de que já existiam projetos bem sucedidos na utilização do mesmo para controle de tarefas de tempo real crítico.

Por ocasião do surgimento da versão 2.2 do núcleo do Linux, a qual se mostrava estável e possuía uma boa organização em seu código fonte, tendo em vista as versões anteriores, tornou-se possível a realização de pesquisas para modificações no Linux, a fim de satisfazer os requisitos de tempo real das aplicações, fossem iniciadas no DIAMP.

Assim como no caso do RTLINUX, o projeto do RTAI procurou fazer o mínimo de modificações no núcleo do Linux. Esta característica tornou-se posteriormente, um atrativo desse sistema, pois os códigos do RTAI e do Linux são praticamente independentes.

5.2 Princípios de Implementação

A arquitetura do RTAI [35] é muito similar a do RTLINUX [7]. Assim como no RTLINUX, o núcleo convencional do Linux é tratado como uma tarefa de tempo real de baixa prioridade, que pode executar suas ações normalmente sempre que não existirem tarefas de tempo real com prioridade mais elevada prontas para a execução. Durante a operação básica do RTAI, as tarefas de tempo real são executadas como módulos do núcleo.

O RTAI recebe as interrupções dos periféricos e após ter assegurado que todas as ações de tempo real necessárias pelas interrupções tenham sido tratadas, as entrega ao núcleo do Linux. A figura 5.1 demonstra a arquitetura básica do RTAI, que é similar a arquitetura do RTLINUX.

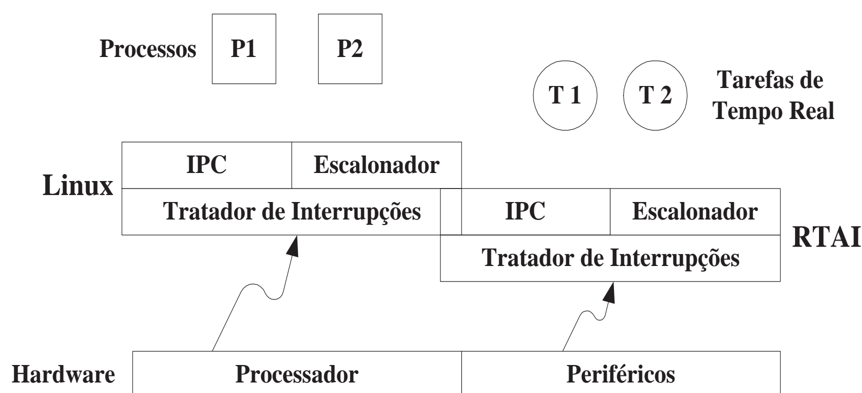

Figura 5.1: Visão geral da arquitetura RTAI Linux [37].

As interrupções podem ser originadas pelo processador ou pelos periféricos. Por ocasião da utilização de um sistema Linux-RTAI, as interrupções que são originadas pelo processador permanecem sendo tratadas pelo núcleo padrão Linux. Contudo, as interrupções ocasionadas por periféricos são atendidas pelo manipulador de interrupções do RTAI. Ao receber o pedido de uma interrupção, o sistema a repassa ao manipulador de interrupções padrão do Linux. Isto ocorre somente quando não mais existirem tarefas de tempo real ativas.

Para possibilitar a implementação de tal mecanismo, as instruções que habilitam e desabilitam as interrupções no núcleo do Linux foram substituídas por macros que repassam as instruções para o RTAI. Quando as interrupções são desabilitadas no núcleo de propósito geral (Linux), o RTAI (SOTR) as enfileira e entrega ao Linux, tão logo

este habilite novamente as interrupções [35]. A figura 5.2 mostra como são tratadas as interrupções neste sistema.

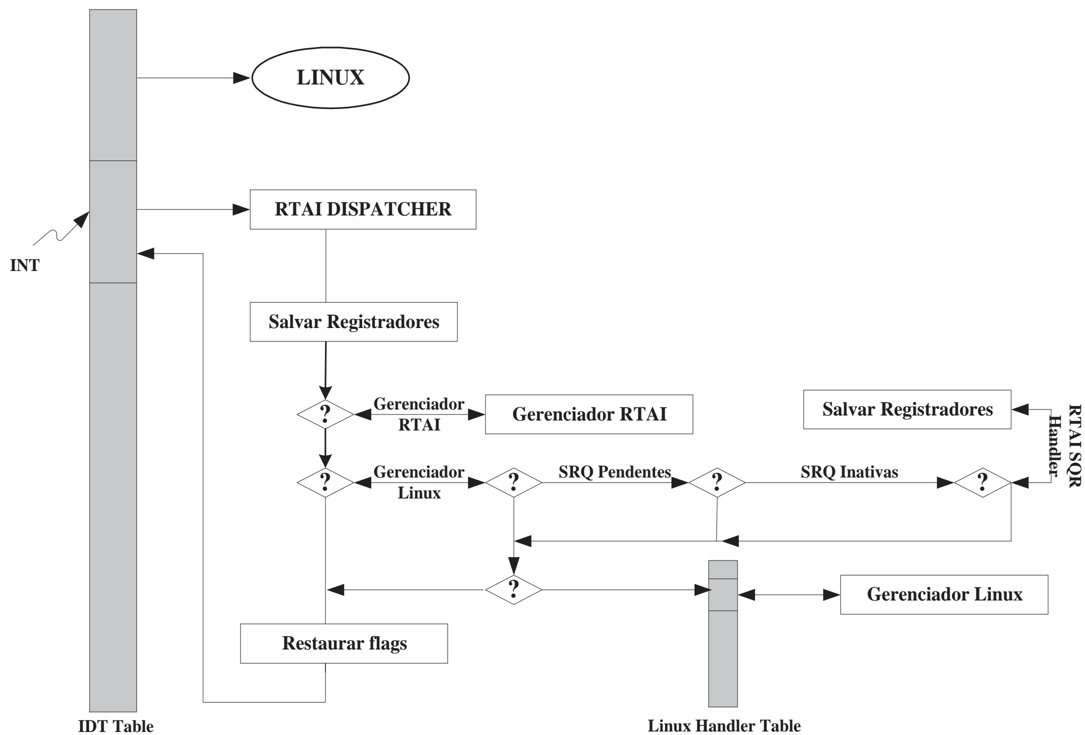

Figura 5.2: Tratamento de Interrupções no RTAI [37].

5.3 Camada de Abstração de Hardware (*Hardware Abstraction Layer*)

Os desenvolvedores deste SOTR introduziram o conceito de uma camada de Abstração de *hardware* de tempo real denominada RTHAL (*Real-Time Hardware Abstraction Layer*). Esta camada é utilizada para interceptar e processar as interrupções de *hardware* [33].

A RTHAL é uma estrutura, instalada junto ao núcleo do Linux, que reúne a tabela do manipulador de interrupções de *hardware* e as funções necessárias para o RTAI operar. O objetivo desta camada é minimizar o número de mudanças necessárias ao código do Linux, visando melhorar a sustentabilidade dos códigos do RTAI e do Linux. Por ocasião

da instalação da RTHAL junto ao núcleo do núcleo de propósito geral, as chamadas de função e as estruturas de dados relacionadas à interação com o hardware são substituídas por ponteiros para a estrutura da camada RTHAL [13]. Inicialmente, as estruturas contêm os ponteiros às funções originais e dados da execução do Linux, mas a quando RTAI é habilitado, é necessário substituir por ponteiros na tabela da RTHAL.

5.4 Escalonamento no RTAI

O RTAI possui unidades escalonáveis que são denominadas tarefas. Há sempre pelo menos uma tarefa em execução no sistema, o núcleo do Linux, que roda como uma tarefa da baixa prioridade. Quando as tarefas de tempo real são adicionadas, o escalonador lhes dá prioridade sobre o Linux.

O escalonador fornece os serviços como *suspend*, *resume*, *yield*, *make periodic*, *wait until*, que são utilizados em vários sistemas operacionais de tempo real. O escalonador é executado como um módulo dedicado do núcleo (similar ao RTLINUX), o que torna relativamente fácil a implementação de novos escalonadores, caso isto se torne necessário.

O RTAI possui 3 tipos diferentes de escalonadores, que estão diretamente relacionados com o tipo de máquinas em que o sistema está sendo executado [33, 10].

O escalonador uniprocessador (UP), cuja utilização é projetada para plataformas de *hardware* com apenas um processador, e não pode ser utilizado em máquinas com multiprocessadores [15].

O escalonador para máquinas com multiprocessamento simétrico (SMP), foi desenvolvido para a utilização em máquinas com suporte a SMP e fornece uma interface para que as aplicações possam selecionar um processador ou um conjunto de processadores, onde uma determinada tarefa deve ser executada [33]. Caso não se determine nenhum processador, onde a tarefa deve ser executada, o SMP selecionará um processador baseado no status da carga do mesmo.

O multi-uniprocessor (MUP) é um escalonador que pode ser utilizado tanto para máquinas com um único processador como para máquinas com múltiplos processadores. Todavia, este age de forma distinta ao escalonador para máquinas com SMP, pois quando o escalonador MUP é selecionado as tarefas são limitadas a um processador específico

[33]. O lado positivo é que o escalonador MUP permite mecanismos mais flexíveis para o temporizador das tarefas em relação aos escalonadores SMP e UP.

O RTAI utiliza o FIFO como política padrão de escalonamento para tarefas de tempo real crítico, mas também oferece as políticas *Round Robin* (divisão de tempo), *Rate Monotonic* e *Earliest Deadline First*.

5.5 Características do RTAI

Com o objetivo de tornar o desenvolvimento de aplicações tão flexíveis quanto possível, os desenvolvedores do RTAI introduziram uma série de mecanismos diferentes para permitir a comunicação interprocessos (IPC) entre tarefas de tempo real e os processos em espaço usuário. Além dos mecanismos de IPC, ainda são fornecidos serviços para a gerência de memória e *threads* compatíveis com o *Posix*.

5.5.1 Comunicação Interprocessos (IPC)

O RTAI fornece uma variedade de mecanismos para comunicação interprocessos. Embora os sistemas Unix forneçam esses mecanismos de IPC, objetivando a comunicação de processos rodando em espaço usuário, o RTAI necessita fornecer seus mecanismos de IPC às tarefas de tempo real. Isto ocorre em virtude de não ser possível que tarefas de tempo real utilizem as chamadas de sistema padrão do Linux.

Os diferentes dispositivos de IPC são incluídos como módulos do núcleo, e podem ser carregados em acréscimo aos módulos básicos do RTAI e do escalonador, quando estes se fazem necessários. Uma vantagem adicional da utilização dos módulos é que os serviços do IPC podem facilmente ser personalizados e expandidos.

5.5.1.1 FIFOS

Outra forma de implementação de IPC dentro do RTAI são as Filas FIFO. Uma FIFO é um canal de sentido único assíncrono, não sujeito a bloqueios, entre um processo Linux e uma tarefa de tempo real [33]. Esta fila possui seu tamanho determinado pela necessidade do usuário. Os dados não podem ser sobrescritos em uma FIFO. Desta maneira, estas

filas podem ser completamente preenchidas impossibilitando que novos dados possam ser escritos até que algum dos dados anteriores seja consumido. Uma FIFO é vista pelo espaço usuário como uma entrada no diretório `/dev`, onde existe um conjunto que pré aloca 64 entradas na FIFO [33].

A implementação das FIFOS no RTAI é baseada na implementação do RTLINUX, mas o RTAI fornece algumas características que não estão disponíveis no outro sistema. Em primeiro lugar, é permitida a ativação de um sinal quando há eventos na FIFO (como a escrita de novos dados). Um processo rodando em espaço usuário pode criar, então, manipuladores para estes sinais, utilizando-se das funções padrões Linux. Contudo, isto não se faz necessário, pois processos rodando em espaço usuário podem simplesmente fazer uso das operações padrão de entrada e saída para escrever ou ler dados contido nestas filas [33].

Adicionalmente, é possível a existência de vários processos leitores e escritores relacionados a uma FIFO, o que não era possível na versão no RTLINUX. Para possibilitar que vários processos se utilizem destes dispositivos de comunicação compartilhados, o RTAI provê uma API para a utilização de semáforos, sendo que cada semáforo é tecnicamente associado a uma FIFO [13].

Os semáforos são ferramentas básicas para a sincronização interprocessos, utilizadas em sistemas operacionais. Os semáforos são contadores manipulados por tarefas ou processos ao alocar ou liberar um determinado recurso. Se um recurso no sistema possuir um semáforo e este não estiver disponível, os demais processos que desejarem fazer uso deste, devem ficar enfileirados aguardando que o recurso seja liberado e, por sua vez, o semáforo seja desbloqueado.

5.5.1.2 Memória Compartilhada

A Memória compartilhada provê um paradigma alternativo de IPC em relação às FIFOS, quando um modelo de comunicação diferente se fizer necessário. Memória compartilhada é um bloco comum de memória que pode ser lido ou escrito por qualquer processo ou tarefa no sistema [33]. Como processos diferentes podem operar de forma assíncrona uma área de memória compartilhada, surge a necessidade de assegurar que os dados na nesta região de memória não sejam sobrescritos, enquanto ainda se fazem necessários

ou mesmo sem que exista a real intenção de que tal ação seja tomada. Como o RTAI pode interromper uma tarefa ou processo em qualquer instante em que se fizer necessário, pode a vir a interromper uma ação de escrita ou leitura de um processo. Sendo assim se faz necessária a utilização de semáforos para garantir a exclusão mútua de um bloco de memória.

A Memória compartilhada é implementada como um controlador de dispositivo. Conseqüentemente, isto requer que uma entrada no diretório `/dev` seja adicionada antes que esta área de memória seja utilizada [33]. Ao alocar um bloco de memória compartilhada, o processo que esta alocando uma área de memória tem de atribuir um identificador a esta e informar o tamanho da mesma. A primeira chamada para um determinado identificador de bloco de memória irá fazer a alocação do mesmo, enquanto que as próximas chamadas ao mesmo identificador apenas devolvem o endereço do bloco de memória alocado anteriormente. Este identificador é um número inteiro, o que leva a um problema de alocação similar ao que esta presente nas FIFOS, caso não exista nenhum controle centralizado da distribuição deste identificador. Entretanto, assim como nas FIFOS, os identificadores de blocos de memória compartilhados podem ser alocados dinamicamente, utilizando-se de nomes simbólicos. Assim, a alocação conflitante de blocos pode ser evitada.

5.5.1.3 Caixas Postais (*Mailboxes*)

O RTAI fornece uma terceira alternativa para IPC através de um mecanismo mais flexível, as caixas postais (*mailbox*). Este dispositivo propicia que qualquer número de processos possam enviar e/ou receber mensagens de, ou para, uma caixa postal [33]. A quantidade de informações a serem armazenadas em uma caixas postal deve respeitar a sua capacidade de armazenamento.

Uma caixa postal que executa uma operação de envio ou recebimento pode ser associada a um temporizador. Isto significa que, caso uma operação de emissão ou de recepção não inicie ou termine em um determinado limite de tempo, o controle pode retornar ao processo ou tarefa que fez a solicitação. Além disso, o processo que acionou uma determinada caixa postal pode especificar que, se a caixa postal não possuir a capacidade de armazenar a mensagem inteira, somente uma parte dela seja enviada. Caso contrário, uma falha será gerada.

5.5.2 Gerenciamento de Memória

Nas primeiras versões do RTAI, a memória precisava ser alocada estaticamente, não sendo possível à alocação em tempo de execução. Atualmente, existe um módulo incluso voltado para a administração da memória, que permite a alocação dinâmica de blocos de memória pelas tarefas, mesmo em tempo de execução. A interface para tal alocação dinâmica é muito similar a da biblioteca C padrão.

O RTAI pré aloca blocos de memória antes da execução em modo tempo real (o tamanho e o número de blocos podem ser configurados). Quando uma tarefa de tempo real chama `rt_malloc()`, a memória solicitada é obtida destes blocos previamente alocados. Quando a quantia de memória livre for menor que o valor solicitado, um novo bloco de memória é reservado para atribuições futuras [13].

De forma similar, por ocasião da liberação de um bloco de memória alocado dinamicamente, através da função `rt_free()`, a memória liberada é agrupada ao bloco disponível para futuras alocações [33].

5.6 LXRT: Interface de Tempo Real para processos em modo usuário

A LXRT é uma API do RTAI com a finalidade de tornar possível o desenvolvimento de aplicações de tempo real em espaço usuário, sem a necessidade da criação de módulos agregados ao núcleo [35]. Isto é algo muito útil, pois utilizar módulos no núcleo para o desenvolvimento de tarefas de tempo real pode representar algumas desvantagens.

Inicialmente, a memória do núcleo não é protegida de acessos inválidos, por parte dos programas que rodam neste modo. A simples modificação de uma área de memória não desejada pode ocasionar a corrupção de dados e, até mesmo o mau funcionamento do sistema como um todo. Outra desvantagem é que, ao utilizarem programas executando como módulos, por ocasião de uma eventual atualização do núcleo do SO, estes módulos adicionados terão de ser recompilados. Portanto, não é possível que os binários de um programa de controle de tarefas de tempo real executadas como módulos possam ser simplesmente movidos entre versões distintas do núcleo.

O RTAI possibilita o desenvolvimento de uma aplicação de tempo real em espaço usuário, além de permitir que o programador faça uso das facilidades oferecidas pelas ferramentas de depuração oferecidas pelo Linux. Além disso, os serviços providos pelas chamadas de sistema no Linux permanecem disponíveis as estas tarefas. Após a aplicação estar plenamente desenvolvida, pode ser convertida para um módulo do núcleo como uma tarefa de tempo real crítico. Neste caso, as chamadas de sistema padrão no Linux deixam de estar disponíveis a esta tarefa de tempo real crítico, e os serviços anteriormente fornecidos pelo Linux devem então ser substituídos pelos providos pelo RTAI [13].

A LXRT permite que aplicações modifiquem dinamicamente seu modo de execução entre os modos de tempo real brando ou crítico, por utilizar apenas uma única chamada de função dentro do programa usuário. Quando uma aplicação estiver sendo executada como uma tarefa de tempo real brando, o escalonador padrão do Linux é utilizado. Contudo, para que uma tarefa tenha seu modo alterado tornando-se de tempo real é necessário que o processo altere sua política de escalonamento para a do tipo FIFO, que é o escalonador utilizado quando se intenciona que uma tarefa de tempo real critico seja executada no Linux [36].

O escalonador FIFO proporciona o uso de prioridades estáticas e escalonamento preemptivo melhorando, desta forma, o controle em relação ao escalonador padrão Linux, que utiliza prioridades dinâmicas. Assim, o escalonador FIFO permite melhorar o tempo de resposta de um processo em relação ao escalonador padrão do Linux. Contudo, os processos ainda podem perder seus prazos por vários motivos, como por exemplo, no caso de uma interrupção ser ocasionada por uma tarefa escalonada pelo RTAI.

A fim de possibilitar que uma tarefa altere para o modo de tempo real crítico, a LXRT cria um agente de tempo real no espaço do núcleo para cada processo rodando em modo usuário que possua exigências temporais críticas, sendo isto o que torna possível uma aplicação trocar de modo de execução. Para transferir o processo para o modo de tempo real crítico, o agente desabilita as interrupções, remove a tarefa da fila de execução do escalonador do Linux e o acrescenta na fila do escalonador do RTAI. A partir deste instante, esta tarefa deixará de ser perturbada ou interrompida por outros processos Linux.

Por ocasião desta troca de modo de operação, para o modo tempo real, o processo necessita ter certeza de que ele ainda permanece utilizando a mesma região de memória,

por conseguinte deve desabilitar a paginação de memória utilizando-se da chamada de sistema `mlockall()` [33, 10].

As chamadas de sistema do Linux não podem ser usadas no modo de tempo real crítico, mas esta restrição pode ser contornada através da utilização de um processo de tempo real brando que faz uso dos serviços do Linux em nome do processo de tempo real crítico. Os dois processos podem manter comunicação por se utilizarem das facilidades para IPC providas pelo RTAI [35]. Na utilização da API LXRT para desenvolver aplicações para tempo real críticas em modo usuário, os tempos de resposta não são tão bons para as tarefas rodando em modo usuário em relação a aquelas que estão rodando como módulos do núcleo. Em particular, a troca de contexto leva um tempo inferior a 100 μs ao se utilizar a LXRT, mas quando as tarefas estão operando como um módulo do núcleo, a troca de contexto leva um tempo inferior a 40 μs [33].

5.7 RTNET

O RTNET é uma pilha de protocolos de rede para tempo real crítico, fundamentada no *hardware Ethernet* utilizada pelo RTAI (extensão do SO Linux para manipular tarefas de tempo real).

5.7.1 Histórico

O projeto do RTNET teve início com David Schleef, no ano de 2000, baseando-se na série 2.2 do núcleo do Linux, RTAI e RTLINUX, sem um controle especial sobre a forma de acesso ao meio. Em 2001, o projeto foi adotado e passou a ser desenvolvido pelo Institute for Systems Engineering, Real-Time Systems Group, da Universidade de Hannover, passando, neste período, a ser portado e reescrito para as versões 2.4 do núcleo do Linux e para o RTAI-24.

No ano de 2002, Marc Leine Budde, deu início à modularização do projeto, sendo responsável, ainda, pela adição de um módulo de controle determinístico de acesso ao meio baseado no TDMA.

5.7.2 Fundamentos

O RTNET implementa a pilha de protocolos UDP/IP, ICMP e ARP, de forma determinística. Embora algumas estruturas de dados e algoritmos sejam derivados da pilha de rede do Linux, sua pilha de rede é proprietária e se baseia no hardware padrão *Ethernet*, provendo suporte a alguns dos mais populares *chipsets* de placas desta tecnologia [60, 28]. Na figura 5.3 é representada a estrutura básica dos componentes do RTnet.

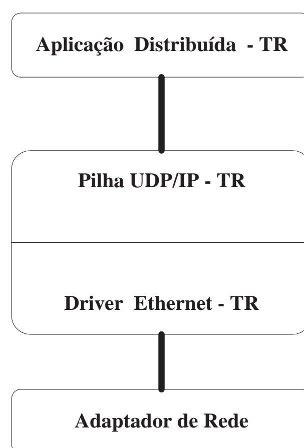

Figura 5.3: Estrutura básica do RTNET [60].

As mensagens que trafegam através do RTNET possuem as suas prioridades herdadas das tarefas de tempo real, que estão realizando as operações de envio e de recebimento de dados [11].

Para a resolução das colisões inerentes ao protocolo em que se baseia (*Ethernet*), foi implementada no RTNET uma camada adicional, denominada RTMAC, que é um protocolo de controle de acesso ao meio baseado no TDMA. O RTNET provê, ainda, uma API de *sockets* padrão BSD, que pode ser utilizada pelo o núcleo do RTAI, bem como pelos processos LXRT. A tabela 5.1 apresenta os requisitos mínimos para o funcionamento do RTNET.

Tabela 5.1: Requisitos para a utilização do RTNET [60]

Requisitos	**Versões**
Sistema Operacional	Núcleo do Linux versão 2.4.x
Patches de Tempo Real	RTAI 24.1.9 ou superior
Plataforma	Intel x86 (pelo menos para as extensões LXRT, a API do núcleo pode trabalhar com outras plataformas)
Adaptadores de Rede	Intel EtherExpress PRO 100, RealTek 8139, DEC 21x4x, 3Com EtherLink III

Como o RTNET utiliza um controle de acesso ao meio baseado em software, apenas é permitido que os nós que estejam em conformidade com o respectivo protocolo comuniquem-se dentro de um mesmo segmento físico de rede. Para as demais estações, que estiverem conectadas apenas via RTNET acessarem a Internet ou Intranet, o tráfego deve passar por um tunelamento através do domínio de tempo real [59]. Na figura 5.4 é representada configuração típica de uma rede RTNET.

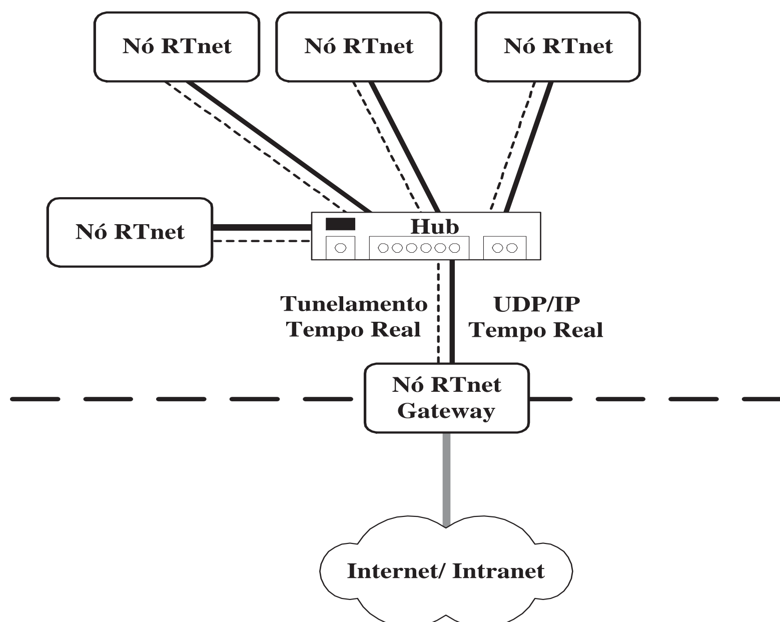

Figura 5.4: Rede RTNET [59].

5.7.3 Princípios de Implementação

O Gerenciamento das interfaces de rede *Ethernet* é efetuado pelos seus respectivos controladores de tempo real (*drivers*). Como o RTNET preserva amplamente o modelo de implementação de *drivers* de rede similar aos do Linux, todos os que estão disponíveis no RTNET foram portados de suas versões originais no Linux, tornando o custo de implementação relativamente baixo.

Atualmente, o RTNET suporta alguns dos principais dispositivos de rede, como Intel EtherExpress PRO 100, RealTek 8139, DEC 21x4x e 3Com EtherLink III. Os *chipsets* adequados para a implementação de tempo real são aqueles que não necessitam de longas fases de sincronização do hardware em suas rotinas de manipulação de interrupções e de

transmissão [28].

No RTNET, o controle de acesso ao meio, que é a permissão para iniciar a transmissão de pacotes pelo *driver* de rede, beseia-se no método CSMA/CD. A fim de prover um maior previsibilidade temporal nas comunicações, um módulo adicional denominado RTMAC foi adicionado ao projeto do RTNET.

Este módulo pode ser carregado opcionalmente e implementa um protocolo de acesso ao meio baseado no TDMA (*Time Division Multiple Access*), garantindo assim, determinismo temporal nas comunicações. Nas futuras versões do RTNET, se planeja a inclusão de alguns protocolos de acesso alternativos, tais como os baseados nos métodos de passagem de permissão [59].

5.8 RTMAC

O RTMAC é um módulo projetado para ser usado com o RTNET, com a finalidade de prover um controle determinista de Acesso ao Meio (MAC). A versão corrente do RTMAC implementa um algoritmo baseado no TDMA. Em virtude de projeto do RTMAC ser modular, novos algoritmos proprietários podem ser agregados ao seu módulo, mais facilmente. Na figura 5.5 está representada a estrutura básica do RTNET com o módulo RTMAC carregado.

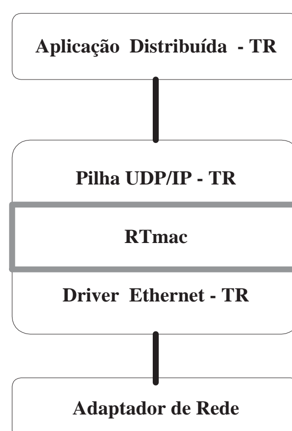

Figura 5.5: RTNET com o módulo RTMAC carregado [60].

Por ocasião da carga do módulo RTMAC, o controle exclusivo sobre a transmissão pelo meio físico passa a ser sua responsabilidade [28]. Todos os pacotes de saída passam

pelo RTMAC e, então, um algoritmo de acesso ao meio é quem decidirá quando os pacotes podem ser enviados pelo *driver* de Rede.

No RTMAC, cada estação possui um fatia de tempo fixa dentro do ciclo elementar TDMA. Uma das estações da rede deverá assumir o papel de mestre e, periodicamente, enviar um pacote de sincronização, com o objetivo de sinalizar o início de um novo ciclo, além de distribuir um *timestamp* global.

A figura 5.6 é uma representação de como o ciclo TDMA é composto dentro do RTMAC.

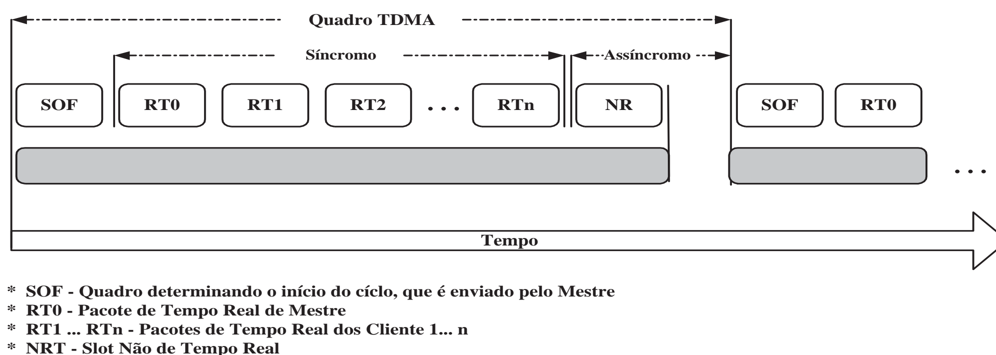

Figura 5.6: Ciclo TDMA no RTMAC [60].

Uma estação pode estar em um dos 3 estados básicos: inativa (*Idle*), mestre (*Master*) ou cliente (*Client*). Durante a carga do RTMAC, todas as estações na rede entram configuradas no estado inativo. Após o módulo RTMAC ser carregado, obrigatoriamente, uma das estações deve ser configurada como mestre. As outras estações, portanto, somente poderão ser configuradas como clientes. Todas as configurações relativas ao ciclo TDMA devem ser realizadas na estação mestra [28]. Os passos para a configuração do RTMAC são:

- Escolher a estação que irá assumir o papel de mestre.
- Trocar seu estado de inativo para mestre;
- Registrar a todos os clientes, antes de a rede ser utilizada. Novas estações clientes não podem ser adicionadas em tempo de Execução.

- Estipular o tempo do ciclo. A estação mestre deve gerar um quadro sinalizando o inicio de cada ciclo. O tempo de ciclo não pode ser alterado durante a execução.
- Determinar o tamanho do Pacote.
- Indicar o *offset* de envio para cada cliente (tempo entre a recepção do SOF e o inicio da fatia de tempo do cliente);

A figura 5.7 é uma representação gráfica da forma como são atribuidos os offsets dentro do ciclo RTMAC.

Figura 5.7: A Atribuição de *offsets* no ciclo RTMAC [60].

5.9 RTNETPROXY

O RTNETPROXY é um dispositivo de software, implementado no RTNET, que visa possibilitar o compartilhamento dos adaptadores de rede para tráfegos *Ethernet*, sejam eles de tempo real ou não, permitindo assim que a pilha TCP/IP do Linux possa ser utilizada através das aplicações LXRT no RTAI [59].

Do ponto de vista do Linux, o RTNETPROXY é visto como um dispositivo de rede virtual que possibilita acesso à rede RTNET para aplicações TCP/IP que necessitem fazer uso desta rede. Esta funcionalidade é denominada " RTPROXY". A figura 5.8 apresenta os componentes da pilha de protocolos do RTNET e a forma como estes se integram com a pilha de protocolos TCP/IP do Linux.

O RTNETPROXY é utilizado para compartilhar um adaptador de rede para tráfegos *Ethernet* de pacotes TCP/IP Linux, possibilitando que estes possam ser transmitidos através do RTNET [59]. Os pacotes derivados dos protocolos ICMP e UDP/IP são interpretados diretamente pela pilha de protocolos de tempo real RTNET, não sendo necessário que sejam repassados para o RTNETPROXY.

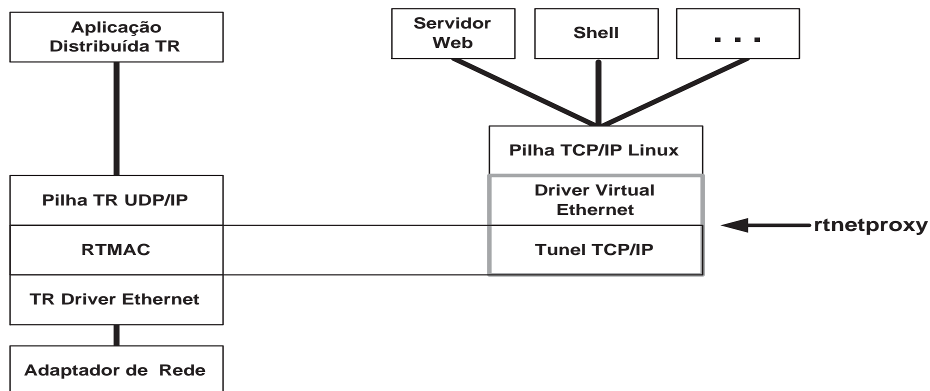

Figura 5.8: Integração com a pilha Linux TCP/IP [59].

Todos os dados a serem enviados ou recebidos são copiados do RTNET para o RTNETPROXY, debilitando assim, a performance em relação a *drivers* de rede do Linux. Todos os pacotes de entrada IPV4 tendo um identificador de protocolo IP, que não seja manipulados pelo RTNET, são repassados para o RTNETPROXY.

No capítulo seguinte serão abordados aspectos relativos ao projeto e a implementação do protocolo SRTP, alvo do presente trabalho

Capítulo 6

Projeto e Implementação do Protocolo SRTP

O SRTP é uma extensão do protocolo proposto em [8] adaptado para o contexto de sistemas de tempo real. Neste capítulo serão apresentadas as considerações a respeito do projeto do protocolo SRTP, através de sua especificação formal, bem como os detalhes relativos à implementação.

6.1 Especificação de Protocolos

A especificação de protocolos de comunicação de maneira formal traz várias garantias em termos de qualidade dos protocolos e correção dos mesmos, visto que a utilização de estruturas matemáticas bem definidas evita ambigüidade, são claras, simples e concisas, além de permitir a prova das propriedades do serviço implementado [18].

Neste trabalho, para a especificação do protocolo SRTP, foi utilizada a metodologia proposta em [18], que baseia-se em sistemas finitos de transição [3]. Esta metodologia divide o processo de desenvolvimento de protocolos em três fases básicas:

Ambiente de Aplicação e Requisitos: Nesta fase são definidos os objetos relevantes e a forma como estes se relacionam;

Projeto do Serviço e do Protocolo: O sistema a ser definido é particionado em módulos funcionais, serviços e protocolos. O objetivo desta fase é prover uma definição correta, sem ambigüidades à implementação, independente de plataformas alvo;

Implementação da especificação: Neste ponto o ambiente alvo é levado em consideração para a definição da implementação do protocolo.

No contexto do protocolo SRTP, considerou-se como premissas para o seu desenvolvimento, a não existência de conflitos tráfego na rede, em virtude da comunicação ser ponto a ponto (através da utilização de *Switches*), e a transmissão ocorrendo em modo *full-duplex* (ambos os sentidos ao mesmo tempo). O ambiente de execução é caracterizado como um sistema de tempo real, que somado aos canais ponto a ponto caracterizam um ambiente síncrono [56]. A especificação do protocolo é feita através de sistemas finitos de transição, que são uma forma geral de modelar o comportamento de um sistema.

Um sistema finito de transições é uma quádrupla $< S, s_0, E, \delta >$ onde:

- S é o conjunto dos estados do sistema;
- s_0 é o estado inicial ($s_0 \in S$);
- E é o conjunto de eventos. Divide-se o conjunto E em três conjuntos disjuntos: E_{in}, E_{out} e E_{int}. Estes conjuntos representam, respectivamente, os eventos de entrada (*input*), saída (*output*) e internos (*intern*). Eventos de entrada são representados pelo símbolo '?', os de saída pelo símbolo '!' e os internos 'τ';
- δ é a função de transição, que mapeia um estado e um

A função de transição geralmente é apresentada como um diagrama de estados, com eventos interligando os estados. A próxima seção apresenta os autômatos que definem o protocolo SRTP.

6.2 Definição do Protocolo SRTP

O protoco SRTP é divido em dois subsistemas, o transmissor T_{SRTP} e o receptor R_{SRTP}. A interação entre estes é realizada através de eventos que representam as ações de comunicação. Os subsistemas do protocolo SRTP e os eventos que os conectam são ilustrados na figura 6.1. Note que os eventos de entrada do sistema T_{SRTP} são os eventos de saída do sistema e R_{SRTP} e vice versa.

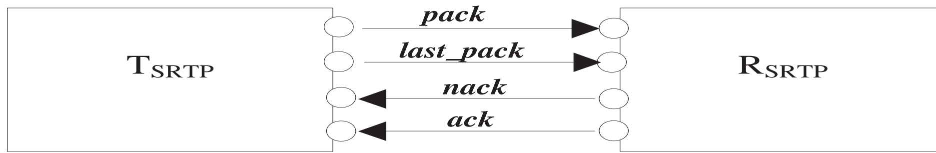

Figura 6.1: Sistemas SRTP.

Os subsistemas T_{SRTP} e R_{SRTP} são apresentados nas figuras 6.2 e 6.3 como sistemas finitos de transição (autômatos) e são definidos a seguir:

$T_{SRTP} =< \{T_1, T_2, T_3, T_4\}, T_1, \{\tau(send), pack!, last_pack!, nack?, ack?\}, \delta_T >$
$R_{SRTP} =< \{R_1, R_2, R_3, R_4\}, R_1, \{\tau(receive), \tau(deliver), nack!, ack!, pack?, last_pack?\}, \delta_R >$

A figura 6.2 apresenta a função de transição que representa o comportamento de um transmissor SRTP. O processamento se inicia quando alguma tarefa chama a primitiva ($\tau(send)$), para o envio de uma mensagem. O sistema particiona esta mensagem em uma seqüência de pacotes (estado T2) e os envia à tarefa receptora (estado T3 e evento $pack!$). O envio segue até o penúltimo pacote. Por ocasião do envio do último pacote (estado T3 e evento $last_pack!$) o sistema irá ficar a espera da confirmação do recebimento deste pacote por parte do receptor (T4 e evento $last_pack!$), permanecendo neste estado até que a confirmação da chegada do último pacote seja recebida (evento $ack?$). Caso contrário, periodicamente, o transmissor volta a enviar o último pacote ($pack!$).

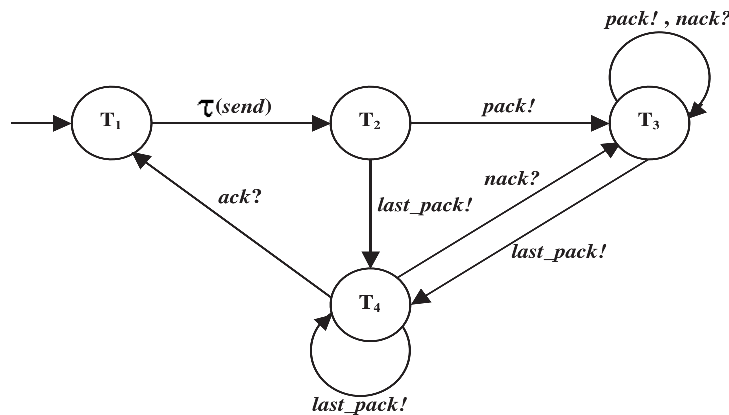

Figura 6.2: Função de transição para o subsistema transmissor δ_T.

Durante a fase de envio de pacotes, a qualquer instante pode ocorrer o recebimento de um pacote informando erros ($nack?$). Na ocorrência de tal transição o pacote solicitado será reenviado ($pack!$). Por ocasião do recebimento do último pacote por parte do receptor, o transmissor receberá um pacote ($ack!$) confirmando a chegada da mensagem. Neste ponto o ocorre a transição para o estado inicial (T1).

O comportamento do receptor SRTP é representado na figura 6.3 , que apresenta a sua função de transição. O processamento é iniciado quando uma tarefa chama a primitiva ($\tau(receive)$), dali em diante o sistema fica pronto para esperar pacotes. Note que, antes desta chamada de sistema, o protocolo ignora quaisquer pacotes recebidos (laço no estado R1). Ao receber os pacotes provenientes do transmissor o receptor permanecerá em um laço aguardando a chegada dos demais pacotes (R2 evento $pack?$). Após a chegada do último pacote (R2 evento $last_pack?$), o receptor enviará um pacote informando o recebimento com êxito da mensagem (R3 evento $ack!$) e liberando-a para a tarefa que requisitou a recepção (R4 evento ($\tau(deliver)$) e retornando ao estado inicial (R1). Durante a recepção, no caso da perda de seqüência ou de um pacote, será enviado ao transmissor um pacote informando o identificador do mesmo ($nack!$).

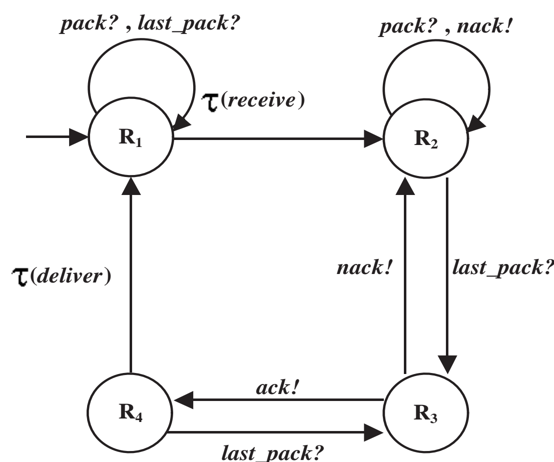

Figura 6.3: Função de transição para o subsistema receptor δ_R.

6.3 Implementação do SRTP

O desenvolvimento do protocolo SRTP, no RTNET, tem como o objetivo adicionar uma funcionalidade de comunicação para tarefas, em espaço usuário, que desejem efetuar transmissão e recepção de pacotes através da pilha RTNET. Para isto, foi implementado um módulo utilizado o mesmo mecanismo proposto em [8], cuja finalidade é tratar as rotinas de recepção e transmissão de mensagens. Foram, ainda, adicionadas duas novas chamadas de sistema, para que os programas de usuário possam vir a fazer uso das funcionalidades do mesmo.

6.4 Visão Geral do Protocolo

Como ambiente para o desenvolvimento do protocolo, utilizou-se a versão do núcleo 2.4.17 do Linux, o RTAI 24.1.10 (extensão do Linux para tempo real) e o RTNET 0.2.10. Durante a fase de implementação, seguindo a filosofia do projeto do RTAI, procurou-se efetuar o mínimo possível de alterações nos códigos fontes do Linux, RTAI e RTNET. Foram necessárias alterações no código fonte do Linux, pois, apesar de o RTNET ser uma pilha de rede proprietária, compartilha algumas estruturas e definições da pilha de rede padrão do Linux.

Para a implementação do protocolo utilizou-se a capacidade destes sistemas de trabalharem como módulos no núcleo, possibilitando, assim, que apenas uma pequena alteração na chamada de sistema `sys_ipc()` fosse realizada, visando a criação de uma interface com as tarefas LXRT [1] rodando em espaço usuário.

Foi necessária, ainda, a inserção de uma referência do protocolo SRTP junto ao sistema de rede RTAI-RTNET, a fim de possibilitar que, ao receber um quadro *Ethernet*, este fosse reconhecido como sendo um pacote `msg_srtp`. Assim, definiu-se o identificador 0x0717(ETH_P_RP), junto a *include if_ether.h*, com o objetivo de assinalar que os pacotes encapsulados em quadros *Ethernet* foram gerados e pertencem ao SRTP. Além disso, foi introduzido o cabeçalho SRTP junto à estrutura *rtskb* no RTNET.

[1] API do RTAI que possibilita o desenvolvimento de aplicações de tempo real em espaço usuário, sem a necessidade da criação de módulos agregados ao núcleo

O protocolo SRTP, inserido no RTNET, é implementado na formula de um módulo do núcleo e atua diretamente junto ao *driver* da placa de rede, sem entretanto, modificá-lo. Afim de prover uma interface com as aplicações LXRT, fez-se necessário a adição de duas novas chamadas de sistema para uso do protocolo SRTP:

1. `srtp_send(dest_task,dest_addr,buffer,buffer_size)`: Envia os dados contidos em `buffer` para a tarefa `dest_task` no nó identificado com o endereço `dest_addr`;

2. `srtp_receive(src_task,src_addr,buffer,buffer_size)`: Recebe os dados provenientes da tarefa `src_task` do nó `src_addr` e os coloca em `buffer`. Esta é uma chamada bloqueante, i.e. caso não existam dados a serem recebidos a tarefa fica bloqueada esperando até que estes cheguem.

Estas chamadas são responsáveis, respectivamente, pela transmissão e recepção de pacotes deste protocolo. Cada chamada de sistema obtém informações dos parâmetros passados, pelas tarefas LXRT, que identificam a tarefa remota (identificador da tarefa e endereço MAC da máquina remota), e a região de memória da tarefa local que armazenará os dados referentes à recepção ou a transmissão.

Com o objetivo de controlar os pacotes de transmissão e de recepção do SRTP, foi criada uma lista duplamente encadeada (`srtp_packt_list`) para manter as estruturas de mensagens `msg_srtp`, sendo estas, fragmentos (pedaços de mensagens) ou mensagens de controle do protocolo. A conteúdo desta estrutura é listado na tabela 6.1.

Tabela 6.1: Estrutura de controle de mensagens SRTP

Campos	**Descrição**
Endereços máquinas de destino e origem	Endereço MAC destino e Origem
Task_id origem e destino	Identificador das Tarefas destino e origem.
Fila de fragmentos (Fila de Recepção `rtskb`)	Lista que armazena os `rtskb`, que compõem uma mensagem para um determinado processo.
flag_info (Flag de Controle)	Controle de Pacotes Enviados ou Recebidos.
Número de Fragmentos	Identificador do número total de fragmentos que irão compor a mensagem.
Tamanho da Mensagem a ser transmitida ou recebida	Tamanho total da Mensagem.
Ponteiros para o bloco anterior ou posterior	Ponteiros da Lista duplamente encadeada.

A tabela 6.2 apresenta a descrição das principais funções do ambiente RTAI-RTNET utilizadas na implementação do protocolo.

Tabela 6.2: Funções RTAI-RTNET utilizadas na implementação

Função	**Descrição**
rtskb_queue_empty	Esta função retorna true se a fila estiver vazia, caso contrario retorna falso.
rt_printk	Rotina utilizada para escrever mensagens quando em modo núcleo.
rtskb_queue_tail	Enfileira um rtskb no inicio da lista.
rtskb_queue_purge	Remove todos os buffers de uma lista rtskb.
rtdev_get_by_index	Localiza uma interface pelo seu índice.
rt_get_time_ns	Retorna o tempo em nano segundos.
rtskb_dequeue	Remove um rtskb do inicio da lista.
rtdev_add_pack	Adiciona um manipulador de protocolo junto a pilha de rede do núcleo.
rtdev_xmit_if	Enfileira um buffer para transmissão por um dispositivo de rede.
rtskb_queue_head_init	Inicializa a fila dos rtskb.
alloc_rtskb	Aloca um novo rtskb.
kfree_rtskb	Desaloca um rtskb.
rtdev_remove_pack	Remove um manipulador de protocolos adicionado a lista de manipuladores de protocolos junto ao núcleo.
rt_schedule	Chamada para o escalodor de tarefas de tempo real.

O protocolo SRTP encontra-se localizado nas camadas de enlace e de aplicação do modelo de referência OSI. A localização do protocolo nessas camadas tem, apenas, o objetivo de evitar o *overhead* causado pelas camadas de rede e de transporte, uma vez que, no contexto de redes locais, as camadas intermediárias (rede, transporte, sessão e apresentação) são dispensáveis. Para o protocolo SRTP são necessárias apenas uma interface com o usuário (camada de aplicação) e uma camada que gerencie o *driver* de rede *Ethernet* na recepção e transmissão (camada de enlace). A figura 6.4 apresenta de maneira comparativa as pilhas de protocolos OSI, UDP/IP RTNET e SRTP RTNET.

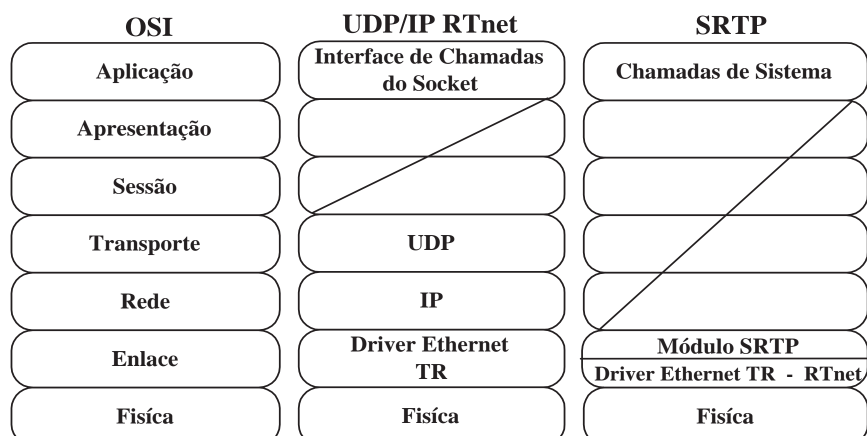

Figura 6.4: Comparativo entre as pilhas de protocolo

6.5 Componentes do protocolo SRTP

O protocolo SRTP é implementado a partir dos seguintes componentes:

- Rotina de transmissão;
- Rotina de recepção;
- Rotina de tratamento de interrupção na recepção e transmissão de pacotes (alteração).

6.5.1 Transmissão de Pacotes no SRTP

O protocolo SRTP implementa transmissão de pacotes através da chamada de sistema `srtp_send()`. Na transmissão, os pacotes gerados pelo protocolo são criados e encapsulados em quadros *Ethernet*, utilizando-se das informações contidas na estrutura de controle. As informações necessárias para o preenchimento, na estrutura de controle, podem ser obtidas de duas formas: considerando os parâmetros passados pela chamada `srtp_send()` como sendo dados (por valor) ou como um ponteiro (por referência). Caso a segunda opção seja adotada, torna-se necessário a utilização da rotina `copy_from_user()`, para realizar a transferência das informações entre os espaços de endereçamento do usuário e do núcleo.

Conforme [44], a cópia destes dados faz-se necessária em virtude de não se poder acessar diretamente o espaço do usuário, pois a memória no espaço de usuários pode estar em *swap* ou o ponteiro desta região de memória pode não ter uma página associada em memória física. Além disso, referenciar ponteiros, quando estes se encontram em espaço do usuário, possui algumas limitações, pois estes não podem ser referenciados ao todo no espaço do núcleo, em virtude de possuírem um mapeamento de memória diferente.

Após a estrutura de controle estar devidamente preenchida, a `msg_srtp` passa a ser montada e, então, é encapsulada em quadros *Ethernet* a partir da região de memória usuário. Caso a mensagem seja maior que a área de dados do pacote *Ethernet*, há a necessidade que esta seja dividida em pedaços menores, denominados fragmentos.

O processo envolvido na montagem das mensagens no protocolo SRTP inicia-se com a alocação de uma estrutura `rtskb`. Esta estrutura deve possuir tamanho suficiente para

armazenar o cabeçalho *Ethernet*, o cabeçalho SRTP e os dados a serem enviados. A figura 6.5 mostra o formato dos pacotes SRTP.

Figura 6.5: Formato dos pacotes SRTP.

A área reservada para dados no protocolo SRTP é de, no máximo, 1488 bytes, respeitando os limites de tamanho dos pacotes *Ethernet* (1500 bytes), e visto que 12 bytes são reservados para a inserção do cabeçalho SRTP. A transferência de dados do espaço usuário para o `rtskb` devem respeitar este limite.

Os pacotes são formados de acordo com a seguinte ordem: primeiro é alocada a área para os dados. Logo a seguir, o cabeçalho SRTP é anexado em uma posição na memória à frente dos dados e, em seguida, é inserido o cabeçalho *Ethernet* à frente do cabeçalho SRTP. A rotina responsável por colocar o cabeçalho *Ethernet* a frente do quadro é denominada `rtdev->hard_header()`.

Todos os fragmentos das mensagens devem conter o cabeçalho SRTP, com o identificador das tarefas origem e destino e o número de fragmentos que compõem a mensagem. As demais informações necessárias para o endereçamento da mensagem encontram-se no cabeçalho *Ethernet*, pois neste estão os endereços MAC das máquinas de destino e origem, e o identificador do protocolo SRTP. Somente de posse destas informações, é que a máquina destino conseguirá receber os quadros *Ethernet*, identificá-los como sendo do protocolo SRTP, e, desta forma, tratar a estes dados.

Após estas informações estarem agrupadas na estrutura `rtskb` é que o quadro *Ethernet* estará pronto para a transmissão. Na existência de um pacote montado, efetua-se a transmissão do mesmo por meio da fila de transmissão do dispositivo de rede. Neste ponto é invocada a rotina `rtdev_xmit_if()` do RTNET , com o objetivo de enfileirar o `rtskb` na fila de transmissão e processar o envio dos pacotes.

Os pacotes dessa fila serão transmitidos pela rotina de envio do *driver* de rede, até que a fila se esvazie ou até que informe uma sobrecarga. Neste ponto podem ocorrer falhas, caso a fila de transmissão esteja totalmente ocupada. Ao receber a informação de que os pacotes estão sendo descartados, em virtude da memória física do dispositivo (*buffer*

de transmissão) estar com sua capacidade esgotada, torna-se necessário que o SRTP suspenda a tarefa que está transmitindo, através de chamada à rotina de escalonamento. Desta forma possibilitando que alguma ação (tal como uma nova tentativa de processamento da fila de envio), seja executada.

A cada retorno da rotina de escalonamento, a função de transmissão irá tentar novamente inserir na fila o pacote que não pôde ser enfileirado. Para isto, é necessário que um novo `rtskb` seja montado. Esta operação deve ser realizada até que este fragmento da mensagem consiga ser enfileirado.

A cada pacote transmitido pela rotina de envio, uma pequena verificação deve ser realizada pelo nó transmissor no campo `flag_info` da estrutura `msg_srtp`, com o objetivo de analisar o status da transmissão. Desta forma é possível a constatação de que algum dos pacotes enviados não tenha sido devidamente processado pela máquina destino (pacotes perdidos ou quebra na seqüência), ou, ainda, se o envio da mensagem obteve sucesso.

A verificação do conteúdo deste campo é feita de modo que, se o transmissor encontrá-lo marcado com o valor 1, a transmissão obteve sucesso, sendo possível, desta forma, que um outro pacote possa ser transmitido. No caso da perda de um pacote, o campo `flag_info` estará marcado com um valor negativo, sendo necessário, portanto, que este pacote seja remontado e retransmitido. O valor 0 é utilizado para assinalar êxito na chegada de todos os pacotes ao receptor.

Neste último caso, a rotina de transmissão retorna para o final da chamada de sistema `srtp_send()`, retirando os componentes `msg_srtp` da lista de mensagens `srtp_pack_list` do módulo. Neste ponto, a chamada de sistema retornará ao código do usuário, pois a mensagem foi enviada com êxito.

O campo `flag_info` é marcado no transmissor pela *softirq* de recepção do SRTP, que, ao identificar um pacote de controle vindo da máquina receptora, executa a devida marcação deste *flag* da estrutura `srtp_pack_list` da tarefa que está transmitindo, possibilitando, assim, que alguma providência seja tomada.

Esse pacote de controle será transmitido pela máquina receptora, somente quando ocorrer alguma quebra na seqüência da recepção ou quando todos os pacotes chegaram sem problemas. Quando a rotina de transmissão chegar ao fim ficará em um *loop* de es-

pera chamando o escalonador de tempos em tempos e retransmitindo o último fragmento da mensagem, até que o `flag_info` contenha o valor zero, indicando que todos os pacotes chegaram ao destino. O algoritmo 1 representa de maneira simplificada a rotina de transmissão do protocolo SRTP.

Algoritmo 1 Rotina de Transmissão

1: $< p_1, p_2, ..., p_n > \leftarrow to_packets(m)$
2: $add_messages(< p1, p2, ..., pn >)$
3: $i \leftarrow 1$
4: **repeat**
5: $send(task_id, p_i)$
6: $i \leftarrow i + 1$
7: **until** $i < n$
8: **while** $!received_ack$ **do**
9: $send(task_id, p_n)$
10: $schedule()$
11: **end while**
12: {Recepção de Nacks.}
13: **when** $receive(nack(id, seq))$
14: $< p_1, p_2, ..., p_n > \leftarrow to_packets(m)$
15: $i \leftarrow seq$
16: **goto** linha 4
17: {Recepção de Acks.}
18: **when** $receive(ack(id))$
19: $recived_ack \leftarrow true$

6.5.2 Recepção de Pacotes no SRTP

O recebimento de pacotes `msg_srtp` fica separado em duas partes de código, uma executada em nível de chamada de sistema e a outra em nível de *softirq*. Uma *softirq* é uma rotina de software rotina que trata a interrupção propriamente dita.

Cada protocolo deve-se ter um registro de uma estrutura `packet_type`, com a finalidade de tornar possível o recebimento de pacotes para um tipo de protocolo dentro do núcleo. Conseqüentemente, para o recebimento de pacotes do protocolo SRTP, deve-se adicionar ao RTNET uma `packet_type` com a especificação SRTP.

Na recepção, além da chamada de sistema, o protocolo se estabelece como uma *softirq* de recebimento de pacotes. Ou seja, o driver de rede é o responsável pelo tratamento de interrupção, enquanto a *softirq* é a responsável pela análise do conteúdo dos pacotes *Ethernet*. No tratamento de interrupção, o *driver* gerencia o hardware de rede, recebe os quadros *Ethernet* e os repassa rapidamente para a *softirq* de recebimento de pacotes.

A *softirq* efetua toda a análise requerida para o tratamento do pacote recebido, como análise dos cabeçalhos e dos dados. Ao chegar um pacote do tipo `msg_srtp`, este será

tratado pela rotina de interrupção do *driver*, e, em seguida, a *softirq* de recepção analisará o cabeçalho do pacote e a rotina de recepção do SRTP será executada.

O objetivo principal da rotina de recepção é identificar a qual tarefa pertencem os dados recebidos nos pacotes. A rotina de recebimento de pacotes irá fazer uma busca nas estruturas `msg_srtp`, a procura da estrutura que se encaixa com a identificação do pacote.

Ao identificar a estrutura SRTP respectiva ao cabeçalho do pacote, a rotina de recepção verifica se o fragmento que chegou é o esperado, o próximo da seqüência. Caso o fragmento recebido não seja o esperado, o `flag_info` na máquina receptora é marcado para evitar que novos pacotes cheguem. Logo a seguir, a máquina receptora monta um único pacote de controle, a fim de informar a máquina transmissora qual foi o fragmento que foi perdido e deve ser reenviado.

Em virtude das características do ambiente, apenas um único pacote é enviado informando a perda, pois o meio físico, por ser *full-duplex*, sempre estará livre, e a máquina transmissora terá o *buffer* de recebimento da placa de rede livre para armazenar pelo menos um pacote até ser atendido. Desta maneira pode-se considerar que o envio do pacote de controle ao transmissor sempre terá êxito.

Quando um fragmento que chegou até a rotina de recepção é o esperado na seqüência, o `rtskb` desse fragmento é adicionado no final da fila de fragmentos. Essa fila está localizada internamente na estrutura de mensagens `msg_srtp` relativa ao cabeçalho desse pacote.

Por ocasião da chegada deste pacote de controle na máquina transmissora, esta deve, imediatamente, interromper o envio de pacotes, passando a remontar e reenviar os pacotes, a partir do fragmento perdido, à máquina receptora. Com a chegada do fragmento esperado à máquina receptora, o `flag_info` será desmarcado, assinalando a volta da seqüência correta no recebimento dos pacotes.

A rotina de recepção de pacotes da *softirq* ao identificar que o pacote recebido trata-se do último, envia uma mensagem para a máquina transmissora informando o sucesso na transmissão dos pacotes. Ao receber este pacote, a máquina transmissora identifica que a mensagem enviada foi processada pela máquina receptora, retornando da chamada de sistema de envio, totalmente ciente da chegada de todos os pacotes ao destino.

Na máquina transmissora, quando o último pacote é enviado, a rotina de transmissão ficará esperando um pacote originado pela máquina receptora, indicando o sucesso da transmissão. A máquina receptora ao perceber que se trata do último fragmento e que não houve quebra na seqüência do recebimento dos pacotes irá, portanto, criar um pacote de controle, enviando ao transmissor uma indicação de que todos os fragmentos chegaram com sucesso. Esse pacote fará com que o transmissor retorne da chamada de sistema ciente de que todos os pacotes chegaram ao destino. O algoritmo 2 representa de maneira simplificada a rotina de recepção do protocolo SRTP.

Algoritmo 2 Rotina de Recepção.

1: **when** $receive(pi)$
2: **if** i = 1 **then**
3: $send(nack(task_id, -1))$
4: $next \leftarrow 1$
5: **else**
6: $pack_list \leftarrow pi$
7: $next \leftarrow 2$
8: **else if** $i \neq next$ **then**
9: $send(nack(task_id, id, -next))$
10: **else**
11: $pack_list \leftarrow pack_list.pi$
12: **if** $pack_list = completo$ **then**
13: $send(ack(task_id))$
14: **end if**
15: **end if**

6.6 Controle de Erros no SRTP

Uma vez que o meio de transmissão foi considerado confiável, durante a implementação do SRTP, buscou-se minimizar a adição de *overhead* no protocolo. O aumento do *overhead* pode ocorrer em virtude da necessidade de se adicionar mecanismos que garantam uma transmissão confiável, em um meio não confiável. Sendo a leveza um fator preponderante na concepção do SRTP, implementou-se apenas um controle de erros de seqüência visando identificar pacotes perdidos durante o recebimento das mensagens. A figura 6.6 ilustra o fluxo normal do SRTP.

A quebra na seqüência dos pacotes, supondo que o meio não seja hostil, ocorre apenas quando o tratamento das interrupções de hardware não ocorrer a tempo, ou quando o limite do *buffer* da interface de rede estiver esgotado.

O controle de seqüência implementado é efetuado através de uma verificação do

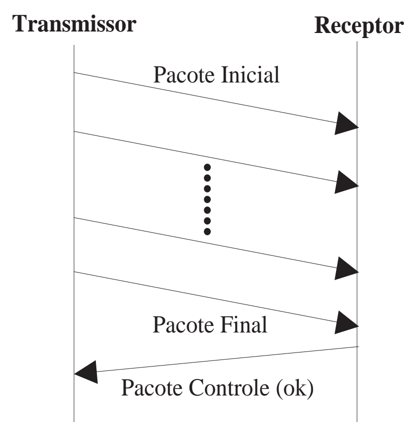

Figura 6.6: Fluxo normal do Protocolo SRTP.

número do fragmento do pacote. Caso ocorra uma quebra de seqüência, a *softirq* de recepção do SRTP na máquina receptora, enviará um pacote de controle ao transmissor informando qual o pacote que não foi processado corretamente e que deve ser retransmitido.

O pacote de controle sempre será atendido na máquina transmissora, em virtude do ambiente utilizado ser *full-duplex* e o *buffer* da interface de rede do transmissor sempre conseguirá armazenar esse pacote. A máquina transmissora ao receber o pacote de controle a partir da *softirq* do SRTP sinaliza a tarefa interrompendo a transmissão imediatamente, passando a gerar os fragmentos a partir do que foi perdido e a reenviá-los.

Já na máquina receptora, a *softirq* do SRTP, após ter enviado o pacote de controle reportando o erro, fica esperando pelo fragmento perdido. Ao chegar o fragmento esperado, a *softirq* volta à execução do fluxo normal. Caso o fragmento novamente não seja processado na máquina receptora, reenvia-se o pacote de controle para a máquina transmissora depois de certo período de tempo, até que o fragmento esperado chegue. A figura 6.7 exemplifica a procedimento executado, por ocasião da ocorrência de erros de recebimento no SRTP.

6.7 Condições de Corrida no módulo

A inserção de algumas proteções de acesso à estrutura `srtp_pack_list` e determinados pontos críticos do protocolo se fizeram necessários na fase de implementação do protocolo

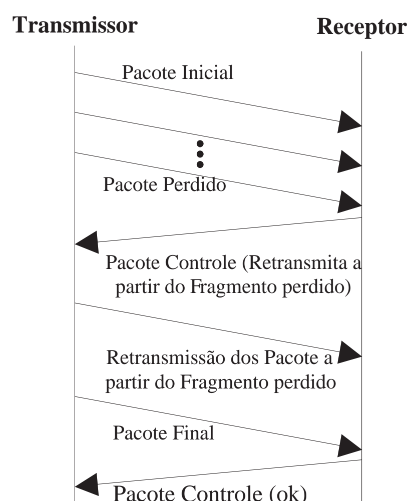

Figura 6.7: Fluxo com ocorrências de erros no Protocolo SRTP.

SRTP.

O principal motivo é o fato de que o código encontra-se dividido em duas partes: uma que manipula as interrupções e a outra as chamadas de sistema acessando, de forma assíncrona, uma região de dados (lista de mensagens `srtp_pack_list`).

No RTAI existem uma variedade de bloqueios que podem utilizados, levando em consideração a intensidade do bloqueio a ser realizado [13].

O objetivo destas proteções, é evitar a ocorrência de condições de corrida dentro do núcleo, pois estas podem corromper a lista de mensagens `srtp_pack_list`, uma vez que as operações efetuadas nesta lista não são atômicas .

Quando a chamada de sistema estiver manipulando a lista de mensagens `srtp_pack_list` uma interrupção da placa de rede pode ocorrer, indicando a chegada de um novo pacote. Após a realização do reconhecimento da interrupção pode ocorrer o processamento das *softirqs* pendentes.

Caso esse novo pacote seja do tipo SRTP, ele será processado pela rotina de recepção do SRTP podendo ser adicionado à lista da estruturas `msg_srtp` que já estava sendo acessada pela chamada de sistema. Para evitar o acesso de tarefas concorrentes, utiliza-se os dois tipos de proteção juntos através da rotina `rt_spin_lock_irqsave()`, impedindo que os códigos críticos sejam acessados simultaneamente por várias tarefas.

A rotina `rt_spin_lock_irqsave()` salva os *flags* e efetua um `hard_cli()`, se-

guido de um `rt_spin_lock()`, impedindo que ocorra alguma interrupção na CPU, evitando desta forma que outras tarefas acessem a mesma região de código.

A fim de desfazer os efeitos ocasionados pela chamada `rt_spin_lock_irqsave()`, a função `rt_spin_unlock_irqrestore()` é executada restaurando todos os textit-flags e, assim, retornando ao seu estado anterior, ou seja, com o *flags* de interrupção ativos.

A fim de evitar a perda de informações a respeito das interrupções, o código a ser executado nos *locks* deve ser rápido, pois o hardware da interface de rede armazena as interrupções que ocorrem durante este período, tratando as interrupções pendentes por ocasião da reativação dos *flags* de interrupção. Todavia, algo que sempre deve ser levado em consideração, é o fato de que o *buffer* do *hardware* de rede possui um limite.

Na seção seguinte serão apresentados os resultados e conclusões a respeito do trabalho. Serão abordados os testes realizados para validação do funcionamento do protocolo, bem como a avaliação de seus resultados.

Capítulo 7

Análise de Resultados

Nesta seção, pretende-se apresentar os testes realizados, na comparação entre os protocolos SRTP E UDP/IP. São apresentados, ainda os resultados obtidos, bem como uma análise dos mesmos.

Na realização dos testes foram utilizadas três diferentes faixas de carga de utilização do sistema e mensagens de sete tamanhos distintos. Assim foram medidos os tempos de transmissão através do algoritmo de *round trip*. As taxas de utilização do sistema tomadas como referência foram de 10 a 20%, 40 a 50% e 100%. Os tamanhos de mensagens utilizados foram de 500 Bytes, 1 Kbyte, 2 Kbytes, 5 Kbytes, 10 Kbytes, 20 Kbytes.

As amostragens utilizadas foram de 2000 mensagens para cada tamanho de pacote em cada faixa de carga, sendo coletadas em 20 experimentos de 100 pacotes cada. De posse dos dados referentes aos tempos de transmissão foram calculados os valores médios para cada tamanho de mensagem, em cada faixa de utilização.

Para a obtenção das cargas do sistema, foram utilizadas aplicações, cujas prioridades foram elevadas para que se tornassem processos de tempo real Linux (processos com prioridades elevadas). A faixa de 10 a 20% foi obtida com a utilização do modo gráfico e demais textitdaemons do sistema operacional. Para a obtenção de uma carga na faixa de 40 a 50% de utilização, foi acrescentada a utilização de um software de descompressão de áudio. A taxa de 100% de utilização foi obtida com a adição da execução de um software renderizador de imagens.

O cenário utilizado para a realização das experiências foi o de uma rede local composta por 3 máquinas conectadas através de um *Switch* 100 Mbits, o que possibilitou a não

existência de conflitos de tráfego na rede, em virtude da comunicação ser ponto a ponto, e a transmissão ocorrer em modo *full-duplex*.

Os dados foram obtidos através da comunicação entre 2 PCs, um AMD K6 266 MHZ com 192Mbytes de memória e um AMD K6 II 500 MHZ como 128 Mbytes de memória. O sistema instalado nas máquinas era o Linux versão do núcleo 2.4.17, o RTAI 24.1.10 e o RTNET 0.2.10. O *driver* utilizado para transmissão dos pacotes foi o 8139too-rt uma adaptação para tempo real do *driver* RealTek 8139, provido pelo Linux.

Foi utilizada a função *rdtsc()*, para a obtenção de tempos em uma grandeza de nanossegundos. Esta função constitui-se em um acesso direto ao contador *timestamp*, sendo composta por uma instrução em *assembly* que possibilita a leitura de tal contador [42]. Para obter o valor equivalente ao valor lido convertido em unidades de tempo é necessário o conhecimento, o mais preciso possível da freqüência do processador. Esta informação pode ser obtida no RTAI em /proc/rtai/scheduler e no Linux em /proc/cpuinfo. A função utilizada é representada na figura 7.1 a seguir:

```
static inline unsigned long long rdtsc() {
        unsigned long long x;
        \_\_asm\_\_ volatile(".byte 0x0f, 0x31":"=A"(x));
}
```

Figura 7.1: Função para a obtenção de tempos em nanossegundos [42].

A figura 7.2 representa, na forma de um gráfico, os valores médios obtidos nestas amostragens de tempos de transmissão.

Analisando os valores obtidos, pode-se verificar que, para uma taxa de utilização do sistema de 10 a 20%, os valores médios para o tempo de transmissão apresenta-se menor no SRTP do que no UDP/IP, com exceção das mensagens de tamanho iguais a 500 bytes e 20 Kbytes.

Para uma taxa de utilização do sistema de 40 a 50%, os valores médios de tempo de transmissão para o SRTP são maiores do que o UDP/IP para os pacotes com tamanho igual a 500 bytes, 1 Kbytes, 10kbytes e 20 Kbytes, sendo menores nos demais tamanhos de pacotes.

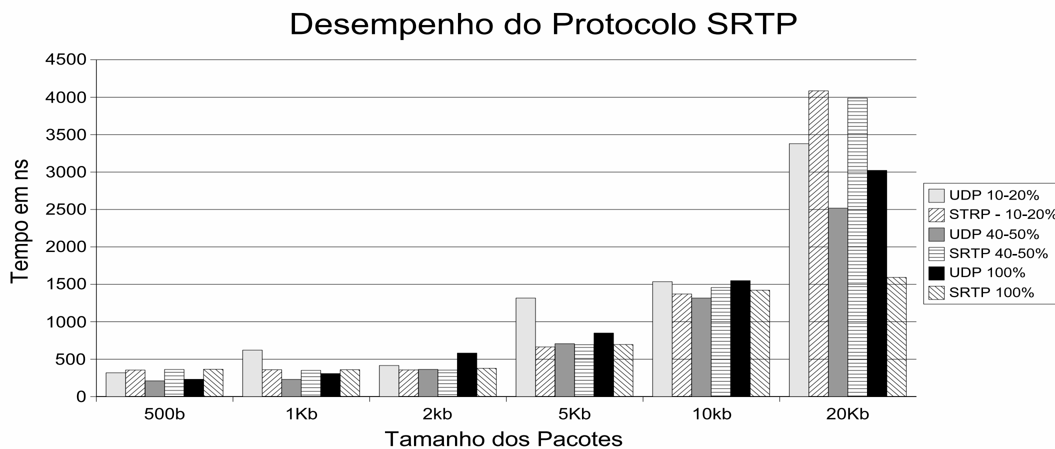

Figura 7.2: Valores médios obtidos.

Utilizando-se uma taxa de utilização do sistema de 100%, os valores médios para o tempo de transmissão do SRTP são maiores do que no UDP/IP apenas nas mensagens de 500 Bytes e 1 Kbytes, sendo menores nos demais tamanhos de mensagens.

Considerando-se a variação do tempo médio de transmissão de mensagens de tamanho igual em diferentes taxas de utilização do sistema, pode-se verificar que o protocolo UDP/IP apresenta um tempo médio de transmissão menor quando a carga no sistema é de 40 a 50% para todos os tamanhos de mensagens. O protocolo SRTP não apresenta uma taxa de utilização onde os tempos médios sejam sempre menores. Entretanto, seu comportamento revela uma estabilidade nos valores médios de tempo de transmissão nas diferentes cargas.

O comportamento de protocolo SRTP demonstrou-se anômalo na transmissão de mensagens com um tamanho igual a 20 kbytes, uma vez que o tempo médio de transmissão com carga de 100% do sistema apresentou-se em cerca de 40% do tempo de transição para cargas de 10 a 20% e 40 a 50%.

Pode-se constatar que apesar do SRTP em alguns casos demonstra um desempenho médio inferior ao do UDP/IP, mas este protocolo mostrou-se mais estável, apresentando um grau de variação inferior a este.

Foram realizadas diversas tentativas para comparar o protocolo SRTP com os protocolos UDP/IP, disponíveis no ambiente RTAI-RTNET, para a comunicação de tarefas em nível de usuário. Todavia, não foi possível efetuar tais testes, pois até mesmo as aplicações

de exemplo, fornecidas pelos desenvolvedores do ambiente, não obtiveram êxito em sua utilização. O funcionamento destes protocolos apenas foi possível para tarefas implementadas como módulo do núcleo, o que foge dos objetivos do trabalho.

A comparação de tempos de transmissão de aplicações implementadas como módulos do núcleo não foi efetuada, uma vez que é esperado um desempenho bastante superior, devido a fatores como a inexistência de troca de contexto e a não necessidade de transferência das informações entre os espaços de endereçamento do usuário para o núcleo.

Capítulo 8

Conclusão

Um sistema computadorizado de tempo real é um sistema em que o correto desempenho é avaliado não só pela resposta adequada a um dado estímulo, mas também se essa resposta é dada em tempo útil, isto é, quando ela é necessária.

Os sistemas de tempo real têm sido utilizados, dentre outras aplicações, na concepção de plantas industriais automatizadas, e em aplicações de controle. Na maioria dos casos, tais sistemas são compostos por vários nós, interligados por meio de uma rede de comunicações.

Visando garantir uma melhor performance temporal, na maioria dos sistemas de tempo real é utilizada uma arquitetura de softwares de comunicação baseada em apenas três camadas: camada física, camada de enlace e camada de aplicação.

Muitas pesquisas foram feitas sobre o desenvolvimento de sistemas operacionais proprietários, com o objetivo de prover suporte para o controle de tarefas de tempo real. Além disto, outras pesquisas tratam do desenvolvimento de extensões para sistemas operacionais de propósito geral, para que estes possam vir a trabalhar com tarefas com requisitos temporais. O Linux e suas extensões têm se mostrado alternativas viáveis e eficazes para tais tipos de sistemas.

Muitos estudos têm sido realizados, ainda, com o objetivo de otimização de redes *Ethernet*, padrão amplamente difundido e de baixo custo, afim de que estas venham a se adequar à transmissão de informações com requisitos temporais. No sistema de tempo real RTAI, a solução implementada para este propósito é a pilha de protocolos RTNET. Esta pilha provê acesso a dois tipos de tarefas de tempo real: as que estão rodando como

módulos agregados ao núcleo de tempo real e as tarefas que rodam como aplicações usuárias (LXRT).

A comunicação entre tarefas em nós distintos da rede ocorre através dos protocolos UDP/IP, implementados pelo RTNET. O SRTP é uma alternativa, proposta neste trabalho, para que as tarefas, em nível de usuário, possam comunicar-se entre máquinas distintas em uma rede RTNET.

Na implementação do protocolo SRTP, houve a preocupação com a confiabilidade nas comunicações. Foi implementada, para isto, uma rotina para o tratamento e recuperação de erros, que se dá em duas partes, uma delas no receptor, através da detecção da perda de pacotes e o envio de um sinal de controle e a segunda no transmissor, que, ao receber a informação de que quadros foram perdidos pelo receptor, passa a retransmití-los.

O SRTP procura manter uma complexidade baixa no algoritmo de recuperação de erros, pois quanto mais completo o algoritmo de recuperação de erros se apresentar, maior será o *overhead* no sistema de comunicação. Entretanto, o SRTP pode ser considerado confiável, uma vez que provê tal rotina em nível de protocolo.

A utilização da arquitetura provida nos `rtskb` permite uma maior flexibilidade nas rotinas que fazem a checagem dos cabeçalhos dos pacotes, evitando também cópias desnecessárias das informações. O protocolo implementado utiliza-se do mecanismo "de uma cópia", uma vez que as cópias de informações do buffer da placa de rede para a memória são efetuados por DMA pelo *driver*, sendo somente necessário fazer uma cópia por ocasião da transferência de informações entre espaço usuário e núcleo.

A implementação do protocolo SRTP demonstrou ser uma alternativa viável para a comunicação de tarefas de tempo real, em espaço usuário, apresentando confiabilidade e diminuição no *overhead*, em relação aos protocolos UDP/IP, uma vez que implementa uma rotina de detecção e recuperação de erros, e utiliza-se apenas de três camadas da pilha OSI. Os resultados obtidos na fase de testes indicam um funcionamento de maior estabilidade, em relação ao UDP/IP, apresentando menores variações de tempo de transmissão em relação à variação de carga do sistema.

Durante a fase de implementação, foram encontradas dificuldades em relação à falta de documentação do módulo RTNET, tornando mais dispendiosa a compreensão do seu funcionamento, bem como o desenvolvimento do módulo do protocolo. Por se tratar de

um objeto de pesquisa, seu código ainda não está completamente maturado, e apresenta um comportamento instável, em relação a algumas de suas funcionalidades.

Algumas sugestões podem ser feitas em relação aos possíveis estudos futuros, relacionados com a continuidade deste trabalho. Uma vez que a implementação desenvolvida tem o caráter de validação de um estudo, é possível que se obtenha uma maior otimização dos algoritmos utilizados, visando diminuir, ainda mais, os possíveis *overheads* existentes.

Quanto ao escopo de aplicação, pode-se sugerir a extensão do protocolo SRTP para o tratamento de comunicação de tarefas de tempo real implementadas como módulos do núcleo.

Uma vez que o protocolo desenvolvido não teve como objetivo prover determinismo temporal no envio de mensagens, é possível que seja adicionado tal atributo, através da integração do protocolo com o módulo RTMAC, ou, ainda, através da criação de uma aplicação de controle das comunicações, que possa vir a utilizar diferentes algoritmos para tal finalidade.

Referências Bibliográficas

[1] AIVAZIAN, T. Linux Kernel 2.4 Internals. [on-line] Disponível na Internet via WWW em 11/2002 : http://www.kernel.org/LDP/LDP/lki/, Jan. 2002.

[2] ANDERSON, J. H.; BARUAH, S. K. & JEFFAY, K. Parallel Switching in Connection-Oriented Networks. In *IEEE Real-Time Systems Symposium*, 1999, p. 200–209.

[3] ARNOLD, A. *Finite Transition Systems*. Prentice Hall, 1994.

[4] AUDSLEY, N. C.; BURNS, A.; RICHARDSON, M. F. & WELLINGS, A. J. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[5] BACON, J. & HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

[6] BARABANOV, M. & YODAIKEN, V. A Real-Time Linux. [on-line] Disponível na Internet via WWW em 01/2002: http://rtlinux.cs.nmt.edu/rtlinx/, 1996.

[7] BARANOV, M. *A Linux based Real-Time Operating System*. M.sc. thesis, New Mexico Institute of Mining and Technology, Socorro - New Mexico - USA, 1997.

[8] BARRETO, F. *Protocolo de Comunicação para Multicomputador*. Dissertação de mestrado, Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis - SC, 2002.

[9] BEAL, D.; RIPOLL, I.; ABENI, L.; GAI, P. & LANUSSE, A. Linux as a Real-Time Operating System. [on-line] Disponível na Internet via WWW em 01/2003 : http://www.metrowerks.com, May 2003.

[10] BIANCHI, E.; L.DOZIO & MANTEGAZA, P. RTAI *a Hard Real Time support for Linux.* DIAPM, Milano - Italy, 1999.

[11] BOCK, H.-P. RTNET Help. Comunicação pessoal com o autor em 13/08/2003, Aug. 2003.

[12] BOVET, D. P. & CESATI, M. *Understanding the Linux Kernel*, 1 ed. O'Reilly, 2000.

[13] BRUYNINCKX, H. Real-Time and Embedded Guide. Relatório técnico., K.U.Leuven - Department of Mechanical Engineering, Leuven - Belgium, 2000.

[14] BURNS, A. Scheduling hard real-time systems: A review. *Software Engineering Journal 6* (1991), 116–128.

[15] CLOUTIER, P.; MANTEGAZZA, P.; PAPACHARALAMBOUS, S.; SOANES, I.; HUGHES, S. & YAGHMOUR, K. DIAPM-RTAI Position paper. In *RTSS 2000 - Real Time Operating Systems Workshop*, Orlando - USA, Nov. 2000.

[16] DERTOUZOS, M. L. Control robotics: the procedural control of physical processes. In *IFIP Congress*, 1974, p. 807–813.

[17] DINKELL, W.; NIEHAUS, D.; FRISBIE, M. & WOLTERSDORFT, J. KURT LINUX User Manual, Mar. 2002.

[18] ERNBERG, P.; ÅKE FREDLUND, L.; HANSSON, H.; JONSSON, B.; ORAVA, F. & PEHRSON, B. Guidelines for specification and verification of communication protocols. Relatório técnico., Swedish Institute of Computer Science, January 1991.

[19] FARINES, J. M.; FRAGA, J. & OLIVEIRA, R. S. *Sistemas de Tempo Real.* IME-USP, 2000.

[20] GUIMARÃES, N. & NUNES, N. Sistemas de Tempo-Real, Dec. 2001.

[21] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. (ISO/IEC) 8802-4 Information Processing Systems - Local Area Networks - Part 4: Token Passing Bus Access Method and Physical Layer Specifications, 1990.

[22] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. (ISO/IEC) 8802-3 Information Processing Systems - Local Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, 1992.

[23] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. (ISO/IEC) 8802-5 Information Processing Systems - Local Area Networks - Part 5: Token Ring Access Method and Physical Layer Specification, 1992.

[24] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. (ISO/IEC) 7894-1 Information Technology - Opens Systems Interconection - Basic Reference Model, 1994.

[25] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. (ISO/IEC) 8802-2 Logical Link Control - Part 2, 1998.

[26] KALINSKY, D. & MICHAEL, B. Priority Inversion. In *Embedded Systems Programming*, Dec. 2002, p. 55–56.

[27] KIRRMAN, H. Fault Tolerance in Process Control an Overview and Examples of European Products. In *IEEE Micro*, 1987, p. 55–56.

[28] KISZKA, J. RTNET: Real-Time Networking for RTAI. [on-line] Disponível na Internet via WWW em 01/2002: http://www.rts.uni-hannover.de/rtnet/, Jan. 2003.

[29] KOEPTZ, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic, 1997.

[30] KOEPTZ, H. & VERISSIMO, P. *Real-Time and Dependability Concepts*. Addison Wesley, 1993.

[31] LANN, G. L. & RIVIERRE, N. Real-Time Communications over Brodcast Networks: the CSMA-DCR and the DOD-CSMA-CD Protocols. Relatório Técnico. 1863, INRIA, França, 1993.

[32] LEUNG, J. Y. & WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation 2* (1982).

[33] LINEO INC. DIAPM. RTAI programming guide. [on-line] Disponível na Internet via WWW em 12/2002: http://www.aero.polimi.it/projects/rtai, 2000.

[34] LIU, C. L. & LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* (Jan. 1973).

[35] MANTEGAZZA, P.; BIANCHI, E.; DOZIO, L. & PAPACHARALAMBOUS, S. RTAI: Real-Time Application Interface. Linux Journal, Apr. 2000.

[36] MAXWELL, S. *Kernel do Linux*. Makron Books, 2000.

[37] MOUROT, P. RTAI Internals Presentation. [on-line] Disponível na Internet via WWW em 11/2002: http://www.aero.polimi.it/ rtai/documentation/, 2002.

[38] NIEHAUS, D.; DINKEL, W. & HOUSE, S. B. Effective real-time system implementation with KURT LINUX. In *Real-Time Linux Workshop*, 1999.

[39] OLIVEIRA, R. S.; CARISSIMI, A. S. & TOSCANI, S. S. *Sistemas Operacionais*. Editora Sagra-Luzzato, 2001.

[40] OWENS, K. Introduction to Linux Kernel Modules. [on-line] Disponível na Internet via WWW em 01/2002: http://www.luv.asn.au/overheads/kernelmodules/, 2002.

[41] PANZIERI, F. & DAVOLI, R. Real Time Systems: A Tutorial. In *Performance Evaluation of Computer and Communication Systems*, 1993, vol. 729, p. 436–462.

[42] PICCIONI, C. A.; TATIBANA, C. Y. & OLIVEIRA, R. S. Trabalhando com o Tempo Real em Aplicações Sobre o Linux. Relatório técnico., Departamento de Automação e Sistemas - DAS - UFSC, Florianopolis - Brasil, 2001.

[43] RAMAMRITHAN, K. & STANKOVIC, J. A. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. In *Proceedings of the IEEE*, 1994, p. 55–67.

[44] RUBINI, A. & CORBET, J. *Linux Device Drivers*, 2 ed. O'Reilly, 2001.

[45] SHA, L.; RAJKUMAR, R. & LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers*, 1990, vol. 39.

[46] SILBERSCHATZ, A. E. & GALVIN, P. B. *Operating Systems Concepts*, 5 ed. JohnWiley and Sons, 1999.

[47] SILCOCK, J. & KUTTI, S. Taxonomy of Real-Time Scheduling, 1993.

[48] SOARES, L. F. G. & G.L SOUZA AND, S. C. *Redes de Computadores - Das LANs, MANs e WANs às redes ATM*. Editora Campus, 1995.

[49] SPURGEON, C. E. *Ethernet -The Definitive Guide*, 1 ed. O'Reilly, 2000.

[50] STANKOVIC, J. A. Misconceptions about Real-Time Computing: A Serious Problem for Next Generation Systems. *IEEE Computer 21*, 10 (Oct. 1988).

[51] STANKOVIC, J. A. Strategic Directions in Real-Time and Embedded Systems. *ACM Computinf Surveys 28*, 4 (Dec. 1996).

[52] STEMMER, M. R. Redes Locais Industriais. Notas de Aula, 2001.

[53] TANENBAUM, A. S. *Moderm Operating Systems*, 2 ed. Prentice Hall, 2002.

[54] TANENBAUM, A. S. *Computer Networks*, 4 ed. Prentice Hall, 2003.

[55] TATIBANA, C. Y. *Estudo experimental do Linux como Plataforma para Aplicações de Tempo Real Brando*. Dissertação de mestrado, Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis - SC, 2002.

[56] VERISSIMO, P. & RODRIGUES, L. *Distributed Systems for Systems Architects*. Kluwer Academic Publisher, 2000.

[57] WANG, Y.-C. & LIN, K.-J. Providing Real-Time Support in the Linux Kernel. [online] Disponível na Internet via WWW em 10/2002: http://linux.ece.uci.edu/RED-Linux/, 1999.

[58] WIRZENIUS, L.; OJA, J. & STAFFORD, S. The Linux System Administrator's Guide: Version 0.7. [on-line] Disponível na Internet via WWW em 01/2002: http://mirrors.kernel.org/ LDP/LDP/sag/index.html, 2002.

[59] WULF, O.; KISZKA, J. & WAGNER, B. A Compact Software Framework for Distributed Real-Time Computing. In *5th Real-Time Linux Workshop*, Valencia - Spain, Nov. 2003.

[60] YAGHMOUR, K. *Building Embedded Linux Systems*. O'Reilly, 2003.

[61] ZHENG, Q.; SHIN, K. & SHEN, G. C. Real-Time Communication in ATM Networks. In *19th Conference on Local Computer Networks*, Minnesota - USA, 1994, p. 156–164.

Glossário

ATM: *Asynchronous Transfer Mode* - Tecnologia de transmissão de qualquer tipo de informação (dados, voz, imagem e vídeo) em redes de computadores com taxas de velocidade que podem variar entre 2 Mbps até a faixa dos Gigabits.

Buffer: Área usada para armazenamento temporário de dados na memória do computador durante operações de entrada/saída.

Chipset: Conjunto de chips que controla as partes básicas do computador (memória, tráfego de dados, barramento e periféricos).

CSMA/CD: *Carrier Sense Multiple Access/ Collision Detection*- Procedimento de acesso no qual as estações envolvidas monitoram o tráfego em uma linha. Se não houver transmissão, a respectiva estação pode enviar informações. Quando as centrais tentam transmitir simultaneamente há uma colisão que é detectada por todas as centrais envolvidas. Ao término de um intervalo de tempo aleatório, os parceiros em colisão tentam a transmissão novamente. Se houver outra colisão, os intervalos de tempo de espera são gradualmente aumentados. As redes com procedimento CSMA/CD podem ser implementadas facilmente, mas não são determinantes no comportamento de transmissão. O procedimento CSMA/CD obedece a um padrão internacional pelo IEEE 802.3 e ISO 8802.3

Deadline: Um *deadline* representa o prazo ao fim do qual uma determinada tarefa deverá ser concluída.

Deadlock: Define-se *deadlock* como sendo uma situação de impasse, que ocorre quando cada processo em um conjunto de processos se encontra a espera de um acontecimento que só outro processo deste mesmo conjunto pode desencadear.

Driver: *Software* que controla um dispositivo, permitindo que sistema operacional de comunique com o dispositivo de *hardware*.

Ethernet: Padrão de rede IEEE, originalmente desenvolvido pela Xerox, para transmitir dados a 10 Mbps.

Full duplex: Um sistema de comunicação ou equipamento capaz de transmitir simultaneamente nos dois sentidos.

IP: *Internet Protocol*- Protocolo de funções básicas da Internet, responsável pelo roteamento de pacotes entre dois sistemas que utilizam a família de protocolos TCP/IP. É o mais importante dos protocolos em que a Internet é baseada.

Kernel: Núcleo do sistema operacional. O kernel controla praticamente tudo, gerencia e controla o acesso ao sistema de arquivos, gerencia a memória, a tabela de processos e o acesso aos dispositivos e periféricos, entre outras tarefas).

LAN: *Local Area Network* - As redes locais (LANs), são redes privadas com um alcance restrito a alguns quilômetros de extensão. Normalmente são usadas para interligar um prédio ou conjunto de prédios, como um campus universitário. Elas são amplamente usadas para conectar computadores pessoais e estações de trabalho em escritórios e instalações industriais, permitindo o compartilhamento de recursos e a troca de informações.

Linux: Sistema operacional multitarefa e multiusuário, criado por Linus Torvalds, que possui alto desempenho e pode rodar tanto em servidores como em computadores domésticos fornecendo um ambiente estável.

LXRT: API do RTAI para o desenvolvimento de aplicações de tempo real em espaço usuário, sem a necessidade da criação de módulos agregados ao núcleo.

MAC: *Media Access Control*- Nível 2 do modelo OSI. O nível de data-link responsável por planejar, transmitir e receber dados em uma LAN.

Módulo: No Linux, um módulo é uma coleção de rotinas que executam funções de sistema, podendo ser dinamicamente carregados e descarregados do kernel em

execução. Normalmente contém programas de controle de dispositivos e são bastante dependentes do kernel.

OSI: *Open System Interconnection Model* - Modelo conceitual de protocolo com sete camadas definido pela ISO, permitindo compreensão e o projeto de redes de computadores. Trata-se de uma padronização internacional para facilitar a comunicação entre computadores de diferentes fabricantes.

Overhead: Custo adicional em processamento ou armazenamento que, como conseqüência piora o desempenho de um programa ou de um dispositivo de processamento. Usado normalmente para se referir a custos adicionais indesejáveis, que deveriam ou poderiam ser evitados.

Processo: Também chamado muitas vezes de job ou tarefa, um processo pode ser considerado uma instância de um programa ou de um comando em execução em um sistema operacional.

Protocolo: Conjunto de regras que organizam e sincronizam a comunicação entre várias máquinas, tanto em nível de software como de hardware

rtskbuf: veja sk_buff.

Sistema Operacional: Processo que roda permanentemente em *background* e que permite efetuar as operações básicas do computador. As tarefas de um sistema operacional incluem a administração e o controle de acesso a todos os recursos específicos da máquina.

sk_buff: estrutura que contém um conjunto de ponteiros para uma única área contínua de memória, que é utilizada pelos drivers de rede do Linux para manipulação de pacotes. Esta estrutura permite uma maior flexibilidade na manipulação de analisadores e dos cabeçalhos dos pacotes, evitando cópias desnecessárias dos dados.

Softirq: Rotina de software rotina que trata uma interrupção.

Tarefa: Veja Processo.

TDMA: *Time Division Multiple Access* - Acesso Múltiplo por Divisão de Tempo - Forma de manter vários fluxos de informação independentes num mesmo canal de comunicação. São atribuidos pequenos intervalos de tempo cíclicos a cada fluxo de informação garantindo assim que estes não se misturam.

TCP: *Transmission Control Protocol* - é a parte da pilha de protocolos TCP/IP que controla o transporte de dados. É um protocolo do nível de transporte e é responsável pela partição e remontagem dos pacotes de dados, assegurando que os dados transmitidos alcancem seu destino, detectando e reenviando pacotes perdidos, adulterados ou duplicados, além de oferecer detecção e correção de erros.

UDP: *User Datagram Protocol*- Protocolo utilizado para o transporte sem conexão e baseado em IP. O UDP não oferece qualquer controle do fornecimento de datagrama.

WAN: *Wide Area Network* Rede que cobre uma grande área geográfica, podendo constituir-se de várias LANs interligadas.