CAPÍTULO 1

IMPLEMENTAÇÃO

1.1 INTRODUÇÃO

Neste capítulo, a implementação do protocolo de código aberto *DoRiS* é apresentada. A plataforma de tempo real escolhido para este desenvolvimento é o sistema operacional de propósito geral Linux, versão 2.6.19.7, dotado do *"patch"* Xenomai, versão 2.4-rc5 e do *nanokernel* Adeos correspondente. Esta plataforma, será simplesmente chamada "Xenomai" daqui para frente. Assim como foi visto no capítulo , Xenomai oferece garantias temporais da ordem de dezenas de micro-segundos, suficiente para a implementação de *DoRiS*. Além destas garantias temporais, duas outras motivações principais justificaram a escolha do Xenomai. Em primeiro lugar, Xenomai oferece uma interface de programação chamada RTDM (*"Real Time Driver Model"*) cujo o objetivo é unificar as interfaces disponíveis para programar controladores de dispositivos em plataformas de tempo real baseadas em Linux (KISZKA, 2005). O uso da interface RTDM garante portanto a portabilidade do protocolo nestes outros ambientes de tempo real. Em segundo lugar, Xenomai já dispõe de um serviço de comunicação Ethernet, baseado na interface RTDM. Esta camada, chamada RTnet (KISZKA et al., 2005), disponibiliza controladores de placas de rede portados para o Xenomai, assim como um protocolo de comunicação Ethernet baseado em TDMA. Portanto, o trabalho de implementação do protocolo *DoRiS* pôde aproveitar estes componentes e exemplos prontos, focalizando-se no desenvolvimento dos componentes específicos de *DoRiS*.

O presente capítulo é organizado da seguinte maneira. Inicialmente, uma descrição da pilha de rede do Linux é apresentada na seção 1.2. Em seguida, a camada de rede RTnet e sua interface RTDM com a plataforma Xenomai são descritos na seção 1.3. A seção 1.4 é dedicada à apresentação da implementação de *DoRiS* e da sua integração na camada RTnet do Xenomai. No final desta mesma seção, alguns resultados preliminares serão apresentados. Por fim, algumas conclusões serão discutidas na seção 1.5.

1.2 A CAMADA DE REDE DO LINUX

Esta seção apresenta apenas as camadas IP e Ethernet do *kernel* Linux, pois somente estas são modificadas pela implementação de *DoRiS*.

A camada de rede do Linux utiliza principalmente duas estruturas para armazenar os dados necessários à transmissão e recepção de pacotes. A primeira, chamada `net_device`, é asso-

ciado ao dispositivo da placa de rede. Esta estrutura utiliza apontadores de funções para definir a interface entre o *kernel* e as aplicações. Vale mencionar, por exemplo, as funções de definição do anel DMA (*"Direct Memory Access"*) utilizadas pela placa de rede para transferir os pacotes na memória. A implementação das funções da estrutura `net_device` pelos controladores de dispositivos é necessária para que o *kernel* possa disponibilizar os serviços associados a um hardware específico. A segunda estrutura, chamada `sk_buff` (de *"socket buffer"*), contém as informações associadas a um pacote de dados necessárias no decorrer do seu encaminhamento nas diferentes camadas do *kernel*. Dentre as principais informações, podem ser citadas a área da memória onde os dados estão armazenados, as eventuais informações de fragmentação, e os diferentes cabeçalhos do pacote.

Quando um pacote é recebido na memória tampão da placa de rede, o dispositivo copia o pacote no anel DMA previsto para este efeito. Por fins de otimização, o dispositivo pode ser configurado para operar com vários pacotes, invés de um. Tal procedimento não muda o modo de operação, pois tratar vários pacotes de uma vez só é equivalente a tratar um pacote de tamanho maior. Portanto, ilustra-se aqui o processo de recepção com um pacote só. Logo que o pacote se torna disponível no anel DMA, o *kernel* precisa ser informado da sua presencia e da necessidade de processá-lo. Para este efeito, duas abordagens principais devem ser mencionadas.

A primeira é baseada em interrupções do processador. Assim que ele termina a operação de DMA, o dispositivo da placa de rede interrompe o processador para informá-lo da presencia do pacote à ser recebido. No tratador desta interrupção, o *kernel* salva as informações relevantes para localizar o pacote e cria um `softirq` (ver seção) para executar as demais operações necessárias. Antes de retornar, o tratador habilita as interrupções novamente para permitir a recepção de um novo pacote. Na ausência de uma nova interrupção, o `softirq` é escalonado imediatamente e executa basicamente as três operações seguinte: (1) alocação dinâmica do espaço de memória de um `sk_buff`; (2) copia do pacote armazenado no anel DMA neste `sk_buff`; e (3) encaminhamento do `sk_buff` (via apontadores) para a camada IP.

Esta abordagem tem a seguinte limitação. Quando a taxa de chegada de pacote chegue a um certo patamar, a chegada de um novo pacote acontece antes que o tratador do pacote anterior termina de executar. A medida que este cenário se repete, a fila de interrupção em espera aumenta, resultando num situação de *"livelock"*. Pois o tratamento das interrupções tem a maior prioridade no sistema, o processador fica monopolizado sem que haja possibilidade alguma de executar os `softirqs` escalonados ou qualquer outro processo. Em algum momento, o anel DMA fica cheio e novos pacotes chegando são descartados.

A segunda abordagem, que resolve o problema do *"livelock"*, é o método chamado de consulta (*"polling"*), no qual o *kernel* consulta periodicamente o dispositivo da placa de rede para saber se tiver algum pacote em espera para ser recebido. Se for o caso, o *kernel* processa parte ou todos dos pacotes que tiverem esperando. Este segundo método tem a vantagem de suprimir

as interrupções do processador pela placa de rede. No entanto, a sua utilização introduz uma sobrecarga inútil do processador quando não há pacote chegando na placa de rede. Além disso, este método introduz uma certa latência para o tratamento dos pacotes. No pior caso, um pacote chegue logo depois da consulta da placa de rede pelo processador. Neste caso, o pacote só será processado depois de um período de consulta.

Para evitar os defeitos e aproveitar das vantagens de ambos estes métodos, o *kernel* utiliza uma solução híbrida proposta por (SALIM; OLSSON; KUZNETSOV, 2001). Esta solução, chamada NAPI (Nova API), utiliza a possibilidade que o *kernel* tem de desabilitar as interrupções de maneira seletiva, isto é, de um dispositivo específico só. Na chegada de um pacote, a seguinte seqüência de eventos é executada.

i) O dispositivo copiá o pacote no anel DMA. Na ausência de espaço neste anel, o pacote pode ser tanto descartado quanto copiado no lugar do mais velho pacote já presente no anel.

ii) Se as interrupções da placa de rede for habilitadas, o dispositivo interrompe o processador para informá-lo que há pacote em espera no anel DMA. Senão, o dispositivo continua a receber pacote e transferi-los para o anel DMA, sem interromper o processador.

iii) Na ocorrência de um interrupção da placa de rede, o tratador comece por desabilitar as interrupções proveniente deste dispositivo, antes de agendar um `softirq` para consultar a placa de rede.

iv) Em algum momento futuro, o *kernel* escalona o `softirq` de consulta da placa de rede e processa parte ou todos os pacotes esperando no anel DMA. Se o número de pacotes para serem processados é maior que o limite configurado, o `softirq` agenda-se para processa-los ulteriormente. Em seguida, o *kernel* executa eventuais outros `softirqs` em espera, antes de escalonar o `softirq` de consulta da placa de rede novamente.

v) Quando o anel DMA não contem mais nenhum pacote, quer seja porque eles foram processados ou porque foram silenciosamente descartados, o `softirq` habilita as interrupções da placa de rede novamente antes de retornar.

Como pode-ser visto, esta solução impede o cenário de *"livelock"* graças a desabilitação seletiva das interrupções. Por outro lado, a consulta da placa de rede só acontece quando pelo menos um pacote chegou, evitando portanto a sobrecarga inútil do processador.

Do ponto de visto das latências, esta solução tem os seguintes gargalhos. O `softirq` de consulta pode ser escalonado depois de um tempo não previsível, na ocorrência de outras interrupções causadas por qualquer outro dispositivo. Durante a sua execução, este `softirq` deve alocar dinamicamente os espaços de memórias necessários (`sk_buff`) ara armazenar os

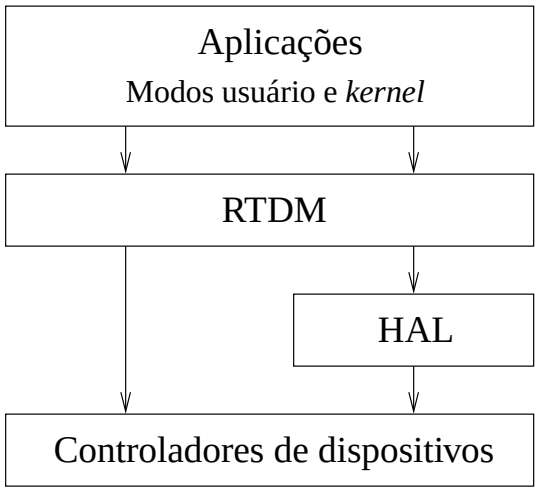

Figura 1.1: A interface do RTDM (reproduzida de (KISZKA, 2005))

pacotes. Esta alocação pode também levar um tempo imprevisível, pois a chamada sistema `malloc` pode falhar, na ausência de memória disponível.

Ver-se-á nas seções a seguir como estes problemas são resolvidos pela camada RTnet da plataforma Xenomai.

1.3 A CAMADA DE REDE DO XENOMAI: RTNET

1.3.1 RTDM

Antes de apresentar o projeto RTnet em detalhes, precisa-se descrever brevemente a interface de programação na qual ele se baseia, isto é o RTDM (*"Real Time Driver Model"*) (KISZKA, 2005), que por sinal, vem sendo desenvolvido pelo próprio Jan Kiszka, principal autor do RTnet.

A API *"Real Time Driver Model"* tem por objetivo oferecer uma interface de programação unificada para sistemas operacionais de tempo real baseados em Linux. O RTDM constitua uma camada de software que estende os controladores de dispositivos e a camada de abstração do hardware (HAL) para disponibilizar serviços à camada de aplicação, conforme representado na figura 1.1. O RTDM foi inicialmente especificado e desenvolvido na plataforma Xenomai. No entanto, em visto dos benefícios trazidos por esta API, ela também foi adotada pelo RTAI.

Escolheu-se neste trabalho de utilizar a interface do RTDM, de tal forma que os produtos de software resultando da implementação de *DoRiS* possam ser utilizado em ambas plataformas.

A interface do RTDM oferecida para as aplicações pode ser divida em dois conjuntos de funções. Aquelas que dão suporte aos dispositivos nomeados e aos dispositivos de protocolos

- as duas classes de dispositivos mais utilizadas em sistemas com requisitos temporais - e as outras que constituem os serviços de tempo real básicos, independentes do hardware.

Em relação ao primeiro conjunto, o RTDM segue o modelo de entrada e saída e o padrão de comunicação via *"socket"* do padrão POSIX 1003.1 (IEEE, 2004). Os dispositivos de entrada e saída, também chamados de dispositivos nomeados, disponibilizam as suas funcionalidades através de um arquivo especial no diretório `/dev`. Enquanto os dispositivos associados a protocolos, dedicados a troca de mensagens, eles registram os seus serviços através da implementação de um conjunto de funções definidas pelo RTDM na estrutura `rtdm_device`. Exemplos de algumas destas funções são `socket, bind, connect, send_mesg, recv_msg`. Para diferenciar tal função das funções usual do *kernel*, o sufixo `_rt` ou `_nrt` é adicionado ao seu nome. Do ponto de visto das aplicações, a API do RTDM utiliza os nomes do padrão POSIX com o prefixo `rt_dev_`. Desta forma, a correspondência entre uma chamada e a função para ser executada é realizada pelo Xenomai, de acordo com o contexto no qual a função é chamada. Por exemplo, se o contexto for de tempo real, a chamada `rt_dev_sendmsg` levará a execução da função `sendmsg_rt`. No caso contrário, a função `sendmsg_nrt` será executada.

O segundo conjunto de funções diz respeito a abstração dos serviços do *nanokernel* de tempo real. Elas contemplam notadamente os serviços de relógios de alta precisão e de temporizadores associados, as operações associadas ao gerenciamento de tarefas, os serviços de sincronização e de gerenciamento das linhas de interrupções, e um serviço de sinalização para permitir a comunicação entre os diferentes domínios registrados no *"ipipe"*. Além destes serviços essências, vários outros utilitários são disponibilizado tal que, por exemplo, a alocação dinâmica de memória e o acesso seguro ao espaço de memória usuário.

Deve ser observado que a API do RTDM tende a crescer rapidamente. Numa consulta do projeto Xenomai (Xenomai, 2008) realizada em dezembro de 2008, enumerou-se um pouco mais de 100 funções definidas por esta API.

1.3.2 A arquitetura do RTnet

O projeto de código aberto RTnet (KISZKA et al., 2005) foi fundado em 2001 na Universidade de Hannover com o objetivo de prover uma infraestrutura flexível e independente do hardware para serviços de comunicação de tempo real baseados em Ethernet. O RTnet também disponibiliza um serviço de comunicação baseado em *"firewire"*, mas este assunto não será discutido aqui, pois ele foge do escopo deste trabalho. Desenvolvido inicialmente na plataforma de tempo real RTAI, este projeto é também disponível na plataforma Xenomai.

A localização do RTnet e das suas relações com as demais camadas do Xenomai é representada na figura 1.2. Como pode ser observado, RTnet se baseia em dispositivos de hardware existentes e introduz uma camada de software para aumentar o determinismo dos serviços de

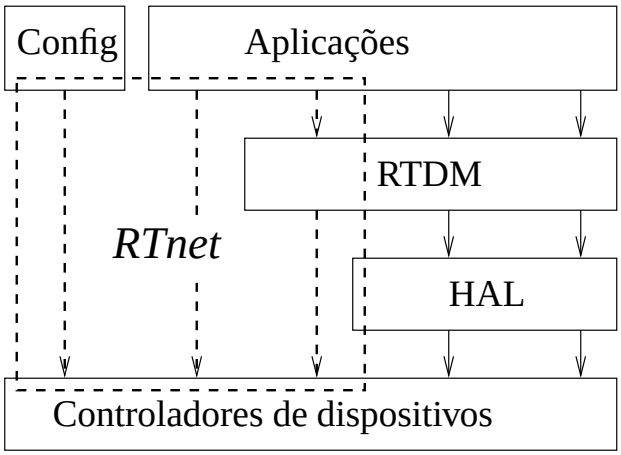

Figura 1.2: Localização do RTnet.

comunicação. Do ponto de vista da sua interface com as aplicações, RTnet utiliza o RTDM (*"Real Time Driver Model"*) (KISZKA, 2005), cuja descrição encontra-se na seção 1.3.1. Em relação ao hardware, RTnet utiliza tanto a camada HAL provida pelo Xenomai quanto controladores de dispositivos próprios. Para a implementação de tais controladores, capaz de prover garantias temporais, o código original dos controladores de dispositivos do Linux é modificado, conforme a descrição disponível na documentação do RTnet (RTnet, 2008). O objetivo principal destas modificações é remover do código do dispositivo qualquer chamada as funções bloqueantes do *kernel* Linux.

O detalhe dos componentes da pilha RTnet é apresentada na figura 1.3. Como pode ser observado, RTnet segue a organização em camada da pilha de rede UDP/IP do Linux. No entanto, a implementação atual do RTnet só oferece serviços de comunicação baseados em UDP/IP e não da suporte ao protocolo TCP/IP.

Acompanhando a figura 1.3 de cima para baixo, veja-se que os serviços oferecidos pelo RTnet utilizam a interface do RTDM para disponibilizar as suas funcionalidades às aplicações. Uma interface específica de configuração, através de funções `ioctl`, é utilizada para as operações de gerenciamento. Encaixado na interface de baixo nível do RTDM, os componentes da camada de rede provém implementações específicas dos protocolos UDP, ICMP, e ARP. Ver-se-á na seção 1.3.3.3 algumas das soluções adotadas para aumentar o determinismo dos protocolos UDP e ARP. O protocolo ICMP, baseado no protocolo IP, é dedicado à funções de controle e gerenciamento da rede. A sua implementação é bastante específica e de pouca relevância para este trabalho, ela portanto não será apresentada.

Os componentes principais do RTnet são localizados abaixo da camada de rede e acima dos controladores de dispositivos e da camada HAL. Eles constituem a camada RTmac e o núcleo RTnet mencionados na figura 1.3. O núcleo contém os serviços de emissão e recepção dos pacotes, baseados nos controladores de dispositivos modificados. Detalhes desta implementa-

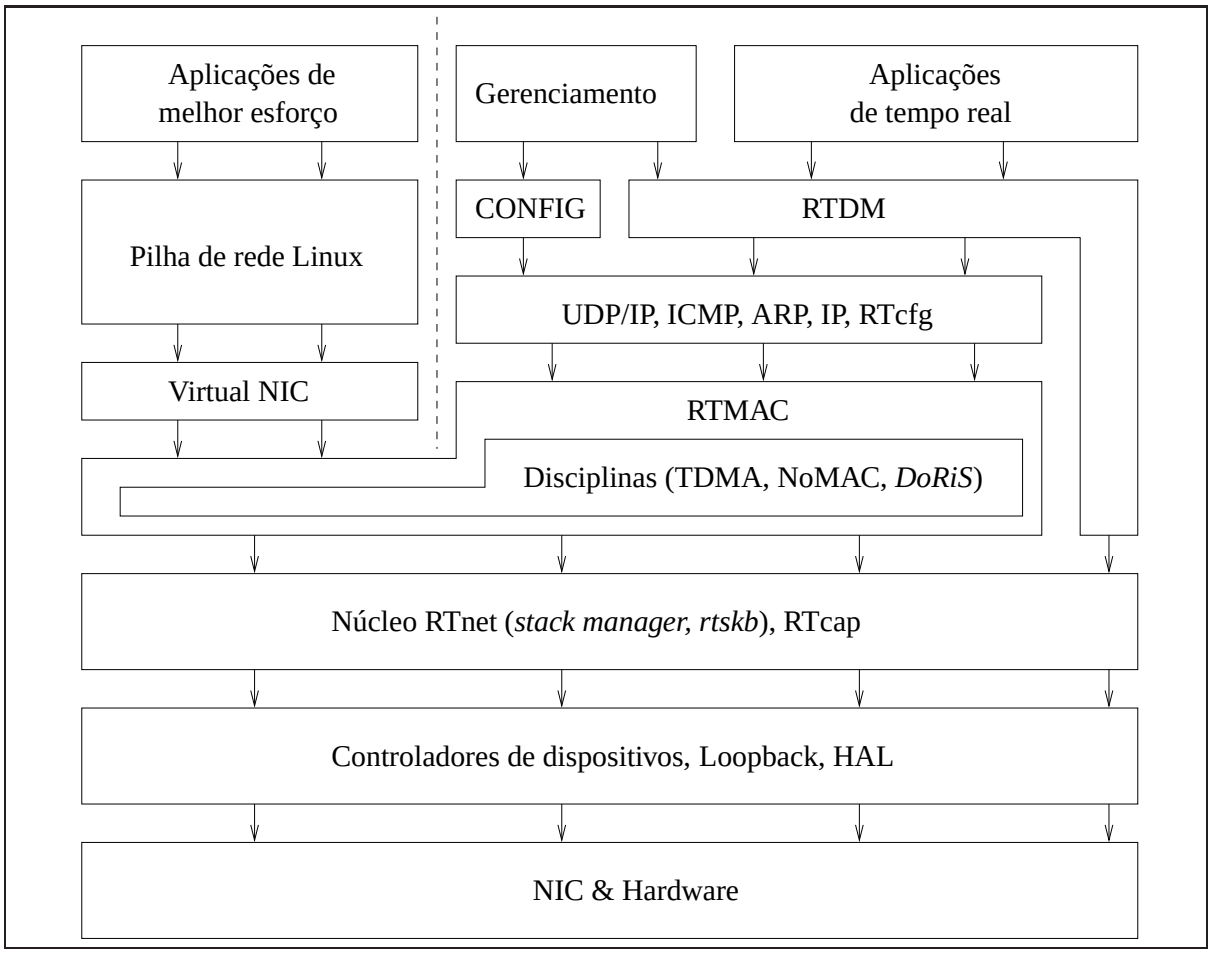

Figura 1.3: Estrutura em camada do RTnet.

ção serão descritos na seção 1.3.3.1. Em relação ao RTmac, ele constitui uma caixa pronta na qual as disciplinas de acesso ao meio são embutidas. Através da estrutura `rtmac_disc`, a interface RTmac indica as funções cuja a implementação é necessária para definir uma política de acesso ao meio. Na versão atual do RTnet, as duas disciplinas TDMA e NoMAC são disponíveis. Elas serão brevemente apresentadas, assim que a interface RTmac, na seção 1.3.4. O presente trabalho de implementação teve por resultado a criação de uma nova disciplina de acesso ao meio, *DoRiS*, além das duas disciplinas já existentes. Uma descrição detalhada da implementação de *DoRiS* será realizada na seção 1.4.

Além das destes componentes essências, Rtnet providencia também serviços de configuração (`RTcfg`) e de monitoramento (`RTcap`). Os primeiros servem tanto para configurar a rede em tempo de projeto, quanto para modificar a composição dos membros da rede te tempo real em tempo de execução. Os segundos permitem de coletar dados temporais sobre os pacotes transmitidos. Estes serviços sendo opcional, eles não serão descritos aqui.

Ao lado da pilha RTnet, aplicações que utilizam os serviços de melhor esforço do da pilha de rede do Linux podem fazê-lo através de interfaces virtuais, cujo mecanismo será apresentado na seção 1.3.3.4, juntamente com a descrição do formato dos pacotes RTnet.

1.3.3 RTnet: principais componentes

1.3.3.1 Gerenciamento de memória Foi visto na seção 1.2 que a camada de rede do Linux aloca dinamicamente um espaço de memória `sk_buff` para armazenar um pacote durante a sua existência na pilha de rede. Devido ao gerenciamento virtual da memória, este pedido de alocação de memória pode resultar numa falta de página. Neste caso, o processo pedindo memória é suspenso e o *kernel* escalona o *"thread"* responsável do gerenciamento da memória virtual para que ele libere algumas páginas inutilizadas no momento. Este procedimento pode levar um tempo imprevisível, ou mesmo falhar em alguma situação específica.

Para evitar esta fonte de latência não determinística, RTnet utiliza um mecanismo de alocação estática da memória em tempo de configuração. Nesta fase inicial, cada um dos componentes da pilha participando dos processos de emissão ou recepção deve criar uma reserva de estruturas `rtskb`. Tal estrutura, similar a estrutura `sk_buff` do Linux padrão, é utilizada para armazenar as informações e os dados de um pacote, desde sua chegada no anel DMA de recepção, até sua entrega à aplicação de destino. Cada `rtskb` tem um tamanho fixo, suficiente para armazenar um pacote de tamanho máximo. Para garantir a permanência das reservas de `rtskb` de cada componente da comunicação, um mecanismo de troca é utilizado. Ou seja, para poder adquirir um `rtskb`de um componente A, um componente B deve dispor de um `rtskb`livre para dar em troca. Observa-se que, como as estruturas são passadas por apontadores, tais operações de troca são quase instantânea em comparação ao tempo que levaria a copia do conteúdo destas estruturas.

1.3.3.2 Emissão e recepção de pacotes A emissão de um pacote acontece no contexto síncrono de uma tarefa e as operações subseqüente são executados com a prioridade desta tarefa. Portanto, as garantias temporais associada a uma emissão dependem exclusivamente da plataforma operacional e da utilização correta dos seus serviços.

No caso da recepção de uma mensagem, a situação é diferente, pois é um evento assíncrono. Em particular, não se sabe, no início do procedimento de recepção, qual é a prioridade da aplicação de destino do pacote. Conseqüentemente, a tarefa de recepção de pacote chamado "gerente da pilha" (*"stack manager"*) tem a maior prioridade no sistema. O início do processo de recepção é parecido com este do Linux. (ver seção 1.2). Após ter copiado um pacote no anel DMA de recepção, a placa de rede interrompe o processador. Em seguida, o tratador da interrupção armazena o pacote numa estrutura `rtskb` da reserva do dispositivo e coloca este `rtskb` numa fila de recepção, antes de acordar o "gerente da pilha". Este *"thread"*,

que comece a executar imediatamente, pois tem a maior prioridade do sistema, determina se o pacote é de tempo real ou não. Se for o caso, ele efetua as operações de recepção necessárias até entregar o pacote para a aplicação de destino. No caso contrário, o "gerente" coloca o pacote na fila dos pacotes de melhor esforços para ser recebidos, acorda o processo de baixa prioridade dedicado ao processamento desta fila e retorna.

1.3.3.3 A camada de rede Os principais problemas de latências para ser resolvidos na camada de rede são devidos ao uso do protocolo ARP (*"Address Resolution Protocol"*) e aos mecanismos de fragmentação de pacote do protocolo UDP/IP.

Em relação a fragmentação de pacote, o RTnet oferece uma opção de configuração que permite transmitir pacote de tamanho superior ao MTU de 1500 bytes do Ethernet padrão. As garantias de previsibilidade desta implementação são obtidas usando:

- Reservas específicas de estruturas `rtskb` para coletar os fragmentos de um mesmo pacote;
- Temporizadores associados a cada coletor para garantir o descarto de cadeias incompletas, antes que o conjunto de coletores se esgota.

Além disso, esta implementação requer que os fragmentos de um mesmo pacote sejam recebidos em ordem ascendente. Esta exigência limita o uso da fragmentação às redes locais, nas quais a reordenação de pacotes acontece pouca ou nunca.

Em relação ao protocolo ARP, os protocolos de redes convencionais utilizam-lo dinamicamente para construir a tabela ARP que associa os números IP e os endereços de roteamento Ethernet (KUROSE; ROSS, 2005). Para determinar o endereço MAC correspondendo a um endereço IP, uma estação envia um pacote ARP usando o endereço um-para-todos (`FF:FF:FF:FF:FF:FF`) do padrão Ethernet, perguntando quem detém a rota para este IP. Se a máquina de destino tiver no mesmo segmento Ethernet, ela manda uma resposta informando o seu MAC. Caso contrário, o roteador encarregado da sub-rede associada aquele IP informe o seu MAC. Quando a resposta chegue, a tabela ARP da estação de origem é atualizado. Para permitir a reconfiguração automática da rede, cada entrada desta tabela é apagada periodicamente.

Este procedimento introduz uma fonte de latência no estabelecimento de uma comunicação entre duas estações, pois o tempo de resposta não é determinístico, nem a freqüência na qual a tabela ARP deverá ser atualizada. No caso do RTnet, este procedimento foi trocado por uma configuração estática da tabela ARP na hora da configuração.

1.3.3.4 Os pacotes RTnet Para manter a compatibilidade com os dispositivos de hardware, RTnet utiliza o formato dos quadros Ethernet padrão, com o cabeçalho de 14 bytes, incluindo os

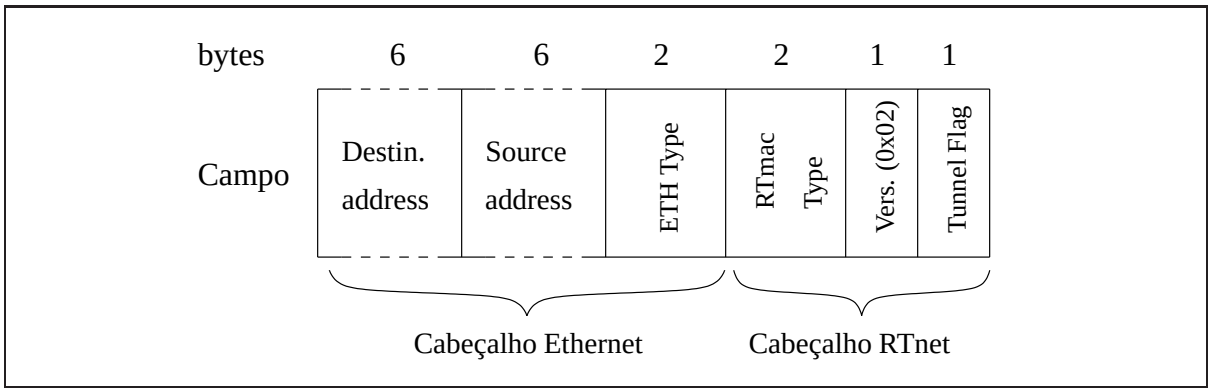

Figura 1.4: Cabeçalhos do RTnet.

endereços de destino e origem e o campo *"type"* de 2 bytes. Em função do protocolo utilizado, o tipo da mensagem é alterado. Por exemplo, mensagens IP utilizam o valor padrão `0x8000`, enquanto mensagens de gerenciamento, associadas a camada RTmac utilizam o valor diferente `0x9021`. Além de distinguir tipos de quadros pelo campo *"type"*, RTnet define um cabeçalho de 4 *"bytes"*, embutido no segmento de dados. Observa-se que a presença deste cabeçalho é necessário para que a comunicação RTnet se estabelece. Portanto, num dado segmento, todas as estações devem utilizar a pilha RTnet para que as propriedades temporais da rede sejam garantidas.

O cabeçalho específico do RTnet comporta os campos `RTmac`, `version` e `flag` como mostrado na figura 1.4. Estes campos são utilizados quando o valor do campo *"type"* do cabeçalho Ethernet (`0x9021`) indica que o pacote é destinado a camada RTmac. Isto acontece, por exemplo, quando se trata de um pacote de configuração. Outra possibilidade acontece quando o campo `flag` tiver o valor 1, indicando que se trata de um pacote de melhor esforço encapsulado num quadro RTmac. Neste caso, o valor do campo `RTmac` é utilizado para armazenar o tipo do pacote Ethernet, pois este valor é sobrescrito com o valor `0x9021` na hora do encapsulamento. Desta forma a camada RTmac consegue determinar qual é a aplicação para qual deve ser encaminhado um pacote de melhor esforço encapsulado.

A integração da comunicação de melhor esforço de processos do Linux com os serviços RTnet de tempo real para as tarefas Xenomai é realizada através de uma interface virtuais, chamada VNIC (*"Virtual NIC"*). Esta interface é configurada com o comando usual do Linux `ifconfig`. Pacotes de melhor esforço enviados pela `VNIC` são então encapsuladas no formato dos pacotes RTnet.

Os diferentes componentes descritos nesta seção, incluindo as estruturas `rtskb`, as funções de emissão e recepção, a implementação da fragmentação, das tabelas ARP estáticas e o formato dos pacotes formem a ossatura do RTnet. Isto é, uma pilha de rede determinística que pode ser utilizada diretamente pelas aplicações para organizar as suas comunicações de tempo

real. No entanto, esta estrutura pode ser completada por uma disciplina opcional de acesso ao meio que preenche a tarefa de organizar a comunicação no lugar das aplicações.

1.3.4 RTnet: As disciplinas TDMA e NoMAC

Apesar de ser opcional, o uso alguma disciplina de acesso ao meio pode ser necessário para prover determinismo, em particular, quando o meio tem uma política de acesso probabilista como é o caso de Ethernet (ver seção). Fiel ao seu modelo modular e hierárquico de desenvolvimento, RTnet fornece a interface RTmac para encapsular as disciplinas de acesso ao meio. Esta interface define os quatro serviços seguintes que uma disciplina deve imperativamente garantir:

- Os mecanismos de sincronização dos participantes da comunicação;
- A recepção e emissão de pacotes e o encaminhamento de cada pacote para o seu respectivo tratador;
- As funções e ferramentas necessárias para configurar a disciplina;
- O encapsulamento dos pacotes de melhor esforço através das interface VNIC.

Na versão atual do RTnet (0.9.10), a disciplina TDMA (*"Time Division Multiple Access"*) é a única disciplina de acesso ao meio disponível. A sua implementação utiliza uma arquitetura centralizada do tipo mestre / escravo. Em tempo de configuração, os diferentes clientes se registram no mestre, que pode ser replicado por motivos de tolerância a falha. Cada escravo reserva uma ou várias janelas de tempo, de acordo com as suas necessidades de banda. A verificação da capacidade da rede em atender as diferentes aplicações requisitando banda deve ser efetuada em tempo de projeto pelos desenvolvedores do sistema.

Num segmento RTnet regido pela disciplina TDMA, o relógio do mestre é utilizado como relógio global. Para organizar a comunicação, o mestre envia periodicamente uma mensagem de sincronização que define os ciclos fundamentais de transmissão. Quando um escravo quer começar a comunicar, a sua primeira tarefa consiste em se sincronizar com o mestre usando um protocolo de calibração. Após esta fase de configuração, um escravo pode utilizar as janelas que ele reservou em tempo de configuração para enviar as suas mensagens.

Para dar suporte a aplicações com requisitos temporais diferentes numa mesma estação, RTnet define 31 níveis de prioridades para as mensagens. Numa janela TDMA, os pacotes são enviados de acordo com esta prioridade. A mais baixa prioridade (32) é reservada para o encapsulamento dos pacotes de melhor esforço provendo das aplicações executadas no *kernel* Linux interface VNIC .

A disciplina NoMAC, como seu nome indica, não é um uma disciplina de fato. Quando carregada, esta disciplina simplesmente disponibiliza os serviços da pilha RTnet sem definir nenhuma política específica de acesso ao meio. No entanto, este exemplo de implementação constitui uma mão na roda que os desenvolvedores do RTnet disponibilizam para facilitar o trabalho de implementação de uma nova disciplina de acesso ao meio.

1.4 DORIS: UMA NOVA DISCIPLINA DO RTNET

A implementação do protocolo *DoRiS* na pilha de rede RTnet da plataforma Xenomai consistiu em criar uma nova disciplina de acesso ao meio encaixada na interface RTmac. Para isto, adotou-se a seguinte metodologia. Usou-se como base de desenvolvimento a disciplina NoMAC e aproveitou-se dos exemplos de implementação mais elaborados encontrados no código da disciplina TDMA. De forma geral, tentou-se minimizar as modificações da pilha RTnet e concentrar todas as novas funcionalidades necessárias ao protocolo *DoRiS* na disciplina *DoRiS* correspondente. Quando outras modificações se mostraram necessárias, utilizou-se as ferramentas C de compilação opcional para introduzir estas modificações somente para *DoRiS*.

Nesta fase de produção de um protótipo do protocolo *DoRiS*, utiliza-se um procedimento de configuração integrado à fase de comunicação, que garante a tolerância a falha, tanto de estações críticas quanto não críticas. Para este efeito, as seguintes restrições são adotadas.

Assume-se que a composição do grupo de participantes num segmento é conhecida e configurada em tempo de projeto. Isto significa que todas as estações que participem de um segmento *DoRiS* conhecem o número de participantes máximo, tanto do anel crítico que do anel não-crítico. Para fins de simplificação, suponha-se que os IDs dos participantes são regularmente alocados, indo de 1 a $nTask$ para as tarefas e de 1 a $nProc$ para os processos e que a comunicação de melhor esforço só se estabelece se tiver pelo menos uma tarefa presente no anel crítico. Caso contrário, todas as tarefas estão falidas e a comunicação de melhor esforço não pode acontecer. Suponha-se também que cada estação hospede uma tarefa, um processo ou os dois simultaneamente. Não se considera, por exemplo, situações nas quais duas tarefas são presentes numa mesma estação. Por fim, assume-se que $nProc \leqslant nTask$. Estas restrições poderão ser levadas pelo uso de um protocolo independente de configuração dinâmica em tempo de execução na fase M-Rd da figura . Esta figura, apresentada no capítulo é reproduzida aqui com o objetivo de facilitar a leitura desta seção.

É importante ressaltar, que para permitir a produção do protótipo de *DoRiS* no prazo deste Mestrado, escolheu-se uma implementação sem o mecanismo de reserva. No entanto, as estruturas necessárias foram previstas e a implementação deste mecanismo deverá ocorrer nos próximos meses.

Além da própria disciplina *DoRiS* cujo o detalhe da implementação será descrito na seção 1.4.5, os mecanismos de configuração e sincronização utilizados serão apresentado nas seções

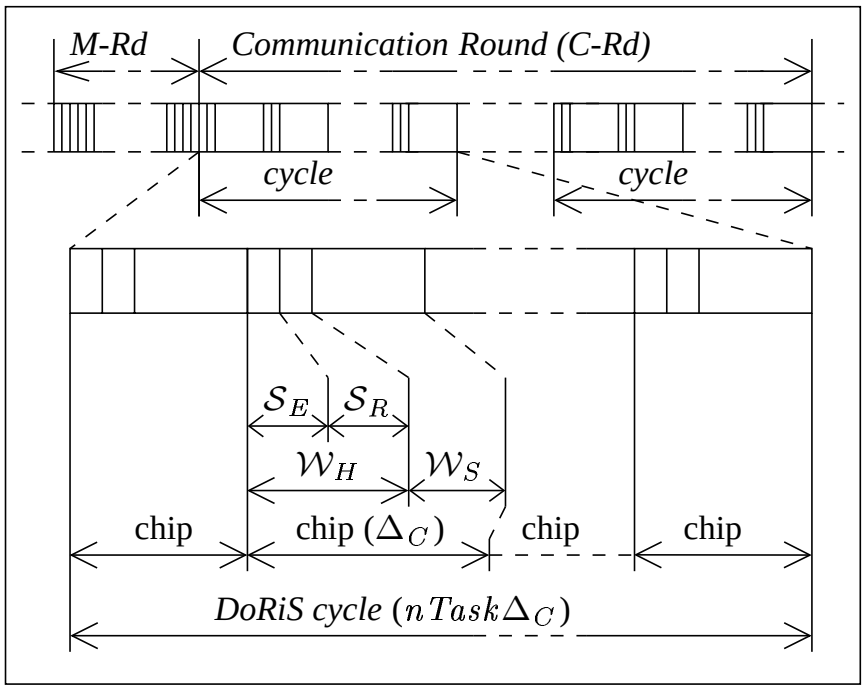

Figura 1.5: O esquema de divisão temporal de *DoRiS*

1.4.2, 1.4.3e 1.4.4. Prealavelmente, a implementação do modelo de comunicação um-para-todos será brevemente descrita na seção 1.4.1

1.4.1 Comunicação um-para-todos

Uma das diferencias do protocolo *DoRiS* com o protocolo TDMA utilizado pelo RTnet é o modelo de comunicação. Invés de usar o modo de comunicação ponto-a-ponto dos *"socket"*, *DoRiS* utiliza o modo de comunicação um-para-todos. Neste modo, os pacotes são enviados com o endereço Ethernet `FF:FF:FF:FF:FF:FF` e recebidos por todos os participantes da comunicação. Para identificar as mensagens, cada participante é identificado por um número único, seu ID. Considerou-se aqui que um *"byte"* era suficiente para codificar este ID, pois os cenários de comunicação projetados para o protocolo *DoRiS*(e também para a pilha RTnet), envolvem no máximo algumas dezenas de participantes. Utilizou-se portanto o último *"byte"* do endereço IP dos nós para este efeito. Este número sendo configurado em tempo de projeto, pode se garantir que todos os nós recebem um IP com o último *"byte"* diferente. Esta escolha permitiu aproveitar os códigos baseados em IP do RTnet, deixando a possibilidade de ter até 255 participantes num segmento *DoRiS*.

No caso dos pacotes de melhor esforço enviados com cabeçalhos RTmac, utilizou-se o cabeçalho IP do pacote encapsulado para obter os endereços IP de destino e de origem.

1.4.2 Configuração e sincronização do anel crítico

Na fase inicial, quando uma tarefa quer comunicar, ela começa por observar a comunicação já existentes no segmento durante dois ciclos. Lembrar que um ciclo dura exatamente $nTask * \Delta_C$. Depois destes dois ciclos de comunicação, se a tarefa não percebe nenhuma mensagem, ela deduz que ninguém está enviando mensagens ainda e escolhe qualquer instante para transmitir uma mensagem elementar. Para evitar qualquer risco de colisão com uma outra mensagem inicial, as estações são disparadas, na fase inicial de um segmento RTnet, com um intervalo de tempo superior a $2 * nTask * \Delta_C$.

Após o início de uma estação de $ID\ i$, uma outra estação j que queira participar da comunicação recebe as mensagens elementares de i. A estação j pode então deduzir o seu instante de transmissão t_j, utilizando o ID carregado pela mensagem elementar e o instante now da chegada desta mensagem:

$$t_j = now - \Delta_E + \{(nTask + j - i)\%nTask\} * \Delta_C \quad (1.1)$$

A única informação desconhecida nesta formula é a duração de um *"slot"* elementar Δ_E. Esta duração é a resultante de várias latências de natureza diferente:

i) A latência no nó emissor que pode ser decomposta em três causas principais: o tratamento da interrupção do temporizador de disparo da mensagem, o processamento da rotina de emissão de mensagens e a latência da placa de rede.

ii) A latência devida a transmissão e a propagação da mensagem no meio físico. Estes dois tempos dependem da taxa da banda e do cumprimento do cabo conectando os nós.

iii) A latência no nó recebedor pode ser decomposta em duas latências: a latência de recepção, que corresponde ao tempo necessário para copiar o pacote no anel DMA de recepção e a latência de interrupção, já amplamente discutida no capítulo .

Observa-se que estas diferentes fontes de latências tem uma variabilidade associada, pelo menos em que diz respeito aos ítens (i) e (iii). A estimativa de cada uma destas fontes de latência é possível em tempo de projeto. No entanto, no caso da implementação de *DoRiS*, uma solução global foi desenvolvida para estimar o valor total de Δ_E usando medidas realizadas durante a execução do protocolo.

1.4.3 Medidas de Δ_E

O procedimento adotado para a determinação do valor de Δ_E utiliza as propriedades das redes baseadas em comunicação um-para-todos (VERíSSIMO; RODRIGUES; CASIMIRO, 1997).

Basicamente, considera-se que quando uma estação emite uma mensagem, esta é recebida por todos os membros do segmento num mesmo instante. Ou seja, considera-se que as latências de propagações e de recepções de uma mensagem são as mesmas para todas as estações. Estas hipóteses podem ser resumidas pelas duas regras seguintes:

i) Desprezar as diferenças dos tempos de propagação entre qualquer duas estações.

ii) Considerar que as latências nas estações recebedores são idênticas.

Observa-se que a latência de emissão não interfere neste procedimento, pois só o instante de recepção é aproveitado para sincronizar a rede.

Em relação ao ponto (i), sabe-se que a velocidade de propagação das mensagens na rede é um pouco inferior a velocidade da luz no vazio. Pode-se assumir, por fins de estimativas, o valor de $2.510^8 ms^{-1}$. Deduz-se que para duas estações separadas de $25m$, o tempo de propagação é de $0.1\mu s$, enquanto que para duas estações separadas de $250m$, este tempo é de $1\mu s$. Percebe-se portanto que estes valores são de uma ordem de grandeza menor que os demais tempos de latência (ver seção).

O ponto (ii) estabelece que, em todas as estações, os tempos entre a chegada do pacote na placa de rede e a interrupção decorrente do processador sejam iguais. Esta hipótese pode ser garantido usando dispositivos de hardware com o mesmo comportamento temporal. No caso de *DoRiS*, esta hipótese não pode ser conservada, pois o uso de micro-controladores e outros dispositivos específicos implica em diferenças de comportamento temporal significativas. No entanto, assume-se que o comportamento temporal de cada dispositivo é determinístico e que fatores corretivos podem ser estimados em tempo de projeto para levar em conta a disparidade dos dispositivos de um segmento *DoRiS*.

Desta forma, as hipóteses (i) e (ii) podem ser consideradas válidas e as propriedades da comunicação um-para-todos podem ser usadas para calcular o valor da latência Δ_E . Para ilustrar este procedimento, considera-se três estações A, B e C e o cenário de trocas de mensagens representadas na figura 1.6.

No instante t_0, a estação A envia uma mensagem elementar (de tamanho $64B$) recebida por B e C num mesmo instante t_1. B espera então um tempo Δ conhecido por C para enviar a sua mensagem elementar. Quando C recebe esta nova mensagem, no instante t_3, C pode deduzir o valor de Δ_E , pois conhece os valores de t_1 e t_3 medidos localmente. Tem-se portanto:

$$\Delta_E = t_3 - t_1 - \Delta$$

Este procedimento foi generalizado para todas as estações de tal forma que a cada recepção de duas mensagens elementares consecutivas, cada estação possa calcular o valor de Δ_E . O

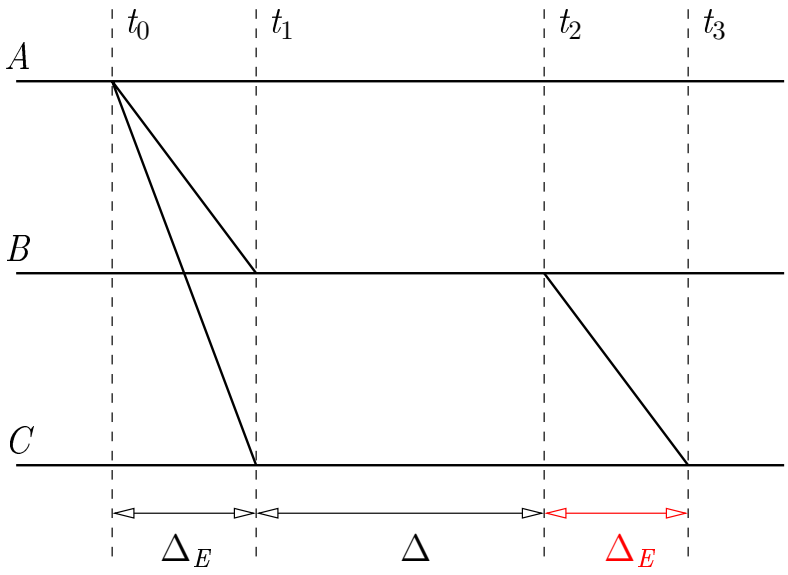

Figura 1.6: Cálculo de Δ_E

valor médio deduzido destas observações pôde ser então aproveitado para configurar *DoRiS*em tempo de projeto.

Em relação a implementação de *DoRiS*, a duração Δ_E do *"slot"* elementar é embutida na duração do Δ_C de um *"chip"*. Portanto, o valor do Δ utilizado nesta seção corresponde ao valor $\Delta_C - \Delta_E$.

1.4.4 Configuração do anel não-crítico

A configuração do anel não-crítico acontece da seguinte maneira. Uma estação querendo utilizar o anel não-crítico começa por observar a comunicação crítica. Usando o ID carregada pelas mensagens elementares e os instante de suas chegadas, a estação determina com precisão o instante de início do primeiro *"chip"* do próximo ciclo. Em seguida, a estação começa a contabilizar o número de janelas não-críticas ($\mathcal{W}_S$) vazias. No final do ciclo, se este número for igual a $nTask$, a estação deduz que o anel não-crítico ainda não tem nenhum participante. Neste caso, a estação adota o valor 1 para o *"token"* não-crítico. Senão, a estação recebe alguma mensagem não crítica durante o ciclo e neste caso, a estação utiliza o ID desta mensagem para iniciar o *"token"*.

Depois desta fase, o *"token"* é incrementada a cada janela não-crítica vazia. Em algum momento futuro, o valor do *"token"* alcance o valor do ID da estação em fase de configuração. Quando isto acontece, a estação pode começar a transmitir suas mensagens não-críticas.

É importante ressaltar que este mecanismo funciona apesar das falhas eventuais de estações

críticas e não críticas.

1.4.5 A implementação da disciplina *DoRiS*

Em cada estação, a disciplina *DoRiS* utiliza três tarefas de tempo real para organizar o acesso ao meio Ethernet. Uma cuida da recepção das mensagens e as duas outras organizam a emissão das mensagens nos dois anéis críticos e não-críticos.

1.4.5.1 Recepção A tarefa que cuida da recepção assíncrona de mensagens corresponde ao gerente da pilha (*"stack manager"*) do RTnet, descrito na seção 1.3.3.2. Ela tem a maior prioridade do sistema. A principal modificação desta tarefa é relacionada à utilização das informações temporais e lógicas associadas aos eventos de recepção de mensagens. Para tal efeito, um código de gerenciamento das variáveis de *DoRiS* é inserido no tratamento da interrupção de recepção do pacote. A execução destas operações no tratador de interrupção permite descartar os pacotes que não são destinadas a estação antes de acordar o "gerente da pilha".

As principais operações de gerenciamento realizadas por uma estação i são:

- Num evento de recepção de uma mensagem elementar:
  - Atualizar o valor do instante de emissão da próxima mensagem elementar de i;
  - Atualizar o valor do próximo instante de emissão de uma mensagem elementar;
  - Atualizar o contador $ChipCount$;
  - Atualizar o valor do *"token"* do anel não-crítico na ausência de mensagens na última $\mathcal{W}_S$;
  - Quando em posse do *"token"*, acordar a tarefa de emissão das mensagens não-críticas.
- Num evento de recepção de uma mensagem não-crítica:
  - Atualizar o valor do *"token"* do anel não-crítico na ausência de mensagens na última $\mathcal{W}_S$;
  - Atualizar o contador de mensagens não-críticas utilizado para detecção de $\mathcal{W}_S$ vazias;
  - Quando em posse do *"token"*, acordar a tarefa de emissão das mensagens não-críticas.

Em ambos os casos, a coerência dos dados locais com as informações carregadas pela mensagem é verificada e os dados necessárias para a análise temporal do protocolo são armazenados.

1.4.5.2 Emissão Duas tarefas de tempo real são encarregadas de gerenciar as operações de emissão de mensagens. A primeira, α, associada ao anel crítico, tem uma prioridade maior do que a segunda, β, responsável pela transmissão de mensagens em $\mathcal{W}_S$. Além do caractere seqüencial das operações de emissões nos dois anéis, a prioridade maior de α garante que a comunicação não crítica não interfere com a emissão de mensagens de tempo real crítico.

Para controlar as operações de emissão, as tarefas utilizam temporizadores e condições lógicas. No caso do anel crítico, α deve conhecer, com a melhor precisão possível, o instante t do início do próximo *"slot"* elementar no qual ela deve enviar uma mensagem . Considera-se por exemplo a estação de *ID* 1 e suponha-se que α_1 acabou de enviar uma mensagem elementar no instante now. A cada recepção pela estação 1 de uma mensagem elementar enviada pelas outras estações do anel crítico, o tempo t_1 é modificado, utilizando a equação 1.1. Portanto, o melhor valor para t_1 é o valor obtido depois da chegada da última mensagem elementar antes de t_1. Efetivamente, este valor minimiza o impacto do desvio de relógio da estação 1 e permite compensar eventuais desvios acumulados nas transmissões anteriores das mensagens elementares. O problema é que este valor de t_1 não é conhecido no instante now no qual o temporizador τ_1 responsável por acordar α_1 deve ser programada. A solução adotada aqui é de programar τ_1 para acordar α_1 no meio do *"chip"* antes do valor de t_1 conhecido no instante now. Quando a tarefa α_1 acorda, ela programa τ_1 utilizando o valor o mais recentemente atualizado de t_1.

Numa estação j, as emissões de mensagens não-críticas são regida pela circulação do *"token"* e por uma condição temporal que garante que estas mensagens sejam enviadas durante uma janela $\mathcal{W}_S$. Foi visto na seção 1.4.5.1 que o *"token"* é incrementado a cada recepção de mensagens não-críticas e que a tarefa β_j é acordada quando a estação j adquire o *"token"*, isto é, quando $token = j$. Nesta ocorrência, β_j estima o tempo ainda disponível na janela $\mathcal{W}_S$. Se este tempo for maior que o tamanho da mensagem em espera, β_j a envia. Caso contrário, β_j programa um temporizador para esperar até o início da próxima janela $\mathcal{W}_S$. Tanto a expiração deste temporizador, quanto um sinal disparado na chegada de uma mensagem crítica, podem então dar início a próxima janela $\mathcal{W}_S$.

Duas filas distintas são utilizadas para as mensagens críticas e não-críticas. Quando vazias, mensagens obrigatórias são criadas e enviadas conforme a especificação de *DoRiS*. Nesta implementação, prioridades não são definidas para as mensagens críticas.

Do ponto de vista das aplicações, o uso dos anéis de comunicação de *DoRiS* utiliza as funções usual do Linux para comunicação não-crítica e a interface do RTDM para as aplicações críticas.

1.4.6 Resultados preliminares

Elementos obtidos por enquanto:

- Medidas de Δ_E foram obtidas para hub de 10Mb e switch de 100Mb;
- Cenários de transmissão cruzadas foram realizados (com 3 estações), 1 e 2 comunicando via o anel crítico e 2 e 3 comunicando pelo anel não crítico;
- Constatou-se a sobrecarga devido a transmissão obrigatória de mensagens não-críticas.

1.5 CONCLUSÃO

Para ser escrita ainda....

A implementação do protocolo *DoRiS* obrigou os autores desta proposta a conhecer o protocolo de forma completa e detalhada. Este conhecimento profundo facilitou o trabalho de implementação, em particular quando tratou-se de modificar algum mecanismo. Por exemplo, a impossibilidade de obter placas de rede que possam informar o estado do meio com tempo de respostas da ordem do micro-segundos resultou numa modificação importante do mecanismo. de circulação do bastão circulante. Modificar a especificação formal foi simples devido ao aspecto modular de TLA+ e , devido ao se aproveitou da especificação formal da seguinte maneira: todos os

REFERÊNCIAS

IEEE. *IEEE Standart 1003.1 (POSIX), 2004 Edition*. 2004.

KISZKA, J. The Real-Time Driver Model and first applications. In: *Proc. of the 7th Real-Time Linux Workshop*. [S.l.: s.n.], 2005.

KISZKA, J. et al. RTnet-A flexible hard real-time networking framework. In: *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*. [S.l.]: IEEE Computer Society Press, 2005.

KUROSE, J. F.; ROSS, K. W. *Computer networking: a top-down approach featuring the Internet*. 3rd. ed. [S.l.]: Addison Wesley, 2005.

RTnet. 2008. http://www.rtnet.org - Last access jan. 08.

SALIM, J. H.; OLSSON, R.; KUZNETSOV, A. Beyond softnet. In: *Proceedings of USENIX 5th Annual Linux Showcase*. [S.l.: s.n.], 2001. p. 165–172.

VERíSSIMO, P.; RODRIGUES, L.; CASIMIRO, A. Cesiumspray: a precise and accurate global time service for large-scale systems. *Real-Time Systems Journal*, 1997.

Xenomai. 2008. http://www.xenomai.org - Last access jan. 08.