Universidade Federal da Bahia
Instituto de Matemática
Departamento de Ciência da Computação

Mestrado em Mecatrônica

ESPECIFICAÇÃO FORMAL, VERIFICAÇÃO E IMPLEMENTAÇÃO DE UM PROTOCOLO DE COMUNICAÇÃO DETERMINISTA, BASEADO EM ETHERNET

Paul Regnier

DISSERTAÇÃO DE MESTRADO

Salvador
Maio de 2008

Universidade Federal da Bahia
Instituto de Matemática
Departamento de Ciência da Computação

Paul Regnier

ESPECIFICAÇÃO FORMAL, VERIFICAÇÃO E IMPLEMENTAÇÃO DE UM PROTOCOLO DE COMUNICAÇÃO DETERMINISTA, BASEADO EM ETHERNET

Trabalho apresentado ao Programa de Mestrado em Mecatrônica do Departamento de Ciência da Computação da Universidade Federal da Bahia como requisito parcial para obtenção do grau de Mestre em Mecatrônica.

Orientador: *George Lima (PhD)*

Salvador
Maio de 2008

TERMO DE APROVAÇÃO

Título da dissertação ESPECIFICAÇÃO FORMAL, VERIFICAÇÃO E IMPLEMENTAÇÃO DE UM PROTOCOLO DE COMUNICAÇÃO DETERMINISTA, BASEADO EM ETHERNET

Autor PAUL DENIS ETIENNE REGNIER

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia – UFBA, pela seguinte banca examinadora:

PR. DOUTOR GEORGE MARCONI LIMA (ORIENTADOR)
Ph.D. em Ciências da Computação, University of York, Inglaterra
Professor do Departamento de Ciência da Computação da UFBA

PR. DOUTOR CARLOS MONTEZ (EXAMINADOR EXTERNO)
Doutor em Engenharia Elétrica, Universidade Federal de Santa Catarina (UFSC)
Professor do Departamento de Ciência da Computação da UFSC

PROF. DR. FLÁVIO MORAIS DE ASSIS SILVA (EXAMINADOR PPGM)
Dr.-Ing, Technische Universität Berlin, Alemanha
Professor do Departamento de Ciência da Computação da UFBA

A Vitória, Omin e

As vezes se assustam com o eco do silêncio do meu grito.
—MESTRE SOMBRA

Using clocks, one can convey information by not doing something.
—LESLIE LAMPORT

Agradeço

Ao meu filho *Omin* e a Vitória, companheira e mãe, pela compreensão e o apoio que permitiram a realização desta dissertação.

Aos membros do Laboratório de Sistemas Distribuídos por criar um ambiente de pesquisa estimulante e agradável, dando sempre vontade de levantar de manhã para ir ao trabalho, e de não deitar de noite para terminar este mesmo trabalho.

Ao meu orientador, George Lima, pela sua tenacidade em tentar me consertar, e pelos incentivos constantes.

A Antônio Marcos Lopez, fiel colaborador, inclusive nas piores horas da busca do *bug* "indebugável"...

A Francisco Carreiro e Paulo Pedreira, por seus conselhos avisados.

Aos membros do CPD que sempre responderam presente para me ajudar a resolver os problemas práticos de implementação.

A FAPESB, que financiou este trabalho de pesquisa através do edital 2630/2006.

DECLARAÇÃO

Declaro que o texto desta dissertação é de minha autoria e que todos os trechos e resultados nela utilizados advindos de outras fontes possuem referência de origem claramente expressa.

PUBLICAÇÕES

O presente trabalho deu origem a um artigo publicado no Workshop de Tempo Real 2006 (REGNIER; LIMA, 2006) que ocorreu em Curitíba no dia 2 de junho 2006.

Um segundo artigo (REGNIER, 2008), aceito para publicação no Workshop de Sistemas Operacionais 2008, apresenta a avaliação do determinismo temporal no tratamento de interrupções em plataformas de tempo real Linux.

RESUMO

Este trabalho apresenta um protocolo que torna o uso compartilhado de Ethernet eficiente para dar suporte aos sistemas de tempo real modernos. O protocolo foi especificado formalmente e sua correção foi atestada automaticamente através de um verificador de modelo. Em seguida, um protótipo foi realizado numa plataforma operacional de tempo real. Os resultados experimentais confirmaram a capacidade do protocolo em atender os objetivos definidos na sua proposta.

As aplicações que podem se beneficiar deste protocolo são principalmente aquelas compostas de dispositivos heterogêneos e distribuídos que têm restrições temporais de natureza críticas e não-críticas. Utilizando o protocolo proposto, tais sistemas podem utilizar o mesmo barramento Ethernet de forma eficiente e previsível. A utilização do barramento é otimizada através da alocação apropriada da banda disponível para os dois tipos de comunicação. Além disso, o protocolo, compatível com os dispositivos Ethernet comuns, define um controle descentralizado do acesso ao meio que garante flexibilidade e confiabilidade à comunicação.

Palavras-chave: Ethernet, Tempo Real, Tolerância a falhas, Especificação Formal.

ABSTRACT

This work presents a protocol that makes shared-Ethernet suitable for supporting modern real-time systems. The protocol has been formally specified and its temporal properties were verified using a model-checker. A prototype of the proposed protocol has been implemented in a real-time operational system. Experimental results show the capacity of the protocol to achieve the goals defined for its proposal.

Applications that can benefit from the proposed protocol are mainly those composed of heterogeneous distributed components, which are specified in terms of both hard and soft timing constraints.

Using the proposed protocol, such systems can efficiently and predictably use an Ethernet bus. The bus utilization is optimized by an appropriate allocation of the available bandwidth into hard and soft communication. Moreover, the protocol, compatible with standard Ethernet hardware, implements a decentralized medium access control that provides communication flexibility and reliability.

Keywords: Field-Ethernet, Real-Time, Fault-tolerance, Formal specification.

LISTA DE SIGLAS

ARP	*Address Resolution Protocol*
BEB	*Binary Exponential Backoff*
bT	*Bit Time*
COTS	*Commercial Off The Shelf*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidance*
DoRiS	Double Ring Service for Real -Time Systems
EOF	*End of Frame*
FCS	*Frame Check Sequence*
HAL	*Hardware Abstraction Layer*
IP	*Internet Protocol*
IFG	*Interframe Gap*
LLC	*Logical Link Control*
MAC	*Medium Acess Control*
RTAI	*Real Time Application Interface*
RTDM	*Real Time Driver Model*
RTnet	*Real Time net*
SOF	*Start of frame*
SOPF	Sistema Operacional de Propósito Geral
SOTR	Sistema Operacional de Tempo Real
STR	Sistema de Tempo Real
TDMA	*Time Division Multiple Access*
TLA	*Temporal Logic of Actions*
TLC	*Temporal Logic Checker*
TPR	*Timed Packet Release*
VTPE	*Virtual Token Passing Ethernet*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Os sistemas computacionais modernos devem atender a requisitos cada vez mais diversos e complexos, de acordo com o número crescente dos seus domínios de aplicação. Dentre os sistemas computacionais, os sistemas chamados de **tempo real** são aqueles especificamente concebidos para atender às necessidades das aplicações com requisitos temporais. Além de garantir a correção dos resultados gerados, um Sistema de Tempo Real (STR) deve também garantir os tempos de execução das suas tarefas e os prazos de entrega dos resultados. Portanto, num sistema de tempo real, as durações e os instantes das ações devem ser conhecidos de tal forma que o comportamento do sistema seja determinista e previsível. Exemplos de tais sistemas encontram-se nos sistemas mecatrônicos, em embarcados de carros, aeronaves e foguetes. As plantas industriais e a robótica constituem também domínios de aplicações destes sistemas.

Distinguem-se duas grandes famílias de STR de acordo com a criticidade dos requisitos temporais das aplicações: os STR críticos e os STR não-críticos. Pode-se entender esta classificação da seguinte maneira. Se a perda de um prazo de entrega de um resultado pode ter alguma conseqüência catastrófica, como a perda de uma vida humana, o STR é considerado **crítico**. Por exemplo, se um freio ABS de um carro ou o trem de pouso de um avião deixar de responder no prazo previsto, a conseqüência pode ser catastrófica em determinadas circunstâncias. Outros sistemas, pelo contrário, podem tolerar a perda de alguns prazos sem que haja nenhum considerado grave. Por exemplo, a perda de algumas partes de uma mensagem telefônica pode até deixar de ser percebida pelos interlocutores. E mesmo se estas perdas provocam uma alteração perceptível do sinal, elas não têm conseqüências críticas, e nem sempre impede a compreensão da conversa pelos interlocutores. Serviço de tempo real desta natureza, tal como tele-conferências ou aplicações multimídia, são exemplos de sistemas com requisitos temporais não-críticos. As falhas temporais do sistema degradam a qualidade do serviço, mas não comprometem o sistema como um todo.

No decorrer das últimas três décadas, a pesquisa relativa a sistemas de tempo real enfrentou

vários desafios. Ainda recentemente, considerava-se que estes sistemas eram compostos apenas de tarefas de tempo real críticas, simples e de duração curta, típicas das malhas de controle. Hoje, os sistemas de tempo real envolvem novas aplicações ligadas às áreas de telecomunicação, multimídia, indústria, transporte, medicina, etc. Portanto, além das habituais tarefas periódicas críticas que requerem previsibilidade dos tempos de respostas (LIU; LAYLAND, 1973), os sistemas atuais tendem a incluir tarefas não-críticas para dar suporte a eventos esporádicos ou aperiódicos (LIU, 2000). Tais sistemas, compostos de aplicações com requisitos temporais variados, críticos e não-críticos, são chamados **sistemas híbridos**.

Outra característica da evolução dos sistemas modernos é a modificação das suas estruturas, que passaram de arquiteturas centralizadas para arquitetura distribuídas. Para dar suporte a tais evoluções, a rede e os protocolos de comunicação devem ser adaptados para poder lidar com vários padrões de comunicação e oferecer qualidade de serviço adequada, tanto para as tarefas com requisitos temporais críticos quanto para as tarefas não-críticas ou aperiódicas que requerem taxas de transmissões elevadas. Em conseqüência, as redes industriais tradicionais tais como Profibus (PROFIBUS, 1996), ControlNet (ControlNet, 1997) ou CAN (ISO, 1993; LIAN; MOYNE; TILBURY, 1999), por exemplo, conhecidas por serem **deterministas e confiáveis**, podem não dispor de capacidade de banda suficiente para atender a estas novas exigências.

Uma alternativa às redes industriais tradicionais é a Ethernet, devido a sua alta velocidade de transmissão e à disponibilidade de componentes de prateleira de baixo custo. Além de ser popular nas arquiteturas de redes a cabo, a tecnologia Ethernet é bastante adequada para a implementação de protocolos de comunicação um-para-muitos (*multicast*), conforme ao modelo *publish-subscribe* (DOLEJS; SMOLIK; HANZALEK, 2004). Isso tem motivado o uso de Ethernet em ambientes industriais (FELSER, 2005; DECOTIGNIE, 2005).

No entanto, o algoritmo *Binary Exponential Backoff* (BEB), utilizado pelo protocolo CSMA-CD (*Carrier Sense Multiple Access with Collision Detection*) para controlar o acesso ao meio físico das redes Ethernet é probabilístico (IEEE, 2001; WANG; KESHAV, 1999), impedindo o uso direto desta tecnologia para aplicações com requisitos temporais críticos, típicos de contextos industriais.

Durante décadas, várias abordagens foram propostas para garantir o determinismo de acesso ao meio (DECOTIGNIE, 2005). Algumas envolvem a modificação do protocolo CSMA/CD no nível do hardware, outras introduzem uma camada de software situada entre a camada de con-

trole do acesso ao meio e a camada de aplicação. No capítulo 2, apresentaremos de forma mais detalhada as diferentes propostas, dando um foco maior às soluções baseadas em software. Apesar da diversidade das abordagens contempladas, mostraremos que algumas propriedades específicas aos sistemas híbridos, isto é, com requisitos críticos e não-críticos, não conseguem ser atendidas pelos protocolos existentes de maneira eficiente.

Para oferecer garantias de qualidade de serviço para estes sistemas híbridos e preservar a modalidade de comunicação um-para-todos (*broadcast*), oferecida pela tecnologia Ethernet, precisa-se então de um protocolo de comunicação determinista e confiável que possa otimizar o compartilhamento da banda entre as tarefas críticas e não-críticas.

No capítulo 3, apresentaremos uma proposta de protocolo de comunicação baseado em software que atende a estes requisitos de sistemas híbridos. Este protocolo possui importantes características que podem ser assim resumidas:

- A camada de enlace do protocolo CSMA/CD não é alterada e pode ser desenvolvida com interfaces Ethernet comuns, isto é, que respeitam a norma IEEE 802.3 (IEEE, 2001);
- As modalidades de comunicação um-para-todos e um-para-muitos, desejáveis para sistemas de tempo real, são contempladas sem perdas significativas de desempenho;
- O algoritmo do protocolo é descentralizado e provê mecanismos de tolerância a falhas de comunicação por omissão e a falhas silenciosas de máquinas (SCHLICHTING; SCHNEIDER, 1983);
- Os tempos de transmissão das mensagens são garantidos para as tarefas com requisitos temporais críticos;
- Através de um mecanismo inovador de reserva dinâmica, o protocolo provê uma alocação adaptável da banda para a comunicação com requisitos temporais críticos.
- O compartilhamento da banda entre as comunicações críticas e não-críticas é eficiente, isto é, provendo altas taxa de transmissão para aplicações não-críticos e garantias temporais para as aplicações críticas.

No processo de desenvolvimento, utilizou-se métodos formais para verificar as funcionalidades do protocolo, à medida que suas definições foram sendo desenvolvidas. Portanto,

escolheu-se apresentar o protocolo e a sua especificação num mesmo capítulo, pois a especificação formal constitui o melhor instrumento de descrição do protocolo e das suas propriedades temporais. A linguagem de especificação formal escolhida para especificar e verificar o protocolo é a linguagem de TLA+ (*Temporal Logic of Actions*) (LAMPORT, 2002), junto com o seu verificador de modelo associado TLC (*Temporal Logic Checker*) (YU; MANOLIOS; LAMPORT, 1999). Efetivamente, a estrutura modular desta linguagem permite um processo de escrita por incrementos sucessivos, de acordo com a elaboração progressiva das funcionalidades do protocolo. Ademais, o verificador de modelo TLC permite verificar automaticamente cada um destes incrementos.

Uma vez finalizada a fase de definição do protocolo, primeiro grande desafio deste trabalho, surgiu a segunda fase, não menos desafiadora, a de implementação. Para enfrentar este segundo desafio, buscou-se uma plataforma de tempo real de acesso livre que fornecesse suporte a aplicações multimídia e de controle via rede Ethernet. Para esta fase, o sistema Linux (BOVET, 2005) foi escolhido. As justificativas para tal escolha serão apresentadas no capítulo 4. No entanto, esta plataforma não oferece as garantias temporais necessárias no contexto de sistemas de tempo real. Algumas soluções serão então descritas para tornar Linux determinista. Entre estas soluções, a plataforma Xenomai (GERUM, 2005), dotada do núcleo Adeos (YAGHMOUR, 2001), surgiu como uma opção interessante.

Alguns experimentos mostraram a capacidade da plataforma Xenomai-Linux em garantir piores casos de latência no tratamento de interrupções abaixo de 20 microsegundos em máquinas atuais. Além disso, Xenomai provê uma camada de comunicação em tempo real, RTnet (KISZKA et al., 2005), baseada em Ethernet. Esta camada provê um protocolo de comunicação baseado em TDMA numa topologia de rede centralizada. Decidiu-se aproveitar desta infraestrutura de software para dotar a camada RTnet do novo protocolo aqui proposto.

Finalmente, a descrição desta implementação e de alguns resultados experimentais serão apresentados no capítulo 5, e conclusões gerais serão dadas no capítulo 6.

CAPÍTULO 2

ETHERNET E TEMPO REAL

A seção 2.1 descreve as características físicas do padrão Ethernet assim como o protocolo de acesso ao meio CSMA/CD (IEEE, 2001). Algumas das soluções propostas para tornar Ethernet determinista serão apresentadas na seção 2.2. Duas destas propostas serão detalhadas na seção 2.3, pois serviram de fontes de inspiração a este trabalho. Finalmente, considerações sobre sistemas híbridos serão discutida na seção 2.4.

2.1 ETHERNET

2.1.1 Características físicas

O padrão de comunicação Ethernet compartilhada (modalidade *half-duplex*) (IEEE, 2001) define um barramento Ethernet como um conjunto de estações utilizando um mesmo barramento (meio físico) e trocando mensagens entre elas. Um barramento Ethernet caracteriza um **domínio de colisão**, isto porque as mensagens emitidas por duas estações do barramento podem colidir de acordo com o protocolo CSMA/CD, como será visto na seção 2.1.2.

As mensagens, encapsuladas de acordo com o padrão (ver figura 2.1), são chamadas de quadro Ethernet, ou simplesmente quadro. Durante a transmissão de um quadro, diz-se que o barramento está **ocupado**, enquanto que, na ausência de transmissão, o barramento é dito **livre**. No fim da transmissão de um quadro, a transição de estado do barramento provoca uma interrupção em todas as estações, chamada de interrupção de fim de quadro (*End-Of-Frame interrupt*) e denotada EOF.

Dois tempos caracterizam a transmissão de um quadro no barramento: o tempo de transmissão e o tempo de propagação. O tempo de propagação T_{prop} de um pacote de uma extremidade a outra do barramento Ethernet só depende das características do meio físico, isto é, da velocidade de propagação de um sinal elétrico na rede e do comprimento total do barramento.

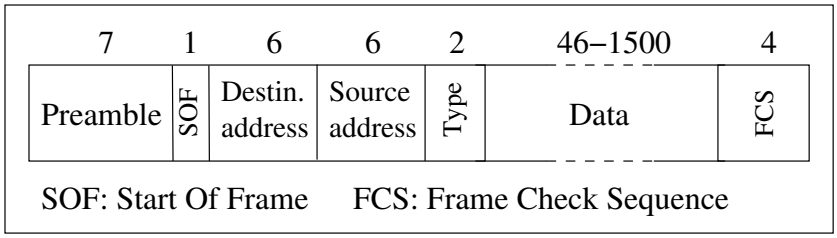


Figura 2.1: Um quadro Ethernet (números em *bytes*)

Nas redes 10Mbps e 100Mbps, o valor máximo deste comprimento é $500m$. Considerando a velocidade de propagação constante de $2,5\,10^8\ m/s$, tem-se $T_{prop} \approx 2\,\mu s$.

Para caracterizar o tempo de transmissão de maneira independente da largura de banda, utiliza-se a unidade temporal elementar *bit-time* (bT) definido da seguinte maneira: $1\,bT$ é o tempo que um bit leva para ser transmitido no meio físico. Por exemplo, numa rede *Fast*-Ethernet (100Mbps), $1\ bT = 10\ ns$. Já numa rede Gigabits (1Gbps), este valor é $1\ ns$.

Entre duas transmissões consecutivas, é preciso reservar um tempo de recuperação para que as diferentes camadas da pilha de rede possam esvaziar as suas memórias locais de recepção e redefinir as variáveis do protocolo CSMA/CD. Portanto, a transmissão de dois quadros consecutivos deve sempre ser separada por um tempo mínimo, chamado ***Interframe Gap*** *(IFG)*, durante o qual o meio permanece livre. Nas redes 10 e 100Mbps, $IFG = 96\,bT$.

A figura 2.1 apresenta o detalhe do formato de um quadro Ethernet com a indicação do tamanho dos campos em bytes. Os *bytes* do preâmbulo e *Start of frame* (SOF) marcam o início de um quadro. Eles são associados à camada física e servem para sincronizar o relógio da interface com a freqüência do sinal de entrada. Em seguida, o quadro contém o cabeçalho Ethernet, com os campos de endereços da estação de destino e da estação de origem, e o campo *type field* utilizado para definir o tipo ou o tamanho do quadro. Os demais cabeçalhos associados a protocolos de rede tais como IP ou TCP (não mostrados na figura) são encapsulados juntos com os dados de aplicações. Depois deste segmento de dados de tamanho variando entre 46 e 1500 *bytes*, o campo *Frame Check Sequence* (FCS) termina o quadro. Este campo serve para conferir a integridade do quadro depois da sua recepção.

Deduz-se que no padrão Ethernet, o tamanho dos quadros varia de 64 a 1518 bytes. Um quadro de tamanho mínimo de 64 bytes (512 bits) é chamado de *slot time*. Usualmente, não se considera no tamanho de um quadro os campos do preâmbulo e do SOF, pois são associados

à camada física. No entanto, o cálculo do tempo de transmissão de um quadro deve levar em conta estes 8 bytes iniciais.

Denota-se $\boldsymbol{\delta}$ o intervalo de tempo necessário para transmitir um quadro mínimo de tamanho 576 bits (7 bytes). Na taxa de 100Mbps, $\delta = 5,76\,\mu s$. Denota-se δ_m o tempo máximo de transmissão de um quadro de 1526 bytes. Na taxa de 100Mbps, $\delta_m = 12.208\,bT = 122,08\,\mu s$.

2.1.2 Controle de acesso ao meio

Como foi visto na seção anterior, o barramento Ethernet constitui um recurso compartilhado pelas estações de um mesmo barramento. Para permitir o desempenho da comunicação e o sucesso das transmissões, é preciso definir uma política justa, eficiente e confiável que organiza o acesso ao meio para que as estações possam se comunicar. Podemos expressar estas propriedades pelos seguintes requisitos:

i) Uma estação que quer transmitir conseguirá o meio (*liveness*);

ii) Um quadro transmitido chega sem alteração ao seu destinatário (*safety*);

iii) Todas as estações têm o mesmo direito de acesso ao meio (*fairness*).

Para atender a estes requisitos, o mecanismo de Controle do Acesso ao Meio (MAC) utiliza a capacidade que as estações têm de monitorar o meio físico para detectar se seu estado está livre ou ocupado (IEEE, 2001; WANG; KESHAV, 1999; WANG et al., 2002). De forma resumida, o protocolo funciona da seguinte maneira. Todas as estações monitoram o meio de maneira contínua. Quando uma estação quer transmitir, ela espera detectar o meio livre durante um tempo padrão igual ao *Interframe Gap* (IFG). Depois deste tempo, ela começa a emitir imediatamente. Enquanto ela está transmitindo, a estação continua monitorando o meio durante um *slot time*. Dois cenários podem então acontecer. No primeiro, a estação consegue transmitir durante um *slot time* sem perceber nenhuma diferença entre o sinal que ela transmite e o sinal que ela monitora. Este é o cenário de transmissão com sucesso. No segundo, a estação detecta uma alteração entre o sinal que ela está transmitindo e o sinal que ela está recebendo. Se isto ocorre, uma colisão está acontecendo, ou seja, uma outra estação começou a emitir um quadro simultaneamente. Neste caso, a estação pára de transmitir o seu quadro e transmite uma seqüência padrão de 48 bits, chamada de *jam*, para garantir que todas as estações do barramento

detectam a colisão. Depois, ela, e as demais estações que participaram da colisão, entram em estado de *backoff* antes de tentar transmitir novamente.

O tempo de *backoff* é um múltiplo do *slot time*. Um fator multiplicativo inteiro K é escolhido aleatoriamente dentro de um intervalo exponencialmente crescente de acordo com o número de colisões. Depois de n colisões, K é escolhido no conjunto $\{0, 1, 2, \ldots, 2^{m-1}\}$ onde $m = min(n, 10)$. Depois da enésima colisão, a estação espera então $K.512\ bT$ antes de poder tentar transmitir novamente. Desta forma, a probabilidade que aconteçam colisões em série decresce exponencialmente, tal como sugere o nome *Binary Exponencial Backoff* (BEB) deste algoritmo, que é parte do protocolo CSMA/CD.

Num barramento 100Mbps de $500m$ de comprimento, o tamanho mínimo de 72 bytes para um quadro Ethernet garante a detecção de uma colisão no pior caso. Para entender esta propriedade, imagine o seguinte cenário envolvendo duas estações A e B a uma distância de $500m$ uma da outra. O tempo de propagação de A a B é aproximadamente $T_{prop} = 2\mu s$ (ver seção 2.1.1). Suponha que o meio está inicialmente livre. Num instante t_1, a estação A começa a emitir um quadro q de tamanho mínimo. Num instante t_2, logo antes da chegada de q, a estação B começa a emitir. Em seguida, uma colisão ocorre num instante t_c tal que $t_2 \leqslant t_c \leqslant t_1 + T_{prop}$. Observe que no instante t_c, A ainda não terminou de enviar q, cuja transmissão leva $\delta = 5,76\mu s$. No instante $t_1 + T_{prop}$, B percebe a colisão, e conseqüentemente, pára de transmitir seu quadro e começa a emitir um quadro *jam*. Este quadro se propaga no meio e chega em A num instante t_3 tal que $t_3 \leqslant t_1 + 2\,T_{prop}$. A condição $2\,T_{prop} < \delta$ garante portanto que o quadro *jam* chegue em A antes que A tenha terminada a transmissão do quadro q envolvido na colisão. Desta forma, A percebe a colisão envolvendo q. Além disto, o quadro *jam* garante que a percepção da colisão acontece tanto em A quanto em todas as outras estações conectadas ao barramento.

Num outro cenário, considerando, por exemplo, duas estações distante de $1.000m$, o tempo de propagação entre A e B seria então de $4\mu s$. Neste caso, a informação da colisão poderá chegar em A quase $8\mu s$ depois do início da transmissão de q. Depois de tanto tempo, não somente a transmissão de q já poderá ter sido terminada, como também, A poderá já ter iniciado uma nova transmissão. O quadro *jam* será então associado erradamente ao segundo quadro. Percebe-se, portanto, que a condição $2\,T_{prop} < \delta$ é necessária para que o mecanismo de detecção das colisões funcione corretamente.

2.1.3 Probabilidade de colisão

Considerando duas estações A e B isoladas, a probabilidade máxima de colisão entre estas duas estações acontece quando A e B estão distantes o máximo possível uma da outra, isto é, de 500m. Imagine que A começa a transmitir, a probabilidade que uma colisão ocorra é a probabilidade que B começa a transmitir antes de ter percebido que A já está transmitindo, ou seja, antes de T_{prop}. Esta probabilidade depende da freqüência de transmissão de quadros por B. Mas, de qualquer forma, ela é relativamente pequena, pois T_{prop} é pequeno em comparação ao tempo médio de transmissão de quadros.

No entanto, um efeito de sincronização das estações pode acontecer e aumentar a probabilidade de colisão significativamente. Efetivamente, quando duas estações esperam o meio ocupado por uma terceira, elas sofrem um efeito de sincronização provocado pela espera conjunta do meio. A figura 2.2 ilustra um cenário de colisão, provocada pela espera conjunta com três estações A, B e C competindo pelo meio. Nesta figura, os tamanhos dos intervalos de tempo são apenas ilustrativos.

No cenário de "espera conjunta", suponha que a estação A esteja entre as duas estações B e C e aproximadamente a igual distância de ambas. Quando as estações B e C tentam transmitir, elas constatam que o meio está ocupado pela estação A que já está transmitindo. Portanto, elas esperam até sentir o meio livre. Quando a interrupção de EOF provocada pelo fim da transmissão de A acontece, B e C esperam IFG antes de começar a transmitir. Como as distâncias entre A e B e entre A e C são quase iguais, a interrupção EOF do fim da transmissão de A chega em B e C aproximadamente no mesmo instante. Portanto, depois de IFG, ambas começam a transmitir simultaneamente, resultando numa colisão, pois, após um tempo curto, as duas estações observam as diferenças entre os sinais que elas estão emitindo e recebendo. Conseqüentemente, após diagnosticar a colisão, ambas mandam imediatamente as mensagens *jam* antes de parar de transmitir e entrar em estado de *backoff* por um tempo aleatório, assim como foi visto no início desta seção. Se os tempos de *backoff* escolhidos foram diferentes, as duas estações conseguem transmitir com sucesso. Senão, uma nova colisão acontece e as estações entram novamente em estado de *backoff*.

Este cenário simples mostra o principal mecanismo responsável pelas colisões na Ethernet e, portanto, o caráter não determinista do algoritmo BEB do protocolo CSMA/CD. Como os

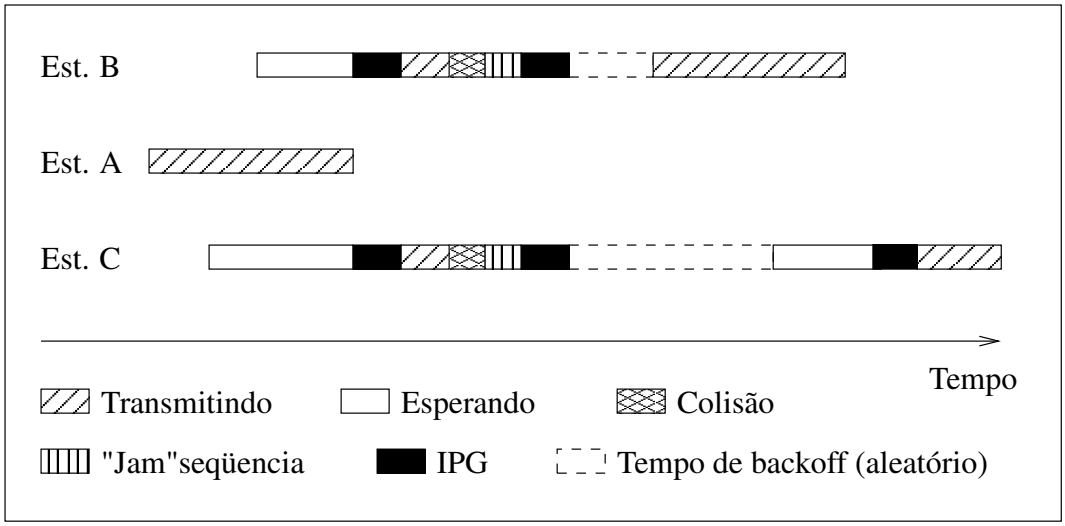


Figura 2.2: Um cenário de colisão

tempos de *backoff* são aleatórios, eles causam atrasos não deterministas nas entregas dos quadros. Utilizando um modelo probabilístico, a distribuição dos atrasos em função da carga da rede pode ser estimada teoricamente (SCHNEIDER, 2000; LAQUA; NIEDERMEYER; WILLMANN, 2002).

O cenário do pior caso é chamado de "efeito de captura". Continuando o cenário da figura 2.2, o efeito de captura do meio acontece, por exemplo, quando ocorrem colisões sucessivas entre C e uma ou mais estações. No exemplo da figura 2.2, quando B termina de transmitir, C começa a transmitir logo em seguida. Suponha que uma outra estação, eventualmente B, começa a transmitir também, provocando uma nova colisão do quadro de C. O contador de colisões de C é incrementado, e portanto, o intervalo de escolha do número aleatório de *backoff* é multiplicado por 2. Logo, a probabilidade que C ganhe o acesso ao meio diminui. A cada nova colisão do quadro de C com um quadro ainda não envolvido em colisão alguma, a probabilidade que C ganhe o acesso ao meio é dividida por 2. No pior caso, este efeito provoca o descarte daquele quadro, depois de 16 tentativas (WANG et al., 2002; DECOTIGNIE, 2005). Só então, com o custo do descarte de um quadro, a estação volta a competir para o meio com o seu contador de colisões igual a 0, e conseqüentemente, com a maior probabilidade de ganhar o meio em casos de colisão. A possibilidade de atrasos ou mesmo de perdas de quadros do algoritmo BEB do protocolo CSMA/CD torna este protocolo impróprio para ambientes *hard real time* (WANG; KESHAV, 1999).

2.2 ETHERNET: O DESAFIO DO DETERMINISMO

As propostas para aumentar a previsibilidade das redes Ethernet e oferecer garantias temporais às aplicações típicas de plantas industriais encontram-se em grande número na literatura (HANSSEN; JANSEN, 2003; DECOTIGNIE, 2005). Na modalidade Ethernet compartilhada (*half-duplex*), distinguem-se duas classes principais de soluções. Aquelas baseadas em modificações do hardware dos cartões Ethernet são apresentadas na seção 2.2.1 e as outras baseadas em software serão descritas na seção 2.2.2. Na seção 2.2.3, a modalidade Ethernet comutada (*full-duplex*) será discutida, pois é a modalidade dominante no mercado.

2.2.1 Soluções baseadas em modificações do hardware padrão

As soluções baseadas em hardware modificam a camada MAC do protocolo CSMA/CD para diminuir ou mesmo anular a probabilidade de perda de quadro. Esta seção apresenta brevemente as principais soluções que seguem esta abordagem.

O protocolo CSMA/CA (*Collision Avoidance*), usado no padrão 802.11 para redes Ethernet sem fio (CROW et al., 1997; IEEE, 1999), implementa um tempo de espera aleatório (*backoff*) não somente depois de uma colisão (como o CSMA/CD), mas também quando uma estação está esperando para o meio ficar livre, na situação de "espera conjunta" descrita na seção 2.1.3. Suponha que uma estação queira transmitir uma mensagem, mas que o meio esteja ocupado. Ela espera até sentir o meio livre, mas ao contrário do CSMA/CD, quando ocorre o EOF, a estação não transmite imediatamente. Ela entra em estado de *backoff* para um tempo de espera aleatório antes de tentar emitir. Desta forma, o efeito de sincronização pela "espera conjunta" é diminuído (KUROSE; ROSS, 2005). No caso de uma colisão, o tamanho do domínio de escolha do tempo de espera aleatório aumenta exponencialmente, assim como para o CSMA/CD.

O protocolo CSMA/DCR (*Deterministic Collision Resolution*) (LELANN; RIVIERRE, 1993) utiliza estruturas de dados adequadas para eliminar estações da competição para o meio depois de uma colisão. Resumidamente, o conjunto de estações de um segmento Ethernet é dividido em uma árvore binária de sub-conjuntos. A cada colisão sucessiva que ocorre, um dos dois sub-conjuntos ainda participando da competição pelo meio é retirado da competição. No caso do protocolo, CSMA/BLAM (*Binary Logarithmic Arbitration Method*) (MOLLE, 1994), uma fase

de arbitragem é utilizada depois de uma colisão para garantir que todas as estações competindo pelo meio utilizam o mesmo contador de colisão. Desta forma, todas as estações em competição têm chances iguais de ganhar o meio. Apesar de serem deterministas, estes dois protocolos apresentam uma variabilidade significativa nos tempos de respostas, entre o pior e o melhor caso. Esta variabilidade é indesejável em sistemas de tempo real, principalmente para aqueles que têm características periódicas (sistemas de controle, por exemplo).

Outras soluções utilizam um tamanho variável do quadro *jam*. Exemplos desta abordagem são os protocolos CSMA/PRI (GUO; KUO, 2005) *Priority Reservation by Interruptions* e o EQuB (SOBRINHO; KRISHNAKUMAR, 1998). Estes protocolos permitem definir prioridades entre as estações da seguinte forma. Quando uma estação quer ter acesso ao meio, apesar de este estar ocupado, ela começa a transmitir sem esperar que o meio fique livre. Desta forma, ela provoca uma colisão. Transmitindo uma mensagem *jam* de tamanho predefinido, ela informa às demais estações que elas devem abandonar a competição pelo meio. As prioridades das estações são associadas ao tamanho dos quadros *jam*, sendo o maior tamanho associado a estação com a prioridade mais alta.

Uma outra abordagem com prioridades utiliza dois valores de IFG diferentes para as comunicações com ou sem requisitos temporais (LO BELLO; MIRABELLA, 2001). A prioridade alta corresponde ao valor de IFG definido pela norma CSMA/CD (IEEE, 2001). A prioridade baixa utiliza este valor de IFG aumentado de 512 bT. Para enviar uma mensagem com prioridade baixa, um processo deve observar o meio livre durante este tempo maior. Portanto, qualquer mensagem com prioridade alta será transmitida antes. Esta solução permite o isolamento efetivo dos dois tipos de comunicações mas não elimina a possibilidade de colisão entre mensagens de alta prioridade.

É importante observar que as soluções baseadas em prioridades não garantem determinismo, a não ser quando a camada de controle lógico (LLC) utiliza alguma política de escalonamento dos envios.

Apesar das suas qualidades, as soluções com modificação do hardware não permitem o aproveitamento dos dispositivos existentes que são de baixo custo e de grande disponibilidade no mercado. Além disso, as alterações de hardware comprometem a compatibilidade do protocolo com o padrão Ethernet 802.3. Neste trabalho, uma abordagem mais flexível foi escolhida, onde o determinismo da rede é garantido exclusivamente no nível de software.

2.2.2 Soluções baseadas em software

As soluções baseadas em software consistem em estender a camada MAC do protocolo CSMA/CD, utilizando a sub-camada de controle lógico (LLC) localizada entre a camada de rede e a camada MAC. Esta camada intermediária filtra as mensagens de acordo com regras predefinidas de forma a evitar as colisões ou reduzir a probabilidade de elas acontecerem.

Um dos mecanismos mais simples para oferecer garantias temporais é a divisão do meio físico em ciclos temporais com atribuição de *slots* temporais de emissão para cada estação. O exemplo mais conhecido desta idéia é o *Time Division Multiple Access* (TDMA) (KUROSE; ROSS, 2005). Apesar de prover alto grau de determinismo, o TDMA apresenta um problema de desempenho em situação de baixa carga da rede, já que mesmo que uma estação não tenha nada para transmitir, nenhuma outra estação pode aproveitar o *slot* de transmissão disponível.

Uma outra abordagem é a utilização do modelo Mestre-Escravo. Uma estação (o mestre) gerencia a rede e distribui as autorizações de emissão por meio de mensagens para as demais estações (os escravos). Baseado neste modelo, o protocolo FTT-Ethernet (PEDREIRAS; ALMEIDA; GAI, 2002) permite a coexistência de comunicação crítica e não-crítica. Nesta arquitetura, o mestre constitui um ponto único de falha, que deve ser replicado a fim de prover tolerância a falhas. Devido a esta estrutura centralizada, a assimetria nas cargas da rede podem implicar problemas de extensão e de desempenho do protocolo.

Vários protocolos, tais como 802.5 e FDDI (HANSSEN; JANSEN, 2003), RETHER (VENKATRAMI; CHIUEH, 1994) e RTnet (HANSSEN et al., 2005) utilizam arquiteturas baseadas em anel lógico e bastão circulante (*token*) explícito para carregar as reservas e os direitos de acesso ao meio. Quando uma estação recebe o bastão circulante, ela adquire o direito de transmitir. Depois de completar a transmissão de uma ou mais mensagens, ela transmite o bastão circulante para o seu sucessor no anel lógico. O bastão circula regularmente, passando em todas as estações do anel uma após a outra. Algumas destas soluções utilizam o bastão circulante para transmitir informações de prioridades e de tempos alocados. No caso geral, estas soluções conseguem oferecer garantias temporais. No entanto, elas apresentam dois problemas. Primeiro, o tempo de transmissão do bastão circulante gera uma sobrecarga que pode ser significativa quando a maioria das mensagens são de tamanho mínimo. Segundo, no caso da perda do bastão circulante, o tempo de recuperação, necessário para detectar a falha e criar um novo bastão

circulante, é maior do que o tempo de transmissão do bastão circulante na ausência de falhas. Isto causa um indeterminismo potencial na entrega das mensagens, pois as falhas ocorrem de forma não previsível.

Para tentar resolver as limitações dos protocolos com bastão circulante explícito, novas abordagens foram desenvolvidas com mecanismos implícitos de passagem do bastão circulante. Muitas vezes, estas abordagens utilizam temporizadores (LAMPORT, 1984) para definir bifurcações no fluxo de execução de um processo, em função de condições temporais e lógicas. Observando a comunicação, cada estação determina o instante no qual ela tem direito de transmitir em função da sua posição no anel lógico e do valor do seu temporizador. Dois protocolos utilizando esta abordagem com bastão circulante implícito serão descritos detalhadamente na seção 2.3.

Outros protocolos utilizam mecanismos baseados em janelas temporais ou em tempo virtual (HANSSEN; JANSEN, 2003), mas estas abordagens são probabilísticas e não oferecem garantias temporais para sistemas críticos. Também probabilísticas, mas com mecanismos diferentes, as técnicas de *smoothing* (CARPENZANO et al., 2002) consistem em observar os padrões de comunicação na rede para impedir que haja transmissões em rajadas (*bursts*) (DECOTIGNIE, 2005).

2.2.3 Ethernet comutada

A tecnologia Ethernet comutada vem sendo muito utilizada nos últimos anos. Nesta arquitetura de rede, todas as estações são conectadas a um comutador que dispõe de memórias de recepção e emissão para armazenar e transmitir as mensagens. Quando uma estação A quer emitir uma mensagem m para uma estação B, ela a envia para o comutador o qual a encaminha para B. Se o comutador já estiver transmitindo uma mensagem para B quando m chega, ele coloca m na fila das mensagens esperando para serem transmitidas para B. Desta forma, a utilização de um comutador elimina o compartilhamento do meio físico entre as estações e a existência das colisões decorrentes. Além disso, nesta tecnologia, uma estação pode transmitir e receber ao mesmo tempo, multiplicando por dois a vazão total disponível para a comunicação.

Apesar destas vantagens, a tecnologia Ethernet comutada apresenta vários desafios relacionados a sua utilização em redes de controle industriais. Em particular, a utilização de filas no

comutador dificulta a implementação de comunicação *broadcast* rápida e determinista por duas razões. Em primeiro lugar, o processamento das mensagens (recepção, roteamento e transmissão) no comutador tem um custo temporal de vários μs (WANG et al., 2002), isso mesmo na ausência de filas de espera. Em segundo lugar, o comutador constitui um ponto de gargalo na comunicação, como ilustra o seguinte cenário. Suponha que várias estações mandem mensagens para o mesmo destinatário A de tal forma que a taxa acumulada de chegada de mensagem para A no comutador seja várias vezes maior que a taxa de transmissão do canal conectando o comutador a A. Portanto, a fila de espera, que tem um tamanho limitado, acaba se enchendo até provocar perdas de mensagens.

Tanto em Ethernet comutada quanto em Ethernet compartilhada, a abordagem do protocolo "Time Triggered Ethernet" resolve estas limitações (KOPETZ et al., 2005), oferecendo previsibilidade temporal e segurança no funcionamento. No entanto, estas garantias dependem de uma análise de escalonamento em tempo de projeto, e portanto, esta abordagem não permite reconfiguração da rede em tempo de execução.

2.3 OS PROTOCOLOS TEMPRA E VTPE

Como foi visto, nem todas as abordagens descritas conseguem atender aos requisitos específicos dos sistemas de tempo real críticos. Algumas apresentam pontos de falhas únicos, ou tempo de latência significativo na ocorrência de certos eventos (ex: perda do bastão circulante). As soluções baseadas em Ethernet comutada são bastante eficientes, mas dificultam a implementação de comunicação "um-para-muitos" deterministas. Além disso, perdas de mensagens podem ocorrer nos comutadores em caso de congestionamento nas memórias internas (DECOTIGNIE, 2005).

No contexto de um barramento Ethernet compartilhada, o protocolo VTPE *Virtual Token Passing Ethernet* desenvolvido por Carreiro et al (CARREIRO; FONSECA; PEDREIRAS, 2003), combina as abordagens de *token implícito* e TPR (*Timed Packet Release*) encontrada no protocolo TEMPRA (PRITTY et al., 1995a). Estes dois protocolos, VTPE e TEMPRA, constituíram as duas fontes principais de inspiração do nosso trabalho. Portanto, dedicaremos a próxima seção às suas descrições detalhadas.

2.3.1 Modelo *publish-subscribe*

Os dois protocolos apresentados nesta seção utilizam o modelo de comunicação *publish-subscribe* (DOLEJS; SMOLIK; HANZALEK, 2004). Quando uma estação precisa emitir uma mensagem, ela a publica no meio físico, utilizando no campo de destino o endereço para a comunicação um-para-todos, reservado pelo padrão Ethernet (48 bits com valor 1). Portanto, qualquer mensagem transmitida é recebida por todas as estações do barramento. Na sua recepção, uma estação determina quem publicou a mensagem através do campo "endereço de origem" do quadro Ethernet (ver figura 2.1). Ela então decide o que fazer com a mensagem: descartá-la ou encaminhá-la para a camada superior.

Como pode ser notado, este modelo de comunicação é coerente com o uso de Ethernet compartilhada, já que esta tecnologia permite disponibilizar modos de comunicação ponto-a-ponto, um-para-muitos e um-para-todos de forma eficiente. Por outro lado, o modelo *publish-subscribe* implica que todos os membros de um mesmo barramento recebem todas as mensagens publicadas. Isto tem conseqüências no que se refere ao uso da banda, como será descrito na seção 2.4.

No TEMPRA e no VTPE, assim como no protocolo de bastão circulante (IEEE, 1992), as estações são organizadas em um anel lógico. Cada estação tem um identificador inteiro único, variando de 1 a N, onde N é o número de estações do barramento. Define-se o tempo de rotação do bastão circulante (T_{RT}) como sendo o tempo entre duas passagens consecutivas do bastão circulante numa mesma estação.

No caso do protocolo TEMPRA, assume-se que o anel lógico coincide com o anel físico, ou seja, quanto maior é o identificador da estação, maior é a distância desta estação à primeira estação da rede. No caso do VTPE, a ordem dos identificadores é arbitrária. A figura 2.3 apresenta estes anéis para estes dois protocolos. As letras correspondem às estações e o eixo horizontal representa as distâncias entre elas. Os números correspondem aos identificadores e as linhas pontilhadas indicam o anel lógico associado.

O protocolo TEMPRA necessita da adição de um hardware específico, enquanto o protocolo VTPE não envolve modificações da camada MAC.

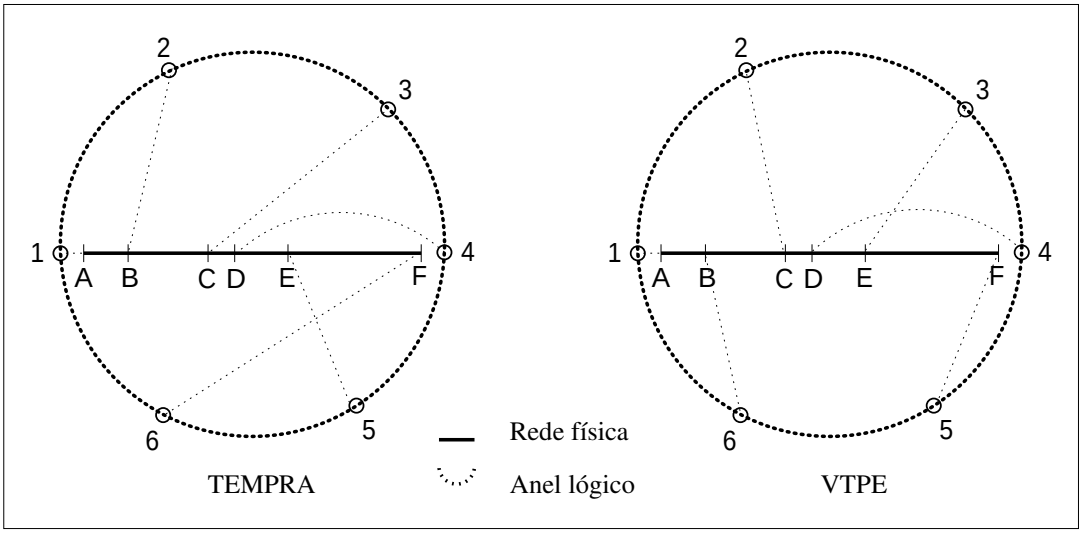


Figura 2.3: Os anéis VTPE e TEMPRA

2.3.2 O protocolo TEMPRA

O protocolo TEMPRA (PRITTY et al., 1995a) utiliza o mecanismo de *Timed Packet Release* (TPR) para impedir as colisões inerentes ao protocolo CSMA/CD. Quando uma estação precisa de um serviço com requisitos temporais críticos, ela emite uma requisição e as demais estações do barramento mudam do protocolo CSMA/CD para o protocolo TEMPRA, utilizando uma outra interface de rede dedicada à comunicação de tempo real. Depois que a comunicação de tempo real termina, as estações voltam para o CSMA/CD.

Para organizar o serviço de tempo real, assume-se que a primeira estação (que tem o menor identificador do anel) está posicionada numa extremidade do barramento. Esta estação, chamada de *monitor*, é selecionada para emitir um pulso (*slot pulse*), com um período predefinido. Este sinal, de duração t_s, tem um padrão único e se propaga unidirecionalmente no barramento, dando uma referência temporal sem ambigüidade para as estações, através da sua interrupção de fim de quadro (EOF). Na passagem de um pulso, cada estação inicia um temporizador. Estes seguem uma regra aritmética de parâmetro $t_d = 1\,\mu s$, ou seja, o temporizador da estação de identificador id vale $id * t_d$. Para adquirir o bastão circulante e ter o direito de transmitir, uma estação deve esperar o fim do seu temporizador e sentir o meio físico. Se este estiver livre, ela começa a emitir imediatamente. Logo depois desta transmissão, o monitor emite um pulso sem espera.

Entre dois pulsos, no máximo uma mensagem pode ser transmitida. Para impedir que uma estação de identificador pequeno possa monopolizar o uso do meio indefinidamente, impedindo as demais estações de transmitirem, uma estação só pode emitir novamente depois que ela observa uma janela livre, isto é, a passagem de dois pulsos consecutivos sem mensagens intercaladas.

Para garantir que todas as estações tenham a oportunidade de transmitir, o período T_{sp} do pulso na ausência de transmissão deve ser igual a $Nt_d + 2\,T_{prop} + T_s$, onde T_s é um tempo de segurança que leva em consideração o tempo de transmissão do pulso. ($T_s = 3\,\mu s$ de acordo com (PRITTY et al., 1995b)).

A principal vantagem do protocolo TEMPRA é sua simplicidade, o que o torna bastante confiável. Por exemplo, o mecanismo de temporizador permite tolerar a ausência (falha) de uma estação. No entanto, TEMPRA é um protocolo centralizado que requer a existência de um monitor para emitir o pulso. Portanto, falhas do monitor podem provocar falhas de comunicação no barramento.

Uma outra limitação do TEMPRA é a grande variabilidade dos tempos de rotação do bastão circulante entre o melhor e o pior caso. Denota-se, respectivamente T_{mc} e T_{pc} estes dois tempos. Se houver uma mistura de mensagens de tamanho máximo ($\delta_m = 122,08 \mu s$), típicas de aplicações não-críticas, e de mensagens de tamanho δ, típicas de aplicações com requisitos temporais críticos, a variabilidade de T_{mc} e T_{pc} torna-se significativa. No melhor caso, o tempo T_{mc} entre duas mensagens consecutivas da mesma estação de identificador id vale: $T_{mc} = \delta + id * t_d + T_s + T_{sp}$. O tempo T_{sp} corresponde ao tempo necessário para que a estação observe uma janela livre. No pior caso, todas as estações (inclusive o monitor) têm mensagens de tamanho δ_m para transmitir. Neste caso, o cálculo do tempo T_{pc} envolve a soma dos temporizadores e dos tempos de segurança:

$$T_{pc} = N\,(\delta_m + T_s) + \sum_{id=1}^{N-1} id * t_d = N\,(\delta_m + T_s + (N-1)t_d/2)$$

Neste cálculo, os tempos de propagação são desprezados.

Considerando um exemplo com 10 estações num barramento de $500m$ de comprimento numa rede 100Mbps, os valores para T_{prop}, T_{sp}, T_{mc} e T_{pc} ficariam tal que: $T_{prop} \approx 2\,\mu s$, $T_{sp} \approx 17\,\mu s$, $26,76 \leqslant T_{mc} \leqslant 35,76\,\mu s$ e $T_{pc} \approx 1.295,8\,\mu s$.

2.3.3 O protocolo VTPE

No protocolo VTPE (CARREIRO; FONSECA; PEDREIRAS, 2003), a circulação do bastão circulante implícito é organizado através do uso de contadores locais e de temporizadores. Cada mensagem que circula no barramento VTPE gera uma interrupção de *EOF*. Quando observado por um processo, esta interrupção provoca o incremento do valor de um contador local. Por outro lado, na ausência de transmissão, o contador é incrementado depois de um intervalo de tempo t_2 definido em tempo de projeto. O bastão circulante é adquirido por uma estação quando o valor do seu identificador id é igual ao valor do seu contador. Portanto, o bastão circulante é passado tanto explicitamente pela transmissão de mensagem quanto implicitamente pela ausência de transmissão. Depois de um período prolongado sem transmissões, um mecanismo baseado em temporizador garante que a estação em posse do bastão envia uma mensagem vazia. Desta forma, problemas de sincronização devido aos desvios dos relógios locais são prevenidos.

Por exemplo, suponha que a estação i adquire o bastão circulante. Os dois casos de passagens do bastão circulante seguintes são possíveis:

Passagem implícita A estação E_i não tem nada para transmitir. Neste caso, depois de um tempo predefinido t_2, as demais estações do barramento monitoram o meio e constatam que este está livre. Portanto, elas incrementam o seus contadores locais. Conseqüentemente, a estação E_{i+1}, sucessora de E_i, adquire o bastão circulante.

Passagem explícita A estação E_i transmite uma mensagem. A interrupção gerada por esta mensagem provoca o incremento do contador de todas as estações do barramento, e portanto, o bastão circulante passa para a estação sucessora da estação E_i.

Neste segundo caso, antes de poder emitir, a estação que adquire o bastão deve esperar um tempo t_1 correspondente ao tempo necessário para que a mensagem transmitida pela estação i seja processada por todas as estações do barramento.

Quando a rede for dedicada exclusivamente à comunicação de tempo real, as mensagens têm um tamanho máximo de δ. Adotando as mesmas definições de t_{pc} e t_{mc} usadas na seção anterior, calculamos que, no melhor caso:

$$t_{mc} = \delta + t_1 + \sum_{i=1}^{N-2} t_2 = \delta + T_1 + (N-1)\, t_2$$

enquanto que, no pior caso, este tempo é dado por:

$$t_{pc} = \sum_{i=1}^{N} (t_1 + \delta) = N\,(t_1 + \delta)$$

Apesar da variabilidade entre pior e melhor caso, esta abordagem apresenta as seguintes vantagens:

- A sobrecarga devido aos temporizadores é menor que a sobrecarga causada pela existência de uma mensagem explícita para gerenciar a passagem do bastão circulante;
- A flexibilidade do controle do bastão circulante permite tolerar falhas (não há diferença entre uma estação silenciosa e uma estação falha);
- A estrutura totalmente descentralizada do protocolo não apresenta ponto único de falha.

Considerando o mesmo exemplo da seção anterior e utilizando os valores de $t_2 = 25\,\mu s$ e $t_1 = 111\mu s$ de acordo com (CARREIRO; FONSECA; PEDREIRAS, 2003), deduzimos os seguintes valores: $T_{mc} \approx 341,76\,\mu s$ e $T_{pc} \approx 1.167,6\,\mu s$.

2.4 CONCLUSÃO

Neste capítulo, diversas soluções para aumentar a previsibilidade da comunicação baseada em Ethernet compartilhada foram apresentadas. Distinguiram-se notadamente duas soluções particulares,TEMPRA e VTPE, pois estas foram as principais fontes de inspiração deste trabalho. Estas duas soluções permitem resolver o não-determinismo do algoritmo BEB do CSMA/CD. No entanto, a utilização destes dois protocolos para sistemas híbridos é limitada pela variabilidade significativa dos tempos de rotação do bastão circulante entre melhor e pior caso. Entende-se aqui por sistemas híbridos, os sistemas compostos de aplicações com requisitos temporais variados, críticos e não-críticos.

Para completar a análise dos sistemas híbridos, temos que considerar os requisitos específicos das redes industriais. Neste contexto, alguns dos dispositivos que têm requisitos temporais

críticos são dispositivos eletrônicos que dispõem de capacidades de processamento menores que os computadores de uso geral. Tipicamente, a velocidade do processador destes dispositivos pode ser de duas a três ordens de grandeza menor que a velocidade dos processadores de computadores. Conseqüentemente, o tempo de processamento máximo de uma mensagem de tempo real de 64 bytes pode ser bem maior que o tempo necessário para a sua transmissão. Por exemplo, no protocolo VTPE (CARREIRO; FONSECA; PEDREIRAS, 2003), os autores utilizam um tempo de processamento máximo de $111\,\mu s$ para micro-controladores do tipo 8051 com processador de 12MHz.

Para impedir a emissão de uma mensagem de tempo real antes do fim do processamento da última mensagem que foi transmitida, o protocolo VTPE utiliza o temporizador t_1. Este parâmetro deve ser maior do que o tempo de processamento do processador mais lento do barramento. Depois de uma interrupção de EOF causada por uma mensagem crítica m, a nova estação em posse do bastão circulante espera o tempo t_1 antes de poder transmitir. Desta forma, ela tem certeza que todos os dispositivos conectados no barramento terminaram de processar a mensagem m.

No entanto, durante todo este tempo, o meio físico permanece livre. Isto porque o tempo de transmissão da mensagem crítica m (aqui $\delta = 5,12\,\mu s$) pode ser muito mais curto que o tempo do seu processamento. Neste caso, o canal de comunicação fica disponível para a comunicação não-crítica durante o tempo $t_1 - \delta$.

Esta constatação motivou o nosso esforço para desenvolver um protocolo sem modificação ou adição de hardware que possa integrar os serviços críticos e não-críticos num mesmo barramento Ethernet, otimizando o compartilhamento da banda entre estes dois serviços e oferecendo garantias temporais deterministas e constantes para o serviço de tempo real. Esta possibilidade baseia-se na existência de memórias internas na interface de rede, que permitem a recepção e a emissão assíncrona das mensagens, e, portanto, a liberação rápida do meio físico. Esta tecnologia encontra-se, por exemplo, nos micro-controladores de redes MC9S12NE64, DS80C410 e DS80C411 (FREESCALE, 2008; MAXIM/DALLAS Semicondutor, 2008).

É importante observar que soluções que permitem o controle de acesso ao meio compartilhado não são incompatíveis com o uso de comutadores. De fato, o mecanismo de controle de acesso ao meio permite evitar sobrecargas no comutador, e por conseguinte, prevenir perdas de mensagens. Apesar de a solução proposta neste trabalho se concentrar no acesso ao meio

Ethernet compartilhada, ela é compatível com o uso de comutador, sendo, portanto, adequada a infraestruturas de redes em uso atualmente.

CAPÍTULO 3

DORIS: ESPECIFICAÇÃO E VERIFICAÇÃO

3.1 INTRODUÇÃO

Neste capítulo, um novo protocolo de comunicação de tempo real, baseado em Ethernet compartilhada, é apresentado através das sua especificação formal na linguagem TLA+ (LAMPORT, 2002).

O protocolo *DoRiS*, cujo significado em inglês é *An Ethernet Double Ring Service for Real -Time Systems*, teve como fontes de inspiração principais os protocolos VTPE (CARREIRO; FONSECA; PEDREIRAS, 2003) e TEMPRA (PRITTY et al., 1995a), apresentados no capítulo 2. *DoRiS* foi concebido para dar suporte a sistemas híbridos nos quais dispositivos industriais como sensores, atuadores e controladores compartilham a mesma rede de comunicação com aplicações ou serviços não-críticos. Assim como foi mencionada anteriormente, a velocidade de processamento e as características da comunicação das aplicações de tempo real críticas e das aplicações com requisitos temporais não-críticos podem ser bastante diferentes. De fato, a maioria dos dispositivos de prateleira disponíveis tem capacidade de processamento pequena em relação à largura da banda Ethernet. Por exemplo, vimos na seção 2.4, que Carreiro et al (CARREIRO; FONSECA; PEDREIRAS, 2003) utilizaram micro-controladores que podem gastar até $111\,\mu s$ para processar uma mensagem de 64B. Como o tempo de transmissão de tal mensagem numa rede 100Mbps é $\delta = 5,76\,\mu s$, isso permite somente cerca de $5,2\%$ de utilização do barramento para a comunicação com requisitos temporais críticos. Por outro lado, aplicações não-críticas têm geralmente capacidade de processamento maiores e podem, portanto, utilizar taxas de transmissão mais elevadas. Considerando estas características dos sistemas híbridos, este trabalho propõe *DoRiS*, um protocolo novo que combina as abordagens de bastão circulante e *Time Division Multiple Access* (TDMA) para:

i) prover previsibilidade num segmento Ethernet compartilhada;

ii) oferecer garantias temporais às tarefas de tempo real;

iii) disponibilizar a largura de banda não aproveitada pela comunicação crítica para a comunicação não-crítica.

A seção 3.2 apresentará brevemente as motivações para a escolha da linguagem formal utilizada para especificar *DoRiS*. Em seguida, após introduzir alguns termos e elementos concernentes o modelo do sistema especificado, a seção 3.3 apresentará uma visão geral do protocolo *DoRiS*. As hipóteses de modelagem e uma breve introdução da linguagem formal escolhida darão o início da seção 3.4, que prosseguirá com a descrição detalhada da especificação formal de *DoRiS*. Finalmente, a seção 3.5 será dedicada à discussão da verificação automática da especificação e das suas propriedades temporais. A seção 3.6 concluíra este capítulo.

3.2 A ESCOLHA DA LINGUAGEM FORMAL

A concepção e implementação de um novo protocolo tal como *DoRiS* envolve decisões importantes. Neste processo, a especificação formal do protocolo e sua verificação automática permitem adquirir confiança no projeto elaborado. De fato, especificar formalmente um sistema e verificar a sua correção de maneira automática aumenta significativamente a compreensão do seu comportamento e pode ajudar a detectar erros ou defeitos de concepção numa fase inicial do desenvolvimento do *software* (AUSLANDER, 1996; CLARKE; GRUMBERG; PELED, 1999). Tal abordagem já foi utilizada em vários domínios de aplicação (BARBOZA et al., 2006; JOHNSON et al., 2004; HANSSEN; MADER; JANSEN, 2006).

Na últimas três décadas, várias linguagens formais e ferramentas de verificação foram desenvolvidas baseadas em lógica temporal (PNUELI, 1979; LARSEN; PETERSON; YI, 1997; HENZINGER; MANNA; PNUELI, 1992). Por exemplo, Esterel (BERRY; GONTHIER, 1992) é uma linguagem síncrona bastante adequada para especificar componentes de *hardware* e para expressar propriedades complexas de circuitos ou sistemas síncronos mais amplos. A linguagem de Especificação e Descrição (SDL) e o padrão Estelle foram recentemente estendidos para permitir a especificação de STR (FISCHER, 1996; SINNOTT, 2004). No entanto, SDL e Estelle são dedicados a descrição formal e a geração automática de código na fase inicial do projeto e ambas

não permitem a verificação automática de modelos. Várias outras ferramentas são baseadas em autômatos temporais tal como Kronos (DAWS et al., 1996) e HyTech (ALUR; HENZINGER; HO, 1996). Promela, por exemplo, é dotada de uma extensão de tempo real que fornece uma semântica para a especificação e a verificação de propriedades temporais (TRIPAKIS; COURCOUBETIS, 1996). Uma outra ferramenta bastante utilizada para especificar e verificar sistemas de tempo real é UPPALL (LARSEN; PETERSON; YI, 1997). Por exemplo, num trabalho recente (HANSSEN; MADER; JANSEN, 2006), o protocolo RTnet, de tempo real baseado em Ethernet, foi especificado e suas propriedades temporais foram verificadas com UPPAAL.

Uma outra alternativa possível para especificar Sistemas de Tempo Real (ABADI; LAMPORT, 1994; LAMPORT; MELLIAR-SMITH, 2005) é a linguagem de especificação formal TLA+ (*Temporal Logic of Actions*) (LAMPORT, 2002), junto com o seu verificador de modelo associado TLC (*Temporal Logic Checker*) (YU; MANOLIOS; LAMPORT, 1999). Baseada na teoria dos conjuntos de Zermelo-Fraenkel e na lógica temporal (PNUELI, 1979), a linguagem TLA+ foi especialmente concebida para expressar propriedades temporais de sistemas concorrentes e distribuídos (JOHNSON et al., 2004). A escolha de TLA+ para especificar e verificar *DoRiS* se deu pelas razões principais seguintes:

i) TLA+ tem uma estrutura modular que permite um processo de escrita por incrementos sucessivos, de acordo com o grau de abstração ou de detalhe desejado;

ii) Uma especificação em TLA+ é bastante parecida com código de programas, e, portanto, oferece uma base sólida para a implementação de *DoRiS*;

iii) O verificador de modelo TLC permite verificar automaticamente a especificação, assim como as propriedades temporais a ela associadas;

iv) Tanto os documentos de definição da linguagem TLA+ como a ferramenta TLC e vários exemplos são disponíveis livremente na Internet (TLA+, 2008).

3.3 *DORIS*: O PROTOCOLO

Localizado nas camadas *LLC* e MAC do modelo OSI, entre as camadas de rede e física, o protocolo *DoRiS* atua como um filtro lógico, evitando as colisões inerentes à camada de controle de acesso ao meio do protocolo CSMA/CD (IEEE, 2001). *DoRiS* é concebido para dar

suporte a sistemas híbridos, nos quais sensores industriais, atuadores e controladores compartilham a rede de comunicação com as demais aplicações com requisitos temporais não-críticos. Dispositivos industriais (micro-controladores, sensores, etc.) são chamados de "estações lentas", pois a velocidade de processamento de tais dispositivo é relativamente baixa em comparação com a dos micro-computadores, chamados de "estações rápidas". Nas seções seguintes, o protocolo é detalhado.

3.3.1 Sistema, modelo e terminologia

O conjunto de estações (lentas e rápidas) interligadas num barramento Ethernet compartilhada constitui um segmento *DoRiS*. Embora vários segmentos *DoRiS* possam ser inter-conectados por comutadores ou roteadores, a especificação e verificação de *DoRiS* se restringe a um segmento isolado. Em tal segmento, cada estação executa um servidor *DoRiS*, que é responsável pela execução das suas funcionalidades. Uma estação rápida pode hospedar tanto tarefas críticas como processos com requisitos temporais não-críticos. Para fins de clareza das notações, as primeiras serão simplesmente chamados de tarefas e os segundos, de processos. Os dois conjuntos, de tarefas e de processos, denotados respectivamente $TaskSet$ e $ProcSet$, definem os dois anéis lógicos de um segmento *DoRiS*, nos quais um único bastão circula.

Mensagens enviadas pelas estações lentas são curtas, usualmente periódicas, e têm requisitos de tempo real críticos. Assume-se que tais mensagem, chamadas de "mensagens críticas", têm um tamanho mínimo de 64 bytes e que, conseqüentemente, são transmitidas no barramento no tempo de transmissão δ. Denota-se π o tempo de processamento, no pior caso, da estação mais lenta do segmento *DoRiS*. Assume-se que $\delta \ll \pi$ e que a recepção e o processamento das mensagens são duas operações independentes que podem ser realizadas simultaneamente. A primeira suposição reflete a existência de estações lentas no segmento, enquanto a segunda corresponde à realidade dos dispositivos de hardware modernos dotadas de memórias locais e de capacidade de DMA (*Direct Memory Access*). A segunda suposição implica, notadamente, que duas ou mais mensagens críticas podem ser enviadas em seguida. Observa-se que se houver somente estações lentas presentes no segmento *DoRiS*, a taxa máxima de utilização do barramento é de $\frac{\delta}{\pi+\delta}$. No entanto, se houver também estações rápidas, a fração da banda não utilizada pode ser aproveitada pelos processos. É desta constatação que surgiu a proposta de

DoRiS.

Assim como os protocolos VTPE e TEMPRA (ver seção 2.3), *DoRiS* utiliza o modelo de comunicação *publish-subscribe* (DOLEJS; SMOLIK; HANZALEK, 2004), de acordo com o qual, quando uma aplicação quer enviar uma mensagem, ela utiliza o endereço Ethernet de comunicação um-para-todos padrão (`FF:FF:FF:FF:FF:FF`). Quando um servidor *DoRiS* recebe uma mensagem, ele determina, de acordo com a identidade do seu emissor, se há alguma aplicação interessada naquela mensagem. Caso positivo, ele entrega a mensagem. Senão, ele a descarta. A princípio, as tarefas não precisam completar o processamento de todas as mensagens críticas. No entanto, para simplificar o modelo, assume-se aqui que todas as mensagens críticas são inteiramente processadas por todas as tarefas. Distinguem-se portanto as operações de recepção e de processamento de uma mensagem.

Em relação ao modelo temporal, assume-se um sistema distribuído síncrono. Isto significa que as operações efetuadas pelas estações podem ser sincronizadas umas com as outras. Esta suposição se baseia no esquema de divisão temporal de *DoRiS*, que, como será visto, tem pontos de sincronização regulares e previsíveis, que ocorrem dentro de uma janela de tempo curta comparada com o desvio dos relógios locais. Isto implica que os relógios locais das estações são sincronizados.

Por fim, assume-se que as estações podem falhar por *crash-recovery*, ou seja, parar qualquer tipo de atividade (SCHLICHTING; SCHNEIDER, 1983), e voltar a funcionar normalmente depois de um tempo arbitrário. Mensagens enviadas podem eventualmente ser perdidas, porém, estações rápidas devem imperativamente perceber a interrupção associada à recepção de uma mensagem, mesmo que o conteúdo da mensagem seja perdido. Como será visto, este requisito é necessário para controlar a circulação do bastão entre os processos.

3.3.2 O esquema de controle de acesso ao meio

A comunicação num barramento *DoRiS* é temporalmente dividida em uma alternância de fases (*rounds*) de comunicação (*C-Rd*) e fases de configuração dos membros (*M-Rd*), assim como ilustrado na figura 3.1. Durante a fase de configuração, o algoritmo de controle da composição do segmento *DoRiS* é responsável por estabelecer uma visão única da composição do grupo de participantes do segmento *DoRiS*. Esta visão é compartilhada por todos os membros

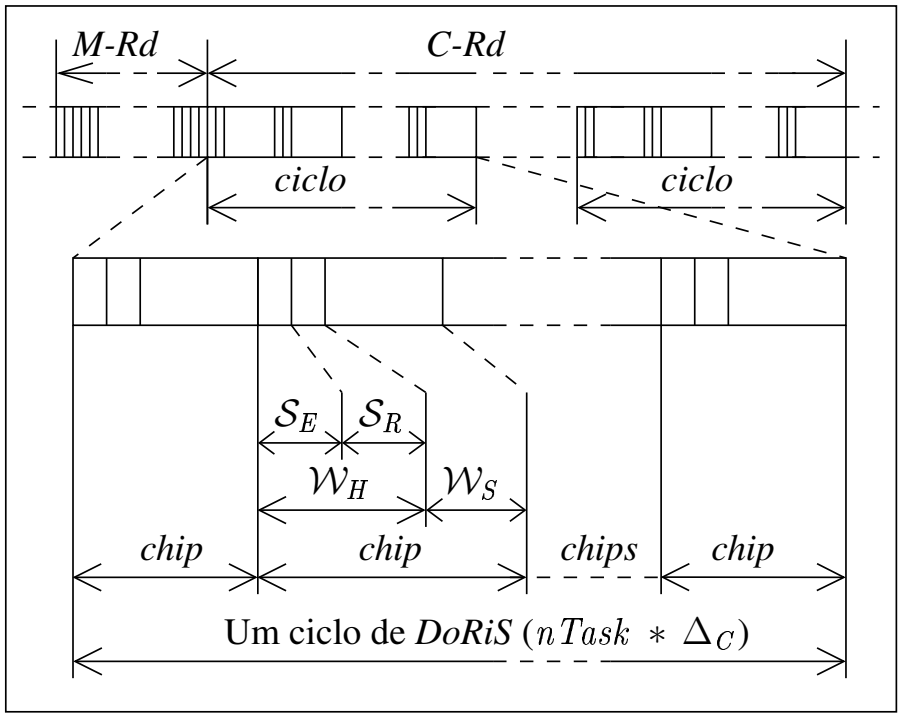


Figura 3.1: O esquema de Divisão Temporal de *DoRiS*

deste segmento. O problema do estabelecimento de tal visão, também conhecido como o problema de composição de grupos, são assuntos de vários trabalhos (CRISTIAN, 1988; CHANDRA; TOUEG, 1996; LAMPORT; MELLIAR-SMITH, 1998). No contexto desta dissertação, considera-se que os grupos de tarefas ($TaskSet$) e processos ($ProcSet$) são definidos em tempo de projeto e especifica-se somente a fase de comunicação do protocolo *DoRiS*. Denota-se $nTask$ e $nProc$ os cardinais destes dois conjuntos.

Usando TDMA, cada fase de comunicação (*C-Rd*) define um número arbitrário, porém fixo, de ciclos periódicos, os quais são subdivididos em exatamente $nTask$ *chips*, assim como pode ser observado na figura 3.1. Um mapeamento dos naturais sobre o conjunto dos *chips* associa um inteiro positivo módulo $nTask$ a um *chip*. Este mapeamento define um contador, que é denotado $chipCount$. Cada *chip* é, por sua vez, dividido em duas janelas, críticas e não-críticas, denotadas $\mathcal{W}_H$ e $\mathcal{W}_S$, e associadas, respectivamente, às comunicações de tempo real críticas e não-críticas. As tarefas só transmitem mensagens durante as janelas $\mathcal{W}_H$, enquanto os processos apenas utilizam $\mathcal{W}_S$ para transmitir as suas. Os tamanhos destas duas janelas são, respectivamente, denotados por Δ_H e Δ_S e o tamanho de um *chip* é definido por $\Delta_C = \Delta_H + \Delta_S$. Para permitir um certa flexibilidade e a definição de políticas de escalonamento

das mensagens, a janela $\mathcal{W}_H$ é também subdividida em dois *slots*: o *slot* elementar ($\mathcal{S}_E$) e o *slot* de reserva ($\mathcal{S}_R$). As mensagens transmitidas nos *slots* $\mathcal{S}_E$ e $\mathcal{S}_R$ são mensagens críticas, chamadas, respectivamente, de mensagem elementar e de reserva. Uma vez por ciclo, cada tarefa envia uma mensagem elementar dentro de $\mathcal{S}_E$ enquanto $\mathcal{S}_R$ é utilizada para oferecer um mecanismo de reserva.

Uma propriedade fundamental de *DoRiS* é que todas as tarefas devem imperativamente enviar uma única mensagem elementar por ciclo, no seu respectivo *chip*. Desta forma, estas mensagens podem ser utilizadas para fins de detecção de falhas e de sincronização dos relógios locais.

Para permitir o aumento da taxa disponível para uma ou outra tarefa, é possível atribuir vários *chips* de um mesmo ciclo a uma mesma estação. No entanto, com o objetivo de facilitar a apresentação e a especificação de *DoRiS*, assume-se nesta apresentação, sem perda de generalidade, que cada estação envia uma única mensagem elementar por ciclo. No entanto, esta suposição poderia ser relaxada, de acordo com o algoritmo utilizado na fase de configuração, alocando um número maior de *chips* por ciclo a cada estação.

O controle de acesso ao meio de *DoRiS* é organizado por um bastão circulante virtual, que circula nos anéis críticos e não-críticos, descritos na seção 3.3.1. O bastão circulante é dito virtual porque sua transmissão é associada a regras temporais e lógicas baseadas na observação da comunicação no barramento. O isolamento dos dois anéis de *DoRiS* é garantido pelo uso do mecanismo de TDMA. Em relação ao anel não-crítico, o bastão circula durante a janela $\mathcal{W}_S$ a cada vez que uma interrupção é gerada pela placa de rede. Quando um processo não tem nenhuma mensagem a transmitir, o servidor envia uma mensagem de tamanho mínimo para passar o bastão circulante ao próximo processo do anel não-crítico.

Observa-se que numa fase inicial de concepção do protocolo *DoRiS*, pensou-se em utilizar a capacidade de sensoriamento do meio para organizar a transmissão do bastão utilizando o mecanismo TPR do protocolo Tempra, apresentado na seção 2.3. No entanto, em face da dificuldade de conseguir placas de rede que permitam o acesso ao estado do meio (livre ou ocupado), esta possibilidade foi descartada.

3.3.3 O mecanismo de reserva

Para aumentar a sua taxa de transmissão de mensagens críticas, uma tarefa pode utilizar os *slots* de reservas de acordo com o seguinte mecanismo.

Cada mensagem elementar enviada por uma tarefa T_i carrega uma lista de inteiros (módulo $nTask$), que especifica os identificadores dos *slots* que T_i pretende usar para enviar mensagens adicionais. Esta lista é um subconjunto do conjunto dos $nTask$ *chips* seguindo o *chip* no qual T_i envia sua mensagem. A tarefa T_i só pode reservar um *slot* que ainda não foi reservado por uma outra tarefa. Para este efeito, cada servidor mantém uma tupla de reservas, denotado res, no qual ele armazena as reservas especificadas em cada mensagem elementar recebida. A consistência da tupla res pode eventualmente ser comprometida por falhas de omissão de mensagens elementares. Efetivamente, quando um servidor não recebe uma mensagem elementar, ele deixa de atualizar res e alguma tarefa pode tentar reservar algum *slot* previamente reservado. Para evitar tal cenário, que poderia levar a uma colisão, uma tarefa T_i só pode reservar um *slot*, se a tupla res do seu servidor estiver em estado consistente, isto é, se o servidor de T_i tiver recebido as $nTask$ mensagens elementares precedendo o *chip* de emissão de T_i. Desta forma, a alocação dinâmica de *slots* de reserva é tolerante a falhas de omissão.

Este mecanismo de reserva é uma inovação do protocolo *DoRiS* que permite a implementação de alguma política de escalonamento. Efetivamente, uma tarefa dispõe de um *slot* elementar por ciclo e pode usar até $nTask$ outros *slots*. No entanto, a discussão de tal política foge do escopo desta dissertação.

3.4 A ESPECIFICAÇÃO FORMAL DE DORIS

A descrição do protocolo *DoRiS* e de sua especificação é realizada de forma progressiva, fornecendo inicialmente uma visão de alto nível, e entrando gradativamente na apresentação dos detalhes do protocolo. A seção 3.4.1 descreve o conjunto das hipóteses de modelagem adotadas para especificar o protocolo. Em seguida, a seção 3.4.2 introduz alguns conceitos básicos de TLA+. As demais seções apresentam as principais fórmulas que compõem a especificação formal de *DoRiS*. No âmbito de focalizar a descrição da especificação no protocolo *DoRiS* e na verificação das suas propriedades, algumas fórmulas foram omitidas. No entanto,

a especificação completa está sendo anexada no final desta dissertação no apêndice A.

3.4.1 Hipóteses de modelagem

Características importantes do sistema devem ser incluídas na especificação de modo que se possa verificar propriedades interessantes. No entanto, é preciso ter cuidado para não especificar muitos detalhes devido ao problema da explosão de estados durante a verificação automática do modelo. As suposições feitas nesta seção têm por finalidade contornar este problema sem comprometer a descrição do protocolo e sua verificação.

Em primeiro lugar, como o nosso principal objetivo aqui é fornecer uma descrição formal do protocolo *DoRiS*, a fim de verificar a sua correção, supôs-se que cada estação hospeda apenas uma tarefa e/ou um processo. Desta forma, evita-se a necessidade de especificar os servidores *DoRiS*.

Em segundo lugar, representou-se o tempo por uma variável inteira. Embora tal representação discreta do tempo possa comprometer a precisão do modelo no caso de sistemas assíncronos em geral (CLARKE; GRUMBERG; PELED, 1999), ela é aceitável para protocolos síncronos baseados em troca de mensagens (LAMPORT; MELLIAR-SMITH, 2005). Uma justificativa intuitiva da correção desta hipótese é que marcas temporais inteiras podem ser associadas ao início e ao fim de cada ação de envio ou recepção de mensagens. Se estas marcas foram iguais, devido a um resolução temporal discreta, isto significa simplesmente que as ações correspondentes são concorrentes.

Em terceiro lugar, considerou-se que, sempre que uma ação especificada for habilitada, ela acontece sem demora ou é desabilitada imediatamente. Isto significa que os temporizadores são especificados sem desvios ou atrasos.

Finalmente, considera-se que todas as estações compartilham um relógio comum global, pois assume-se um modelo síncrono para evitar a especificação de detalhes de sincronização. Observa-se que, na prática, todas as estações podem sincronizar os seus relógios locais com precisão, pois as mensagens elementares são obrigatórias e periódicas. Detalhes de tal procedimento serão abordados na fase de implementação descrita no capítulo 5.

É importante notar que as considerações sobre o desvio nulo dos relógios e sobre a sincronia do sistema permite a definição de temporizadores globais na especificação, o que reduz

consideravelmente os problemas de explosão de estados.

3.4.2 Conceitos de TLA+

A Lógica Temporal de Ações (TLA) e sua linguagem formal associada (TLA+) combina a Lógica Temporal de TLA (LAMPORT, 1994) com a expressividade da lógica dos predicados e a teoria dos conjuntos de Zermelo-Fraenkel. Dotado do verificador de modelo TLC (YU; MANOLIOS; LAMPORT, 1999), TLA+ permite a especificação e a verificação tanto de protocolos de *hardware* quanto de sistemas distribuídos. Esta seção apresenta a sintaxe básica de TLA+. Quando necessário, informações complementares sobre TLA+ serão dadas ao longo da descrição da especificação de *DoRiS*. Leitores interessados numa descrição ampla de TLA+ podem se referir à publicação de Lamport (LAMPORT, 2002).

Numa especificação em TLA+, uma computação de um sistema é representada por uma seqüência de estados, também chamada de comportamento. Para caracterizar os estados do sistema, uma especificação define o conjunto de variáveis (VARIABLES) utilizado para descrever o sistema e o conjunto de constantes (CONSTANTS), que são utilizadas para definir os valores eventualmente atribuídos às variáveis. Um **estado** do sistema é, portanto, definido pela atribuição de valores constantes às variáveis da especificação.

Um par de estados consecutivos, suponha i e f em referência a inicial e final, é chamada de **passo** (*step*) e é denotado $i \rightarrow f$. O operador linha " $'$ " é utilizado para distinguir os valores de variáveis num passo. Considerando um certo passo $S : i \rightarrow f$ e uma variável v, a ocorrência de v sem linha (v) faz referência ao valor de v em i, enquanto a ocorrência de v com linha (v') faz referência ao valor de v em f.

Uma **função de estado** é um expressão na qual aparecem somente variáveis sem linhas. Tal função associa valores constantes aos estados de um comportamento. As funções de estado que associam a valores booleanos são simplesmente chamados de **predicados** de estado.

Uma **função de transição** é uma expressão na qual aparecem variáveis sem linhas e com linhas. Portanto, denotando $\mathcal{S}$ o conjunto dos passos de um sistema, e $\mathcal{C}$ um conjunto qualquer, uma função de transição F é um mapeamento de $\mathcal{S}$ sobre $\mathcal{C}$). Por exemplo, seja um passo $S : i \rightarrow f$ e v uma variável, tal como $v = 0$ no estado i e $v = 1$ no estado f, e seja F a função de transição definida por $F(S) = v' - v$, tem-se $F(S) = 1$.

Finalmente, uma **ação** é, por definição, uma função de transição a valores booleanos. Portanto, uma ação é um mapeamento de $\mathcal{S}$ sobre $\{V, F\}$, onde V e F correspondem aos valores de verdade e falso da lógica de predicados. Considerando o exemplo apresentado acima, a ação $\mathcal{A}$, definida pela expressão $\mathcal{A} \triangleq v' = v + 1$, na qual o símbolo TLA+ $\triangleq$ significa "igual, por definição", é verdadeira no passo S. Para um dado passo S, a relação de sucessão do estado i para o estado f, usualmente chamada de função de transição de estado no formalismo das Máquinas de Estado Finitos, é definida pelo conjunto de ações definidas sobre o passo S. Este conjunto também é uma ação, pois uma ação pode ser composta de várias ações.

As fórmulas temporais de TLA+, como, por exemplo, a relação de transição entre estados, são asserções booleanas sobre comportamentos. Diz-se que um comportamento satisfaz uma fórmula $\mathcal{F}$ se $\mathcal{F}$ é uma asserção verdadeira deste comportamento. O operador da lógica temporal $\Box$ é utilizado para escrever as fórmulas temporais. A semântica do operador $\Box$ é definida da seguinte maneira. Para algum comportamento Σ, e alguma ação $\mathcal{A}$, a fórmula temporal $\mathcal{F} = \Box[\mathcal{A}]_{vars}$ é verdadeira – ou simplesmente "Σ satisfaz $\mathcal{F}$" – se e somente se, para qualquer passo $S : i \rightarrow f$ de Σ que altera o conjunto $vars$ de todas as variáveis, A é verdadeira em S.

3.4.3 Elementos da linguagem TLA+

Com o intuito de facilitar o entendimento dos detalhes da especificação de *DoRiS*, esta seção agrupa algumas definições dos operadores de TLA+ utilizados.

• ***A construção "***$[e \in E \mapsto f(e)]$***"*** – Seja n o cardinal de E. A expressão $[e \in E \mapsto f(e)]$ representa a tupla de n elementos definida por $(f(e_1), f(e_2), \ldots, f(e_n))$. O símbolo $\mapsto$ é utilizado aqui para atribuir valores às entradas da tupla. Por exemplo, a expressão $[j \in Task \mapsto -1]$, representa a tupla de dimensão $nTask$ que recebe o valor -1 para todas as suas coordenadas.

• ***Tarefas e Processos*** – Para representar os conjuntos $TaskSet$ e $ProcSet$, definiam-se as tuplas Ti e Pj das tarefas e dos processos:

$$Ti \triangleq [i \in Task \mapsto \langle \text{“T”}, i \rangle] \quad \wedge \quad Pj \triangleq [j \in Proc \mapsto \langle \text{“P”}, j \rangle]$$

Utilizando estas duas tuplas, as expressões para $TaskSet$ e $ProcSet$ são dadas por:

$$TaskSet \triangleq \{Ti[i] : i \in Task\} \wedge ProcSet \triangleq \{Pj[j] : j \in Proc\}$$

O símbolo " : " tem aqui o sentido "tal que" usual na matemática.

• ***O operador "* CHOOSE *"*** – Numa expressão do tipo CHOOSE $x \in C : F(x)$, o operador CHOOSE escolhe um elemento x qualquer do conjunto C tal que (" : ") a proposição $F(x)$ seja verdadeira. Por exemplo, considerando um elemento $t \in TaskSet$ ou $p \in ProcSet$, o seu índice é encontrado pelas funções $taskId$ e $procId$ assim definidas:

$$\begin{array}{l} taskId(T) \triangleq \text{CHOOSE } i \in Task : T = Ti[i] \\ procId(P) \triangleq \text{CHOOSE } j \in Proc : P = Pj[j] \end{array}$$

Nestes exemplos, a escolha é única, pois existe um único índice verificando a proposição especificada em ambos os casos.

• ***A construção "* LET ... IN *"*** – Esta construção sintática é utilizada para definir constantes locais. O escopo das definições efetuadas na cláusula do LET é o conjunto de fórmulas da cláusula do IN. Por exemplo, na seguinte expressão

$$\begin{array}{c} \exists\, T \in TaskSet : \text{LET } i \triangleq taskId(T) \\ \text{IN } F(i) \end{array}$$

a constante i, definida como o índice da tarefa T, é conhecida apenas na proposição $F(i)$.

• ***A construção "* EXCEPT , ! *e* @ *"*** – A palavra EXCEPT , juntamente com os símbolos ! e @, são utilizados para definir as ações que envolvem tuplas. Por exemplo, a ação A, definida por

$$A \triangleq History' = [History \text{ EXCEPT } !.elem = @ + 1]$$

é verdadeira num passo $i \rightarrow f$ se a tupla $History$ for igual nos dois estados i e f, com a exceção do campo $elem$, cujo valor no estado f $(!.elem)$ deve ser igual ao seu valor no estado i (@) incrementado de 1. O símbolo " ! " é uma abreviação para o sujeito da palavra EXCEPT, e o símbolo "@", por sua vez, representa o valor, no estado i, do campo a ser modificado pela cláusula do EXCEPT. Portanto, uma sintaxe equivalente de A, porém menos concisa, seria:

$$A \triangleq History' = [History \text{ EXCEPT } History.elem' = History.elem + 1]$$

• ***A construção "* CASE , □ *e* → *"*** – O operador CASE permite a escolha múltipla de valores, baseado no caráter de verdade de um conjunto de predicados logicamente exclusivos. Um exemplo de uso desta construção será visto na definição da função $list$, apresentada na figura 3.4.

• ***O operador "*** $@@$ ***"*** $-$ Este operador é utilizado para construir tuplas a partir de tuplas menores. Considerando, por exemplo, dois conjuntos C e D disjuntos, o operador " $@@$ " pode ser definido pela fórmula seguinte:

$$[i \in C \mapsto \langle \text{“V”}, i \rangle] \, @@ \, [i \in D \mapsto \langle \text{“V”}, i \rangle] \triangleq [i \in C \cup D \mapsto \langle \text{“V”}, i \rangle]$$

Por exemplo, se $C \triangleq 1 \,..\, 3$ e $D \triangleq 4 \,..\, 7$, tem-se:

$$[i \in 1 \,..\, 3 \mapsto \langle \text{“V”}, i \rangle] \, @@ \, [i \in 4 \,..\, 7 \mapsto \langle \text{“V”}, i \rangle] \triangleq [i \in 1 \,..\, 7 \mapsto \langle \text{“V”}, i \rangle]$$

• ***O operador "***UNCHANGED ***"*** $-$ Para otimizar o algoritmo de verificação automática e garantir que a especificação de alguma ação não seja esquecida, uma especificação em TLA+ deve imperativamente especificar, para cada uma das ações principais da relação de transição, as regras de evolução de todas as variáveis do conjunto $vars$. Se uma variável v não for modificada, a ação particular $v' = v$ é utilizada. Para efeito de concisão, a palavra UNCHANGED E é utilizada para definir o conjunto E das variáveis não modificadas por uma determinada ação. Por exemplo, UNCHANGED $\langle v \rangle \triangleq v' = v$.

• ***Indentação*** $-$ Deve ser ressaltado que, em TLA+, a indentação é utilizada preferencialmente no lugar de parênteses. Desta forma, os operadores $\wedge$ e $\vee$ são utilizados para escrever fórmulas lógicas complexas sob a forma de listas indentadas.

3.4.4 Constantes e variáveis

As constantes da especificação de *DoRiS* são definidas na tabela 3.1. Um conjunto de valores possíveis para verificar *DoRiS* é, por exemplo: $nTask = 8, nProc = 7, deltaChip = 300, delta = 6, pi = 111, maxTxTime = 122$. Os valores de $nTask$ e $nProc$ são escolhidos arbitrariamente. Porém, valores muito grandes provocam uma explosão do número de estados do sistema. Os valores de delta (δ) e $maxTxTime$ correspondem aos tempos de transmissão em μs de mensagens de 72 bytes e 1524 bytes num barramento Ethernet de 100Mbps (ver seção 2.4). pi (π) é o tempo utilizado por (CARREIRO; FONSECA; PEDREIRAS, 2003). Finalmente, Δ_C deve ser maior que 2π, para garantir que as duas mensagens críticas enviadas num *chip* sejam processadas antes do início do próximo *chip*.

Outras constantes podem ser definidas no decorrer da especificação, utilizando o símbolo " $\triangleq$ ". Por exemplo, define-se $Task \triangleq 1 \,..\, nTask$ e $Proc \triangleq 1 \,..\, nProc$, os dois conjuntos

Constantes	Descrição
$nTask$	Número de tarefas participando do segmento *DoRiS*.
$nProc$	Número de processos participando do segmento *DoRiS*.
$deltaChip$	Duração (Δ_C) de um *chip*.
$delta$	Tempo de transmissão (δ) de um quadro Ethernet de tamanho mínimo.
pi	Tempo de processamento (π) de uma mensagem crítica pelo dispositivo mais lento.
$maxTxTime$	Tempo de transmissão de um quadro Ethernet de tamanho máximo.

Tabela 3.1: O conjunto CONSTANTS da especificação de *DoRiS*

de inteiros isomorfos a $TaskSet$ e $ProcSet$, respectivamente. Nestas definições, a definição do símbolo " .." é: para dois inteiros i, j tal que $i < j$, $i \ldots j \triangleq \{i, i+1, \ldots, j\}$. Outros exemplos são as definições subseqüentes dos conjuntos: $TaskSet \triangleq \{T[i] : i \in Task\}$ e $ProcSet \triangleq \{P[j] : j \in Proc\}$.

As variáveis da especificação, apresentadas na tabela 3.2, são agrupadas em quatro estruturas chamadas $Shared$, $TaskState$, $ProcState$ e $History$ de acordo com as suas características. A variável $Shared$ é uma tupla com quatro campos, utilizada para armazenar a visão distribuída compartilhada por todos. As duas outras variáveis, $TaskState$ e $ProcState$, são tuplas de dimensão $nTask$ e $nProc$, com 4 e 3 campos, respectivamente, que armazenam as informações locais de cada tarefa e processo. Finalmente, a variável $History$ é uma tupla com 2 campos, utilizada para especificar propriedades temporais. Esta variável é apenas um "observador" que não interfere na especificação do protocolo.

3.4.5 A fórmula principal de *DoRiS*

• ***Spec*** $-$ A fórmula $Spec$, apresentada na figura 3.2, é a fórmula principal da especificação de *DoRiS*.

$Init$ é o conjunto dos estados iniciais, $\Box[Next \vee Tick]_{vars}$ é a relação de sucessão, aqui composta da disjunção das duas ações $Next$ ou $Tick$, e $Liveness$ é uma condição de evolução

Variável	**Campo**	**Descrição**
$Shared$ (global)	$chipTimer$	é um temporizador crescente, que varia de 0 a Δ_C, e representa o fluxo do tempo durante um *chip*.
	$chipCount$	é um contador que identifica os *chips* módulo $nTask$.
	$medium$	representa o estado do meio. Na ausência de transmissão, $medium$ vale $\{\}$. Senão, $medium$ contém uma mensagem em fase de transmissão.
	$macTimer$	é um temporizador decrescente que indica o tempo que falta para completar a transmissão de uma mensagem. Em particular, $macTimer = 0 \implies medium = \{\}$.
$TaskState$ (tupla local)	msg	representa a lista de mensagens críticas armazenadas na memória local após recepção na placa de rede.
	$execTimer$	é um temporizador decrescente que indica o tempo de execução que falta para completar o processamento de uma mensagem crítica.
	res	é a lista de reservas existentes para os $nTask$ próximos *slots*.
	$cons$	é um contador que contabiliza o número de mensagens elementares recebidas num ciclo completo de *DoRiS*.
$ProcState$ (tupla local)	$token$	é o contador associado ao bastão circulante utilizado nas janelas $\mathcal{W}_S$.
	$list$	é a lista das mensagens não-críticas esperando para ser enviadas.
	$count$	é um contador que contabiliza o número de mensagens não-críticas recebidas durante uma janela $\mathcal{W}_S$.
$History$ (observador global)	$elem$	é um contador que contabiliza o número de mensagens elementares enviadas num ciclo.
	$rese$	é um contador que contabiliza o número de mensagens de reservas enviadas num ciclo.

Tabela 3.2: O conjunto VARIABLES da especificação de *DoRiS*

$$Spec \triangleq Init \wedge \Box[Next \vee Tick]_{vars} \wedge Liveness$$

Figura 3.2: A fórmula Spec

do sistema. $Next$ e $Tick$ são os conjuntos de ações que podem ser verdadeiras num certo passo de um comportamento. Conseqüentemente, um comportamento Σ satisfaz $Spec$ se e somente se o primeiro estado de Σ é um elemento de $Init$ e todos os passos de Σ satisfazem $Next$ ou $Tick$ e a condição $Liveness$.

• ***Init*** $-$ A fórmula $Init$, apresentada na figura 3.3, descreve o conjunto dos estados iniciais do sistema, ou seja, $Init$ é a definição dos conjuntos de valores inicias possíveis para cada variável da especificação. Aqui, cada conjunto contém apenas um elemento, pois considera-se um estado inicial único do sistema.

$$\begin{aligned}
Init \triangleq\ & \\
& \wedge Shared = [chipTimer \mapsto 0,\ chipCount \mapsto 1,\ macTimer \mapsto 0,\ medium \mapsto \{\}] \\
& \wedge TaskState = [i \in Task \mapsto [msg \mapsto \langle\rangle,\ execTimer \mapsto Infinity, \\
& \qquad\qquad\qquad\qquad res \mapsto [j \in Task \mapsto -1],\ cons \mapsto nTask - i + 1]] \\
& \wedge ProcState = [j \in Proc \mapsto [token \mapsto 1,\ list \mapsto list(j),\ count \mapsto 0]] \\
& \wedge History = [elem \mapsto 0,\ rese \mapsto 0]
\end{aligned}$$

Figura 3.3: A fórmula $Init$

Os valores definidos para os quatro campos da variável global $Shared$ indicam que o *chip* 1 está iniciando ($chipTimer \mapsto 0, chipCount \mapsto 1$), que o meio está livre ($medium \mapsto \{\}$) e que $macTimer$ é nulo ($macTimer \mapsto 0$), pois nenhuma mensagem está sendo transmitida. Assim como foi visto na seção 3.4.3, a expressão $res \mapsto [j \in Task \mapsto -1]$ significa que, no estado inicial, todas as entradas da tupla res valem -1. Lembrar da seção 3.3.3 que res é a tupla que armazena as reservas recebidas por uma tarefa. O valor arbitrário -1 é utilizado aqui para indicar que não há reserva para o *slot* correspondente.

No caso das variáveis locais, $TaskState$ e $ProcState$, as fórmulas têm o seguinte significado. Cada tarefa i do conjunto $Task$ assume que:

- A sua lista de mensagens críticas para serem processadas está vazia ($msg \mapsto \langle\rangle$)
- Conseqüentemente, o temporizador $execTimer$, utilizado para representar o tempo de execução que falta para processar um mensagem crítica, é desabilitado ($execTimer \mapsto Infinity$);
- A sua lista de reservas está vazia, ou seja, todas os campos da tupla res valem -1 ($res \mapsto [j \in Task \mapsto -1]$);
- Ela está num estado consistente ($cons \mapsto nTask - i + 1$). Lembrar que uma tarefa está num estado consistente se ela recebeu todas as mensagens elementares desde o último $\mathcal{S}_E$ no qual ela enviou uma mensagem (ver seção 3.3.3).

Cada processo j do conjunto $Proc$:

- Assume que o processo 1 está inicialmente em posse do bastão circulante ($token \mapsto 1$);
- Utiliza a função de estado $list$ (ver figura 3.4) para definir a lista da suas mensagens não-críticas.
- Assume que ele ainda não recebeu nenhuma mensagem não-crítica ($count \mapsto 0$).

Finalmente, os dois contadores de mensagens elementares e de reservas da variável de observação $History$ são iniciados com "0" ($elem \mapsto 0,\ rese \mapsto 0$).

• ***list*** $-$ A função $list$ define a lista de mensagens a serem enviadas por cada processo durante um ciclo de *DoRiS*. Esta função é escolhida arbitrariamente de acordo com o cenário de verificação desejado. Uma das possíveis definições desta função é apresentada na figura 3.4.

$$
\begin{array}{l}
list(j) \triangleq \text{CASE } j \in \{1\} \to [i \in 1\,..\,4 \mapsto [txTime \mapsto maxTxTime]] \\
\qquad\qquad \square\ j \in \{2\} \to \langle\rangle \\
\qquad\qquad \square\ j \in Proc \setminus \{1,\,2\} \to 1 :> [txTime \mapsto maxTxTime]
\end{array}
$$

Figura 3.4: A função $list$

Aqui, a função $list$ utiliza a construção TLA+ "CASE, $\square$, $\to$ " para diferenciar as tuplas de acordo com o valor de j. Em tal construção, o símbolo $\square$ é utilizado para listar todos os

casos do CASE, e o símbolo $\rightarrow$ indica o valor para ser adotado em cada caso. Neste exemplo, as tuplas de mensagens definidas para cada processo são: $list(1)$ de $P[1]$ contém 4 mensagens, cada uma com o tempo de processamento ($txTime$) de uma mensagem Ethernet de tamanho máximo ($maxTxTime$); $list(2)$ de $P[2]$ está vazia; para os demais processos, $list(j)$ contém apenas uma mensagem de tamanho máximo. Outras definições de $list(j)$ são possíveis, de acordo com o cenário de comunicação para ser verificado.

3.4.6 Ações principais

A especificação de *DoRiS* é constituída por 7 ações principais, cinco que compõem a fórmula $Next$, e duas que compõem a fórmula $Tick$. O conjunto de fórmulas de cada uma destas ações principais é dividido em dois conjuntos lógicos diferentes. O primeiro conjunto é constituído pelos predicados de estados, também chamados de guardas, nos quais figuram constantes e variáveis sem linhas. Este conjunto representa as condições de realização da ação. O segundo conjunto é constituído pelas ações – nas quais figuram constantes, variáveis sem linhas e com linhas – que especificam as operações de *DoRiS*.

• ***Next*** – Esta ação, apresentada na figura 3.5, descreve as operações do protocolo que acontecem instantaneamente, ou seja, as ações que não decrementam os temporizadores, pois suas durações são muito curtas, comparativamente com os tempos de transmissões das mensagens. Este é o caso, aqui, das operações de emissão e recepção de mensagens. Lembrar que, para as tarefas, receber não significa processar, mas somente colocar na memória local da placa de rede.

$$
\begin{aligned}
Next \;\triangleq\; & \vee \exists\, T \in TaskSet : SendElem(T) \vee SendRese(T) \\
& \vee \exists\, P \in ProcSet : SendSoft(P) \\
& \vee \exists\, msg \in Shared.medium : RecvHard(msg) \vee RecvSoft(msg)
\end{aligned}
$$

Figura 3.5: A ação $Next$

Como pode ser observado, a ação $Next$ é uma disjunção de cinco ações, correspondendo à emissão e recepção de mensagens. O uso do operador $\exists$ permite expressar a dependência de uma ação com relação ao seu argumento. Por exemplo, a ação $SendElem$ só pode satisfazer

um passo S se existir uma tarefa T tal que $SendElem(T)$ seja verdadeira em S. A primeira linha da ação $Next$ especifica a emissão de uma mensagem por uma tarefa T e a segunda, por um processo p. As duas ações $SendElem$ e $SendRese$ correspondem às janelas $\mathcal{W}_H$ – subdividida em $\mathcal{S}_E$ e $\mathcal{S}_R$ – e a ação $SendSoft$ corresponde à janela $\mathcal{W}_S$. A terceira linha especifica a recepção de uma mensagem presente no meio, de acordo com o seu tipo. $RecvHard$ corresponde à recepção de uma mensagem crítica, enquanto $RecvSoft$ corresponde à recepção de uma mensagem não-crítica.

Assim como será visto nas próximas seções, cada uma destas ações é regida por um conjunto de predicados, também chamado de condições de realizações ou guardas (*enabling conditions*). Em conseqüência da estrutura temporalmente seqüencial de *DoRiS*, este conjunto de guardas corresponde, na maioria das ações de *DoRiS*, às condições exclusivas de realização. Portanto, o operador " $\vee$ " é exclusivo em quase todas as suas ocorrências.

Considera-se agora um estado e alcançável da especificação, isto é, tal que exista uma seqüência de passos levando de $i \in Init$ a e. Suponha, ainda, que no estado e, pelo menos um dos guardas das cinco ações sejam falsos. Neste caso, não existe estado f tal que $Next$ satisfaz o passo $e \rightarrow f$. Isto pode acontecer por duas razões. O sistema ficou numa situação de bloqueio (*deadlock*) por causa de um erro de especificação ou de projeto, ou a ação $Tick$ está habilitada.

• ***Tick*** – Esta ação, definida pela fórmula $Tick \triangleq NextTick \vee NextChip$, especifica o fluxo do tempo. Para permitir a verificação de alguns modelos finitos do sistema, apesar da natureza sem limite do tempo, utilizou-se uma representação do tempo circular. Isto foi realizado dividindo a ação $Tick$ na disjunção de duas ações: $NextTick$, que incrementa o tempo por passos discretos, e $NextChip$, que realiza a transição de um *chip* para o próximo, incrementando o valor do contador $chipCount$. Se fosse só a ação $NextTick$, o temporizador $chipTimer$ poderia crescer indefinidamente. Porém, a cada vez que a ação $NextChip$ acontece, ela redefine o temporizador $chipTimer$ para o valor "0" de tal forma que este temporizador varia de "0" a Δ_C. Além disso, o contador $chipCount$ é também redefinido para o seu valor inicial 1, sempre que ele atinge o valor $nTask$. Desta forma, a representação do tempo permite verificar comportamentos do sistema durante um ciclo completo de *DoRiS*.

• ***Liveness*** – Esta restrição, definida pela fórmula $Liveness \triangleq \Box\Diamond Tick$, (na qual $\Diamond F \triangleq \neg\Box\neg F$) garante que um comportamento que satisfaz $Spec$ tenha estados em todos os *chips* de um ciclo de *DoRiS*. De fato, devido à representação circular do tempo, $Liveness$ é satisfeita apenas por comportamentos cíclicos, porém sem bloqueio, permitindo a verificação de modelos finitos de *DoRiS*.

3.4.7 O anel crítico

Assim como foi visto na seção 3.3.2, a alternância das janelas $\mathcal{W}_H$ do anel crítico e $\mathcal{W}_S$ do anel não-crítico é definida pelo mecanismo TDMA. As janelas $\mathcal{W}_H$ e $\mathcal{W}_S$ iniciam quando o valor de $chipTimer$ é igual a "0" e 2δ, respectivamente. Dentro da janela $\mathcal{W}_H$, o valor δ de $chipTimer$ indica que o *slot* de reserva começa. Além destas condições temporais, a circulação do bastão no anel crítico utiliza condições lógicas baseadas no valor de $chipCount$, o contador de *chip*. Detalhes desta condições serão dadas durante a descrição das ações correspondentes.

O anel crítico é especificado por três ações principais $SendElem$, $SendRese$ e $RecvHard$, cujas descrições são objetos desta seção.

• ***SendElem*** – Esta ação, apresentada na figura 3.6, descreve as regras que regem a emissão de uma mensagem elementar.

Antes de descrever passo-a-passo esta ação, alguns conceitos e termos precisam ser introduzidos. O mecanismo de rotação do bastão circulante utiliza o contador $chipCount$, definido módulo $nTask$, que identifica o *chip* corrente. Assim como será visto na seção 3.4.9, este contador é periodicamente incrementado pela ação $NextChip$, quando o temporizador $chipTimer$ expira, no fim de cada *chip*. O temporizador $macTimer$ especifica o tempo de transmissão de uma mensagem. Ele é igual a "0" quando o meio físico está livre. Senão, seu valor indica o tempo restante para completar a transmissão corrente. Finalmente, o campo $medium$ da variável $Shared$ representa o estado do meio. Quando uma mensagem msg é enviada, ela é armazenada na variável $medium$. Quando uma transmissão termina, o meio volta a estar livre, o que é representado pela fórmula $medium = \{\}$.

Voltando agora para a descrição da ação $SendElem$, os dois primeiros guardas estabelecem que uma tarefa T pode enviar uma mensagem elementar se: (i) o meio está livre ($Shared.medium = \{\}$); e (ii) o *chip* está começando ($Shared.chipTimer = 0$). Em se-

$$
\begin{array}{l}
SendElem(T) \triangleq \\
\quad \wedge\ Shared.medium = \{\} \\
\quad \wedge\ Shared.chipTimer = 0 \\
\quad \wedge\ \text{LET } i \triangleq taskId(T) \\
\quad\ \ \text{IN } \wedge\ Shared.chipCount = i \\
\qquad\quad \wedge\ \text{LET } resSet \triangleq reservation(i) \\
\qquad\qquad \text{IN } \wedge\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.macTimer = delta, \\
\qquad\qquad\qquad\qquad !.medium = \{[id \mapsto i,\ type \mapsto \text{“hard”},\ res \mapsto resSet]\}] \\
\qquad\qquad\qquad\ \wedge\ TaskState' = [TaskState \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad ![i].res = [j \in Task \mapsto \text{IF } j \in resSet \text{ THEN } i \text{ ELSE } @[j]], \\
\qquad\qquad\qquad\qquad ![i].cons = 1] \\
\qquad\quad \wedge\ History' = [History \text{ EXCEPT } !.elem = @ + 1] \\
\qquad\quad \wedge\ \text{UNCHANGED } ProcState
\end{array}
$$

Figura 3.6: A ação $SendElem$

guida, a primeira construção “LET ... IN” define o índice i da tarefa T (tal que $T = T[i]$) para qual a ação $SendElem$ é verdadeira e o guarda $Shared.chipCount = i$ garante que esta tarefa esteja em posse do bastão circulante. Desta forma, assegura-se que uma tarefa só pode mandar uma única mensagem elementar por ciclo de *DoRiS*.

Como pode ser observado na figura 3.6, a ação $SendElem$ altera os valores dos campos $macTimer$ e $medium$ da variável $Shared$ e $elem$ da variável $History$. Estas alterações globais têm o seguinte significado:

i) O valor δ é atribuido ao temporizador $macTimer$ para representar o tempo durante o qual o meio será ocupado pela transmissão da mensagem elementar enviada.

ii) A mensagem elementar enviada, cujo identificador é i, é do tipo “hard” e carrega a lista de reservas $resSet$, armazenada no campo $medium$.

iii) O contador $elem$ é incrementado para indicar que uma mensagem elementar foi enviada.

Nota-se que a lista de reserva $resSet$ é definida, na cláusula do segundo LET, pela função $reservation(i)$. Esta função é apresentada logo a seguir, na figura 3.7.

Finalmente, para todas as tarefas, os campos res e $cons$ da variável $TaskState[i]$ são atualizados.

i) A lista de reservas da tarefa $T[i]$ é atualizada, de acordo com a lista de reservas $resSet$ enviada.

ii) O contador $cons$ é redefinido ao valor 1. Este contador é utilizado para criar um mecanismo de tolerância a falhas de omissão, como explicado a seguir.

O contador $cons$ contabiliza o número de mensagens críticas recebidas por uma tarefa T entre duas emissões consecutivas de mensagens elementares por esta tarefa. Cada vez que uma mensagem elementar é recebida, $cons$ é incrementado. Portanto, quando uma tarefa está em posse do bastão circulante num *slot* $\mathcal{S}_E$, ela pode detectar se uma falha de omissão ocorreu desde seu último $\mathcal{S}_E$: se $cons$ for menor que $nTask$, T deixou de receber pelo menos uma das mensagens elementares eventualmente enviadas nos $nTask$ prévios *chip*. Neste caso, diz-se que T está inconsistente. Senão $cons = nTask$ indica que não houve falhas de omissão e T está consistente.

$reservation(i) \triangleq$
IF $TaskState[i].cons = nTask$
THEN $\{j \in Task : TaskState[i].res[j] = -1\}$
ELSE $\{(((i-1) + (nTask - 1)) \ \% \ nTask) + 1\}$

Figura 3.7: A função $reservation$

A função de reserva $reservation(i)$, utilizada para definir $resSet$, na ação da figura 3.6, é apresentada na figura 3.7. Esta função define a lista de reservas realizadas pela tarefa $T[i]$ para os $nTask$ próximos *chips*. Sua definição depende da necessidade de cada tarefa em largura de banda extra. A função utilizada aqui assume que todas as tarefas em estado consistente querem reservar todos os *slots* ainda não reservadas. Observa-se que a definição desta lista é diferente se uma tarefa estiver em estado inconsistente. Nesta caso, é importante ressaltar que a tarefa $T[i]$ ainda pode reservar o *slot* $\mathcal{S}_R$ do *chip* i do próximo ciclo, pois nenhuma outra tarefa pôde ainda ter reservado este *slot*.

A tupla $TaskState[i].res$ armazena a visão das reservas da tarefa $T[i]$ para os $nTask$ próximos *chips*. Na ação da figura 3.6, a visão de $T[i]$ é atualizada de acordo com as reservas presentes na lista $resSet$ enviada por $T[i]$. Se a lista $resSet$ contiver o elemento j, então $TaskState[i].res[j]$ recebe o valor i, indicando que $\mathcal{S}_R$ do próximo *chip* no qual $chipCount$ valerá j é reservado por $T[i]$. Senão, $TaskState[i].res[j]$ conserva o seu valor anterior ($@[j]$).

• ***SendRese*** − Esta ação, apresentada na figura 3.8, descreve as regras que regem a emissão de uma mensagem de reserva.

$$
\begin{array}{l}
SendRese(T) \triangleq \\
\quad \land\ Shared.medium = \{\} \\
\quad \land\ Shared.chipTimer = delta \\
\quad \land\ \text{LET } i \triangleq taskId(T) \\
\quad\ \text{IN } \land\ TaskState[i].res[Shared.chipCount] = i \\
\qquad \land\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\quad !.macTimer = delta, \\
\qquad\quad !.medium = \{[id \mapsto i,\ type \mapsto \text{“hard”},\ res \mapsto \{-1\}]\}] \\
\qquad \land\ TaskState' = [j \in Task \mapsto [TaskState[j] \text{ EXCEPT} \\
\qquad\quad !.res[Shared.chipCount] = -1]] \\
\qquad \land\ History' = [History \text{ EXCEPT } !.rese = @ + 1] \\
\qquad \land\ \text{UNCHANGED } ProcState
\end{array}
$$

Figura 3.8: A ação $SendRese$

Os dois primeiros guardas estabelecem que uma tarefa T pode enviar uma mensagem de reserva se: (i) o meio está livre ($Shared.medium = \{\}$); e (ii) o *slot* de reserva $\mathcal{S}_R$ está começando ($Shared.chipTimer = delta$). A construção “LET ... IN” define o índice i da tarefa T (tal que $T = T[i]$) para o qual a ação $SendRese$ é verdadeira e o guarda $TaskState[i].res[Shared.chipCount] = i$ garante que a tarefa $T[i]$ tem uma reserva para o *slot* $\mathcal{S}_R$ deste *chip*. Em seguida, os valores dos campos $macTimer$ e $medium$ da variável $Shared$, os campos res, $cons$ da variável $TaskState[i]$ e o campo $rese$ da variável $History$, são atualizados:

i) O valor δ é atribuído ao temporizador $macTimer$ para representar o tempo durante o

qual o meio será ocupado pela transmissão da mensagem de reserva enviada.

ii) $medium$ armazena a mensagem de reserva enviada, cujo identificador é i e o tipo é "hard". No caso das mensagens de reservas, atribui-se o valor arbitrário "$\{-1\}$" para o campo res. Desta forma, mensagens de reserva podem sempre ser diferenciadas de mensagens elementares. Tal utilização do campo res não compromete o mecanismo de reserva, pois mensagens de reserva não podem reservar *slots*.

iii) A lista de reservas da tarefa $T[i]$ é atualizada, redefinindo o valor de $TaskState[i].res[Shared.chipCount]$ para -1, indicando que a tarefa i consumiu a reserva que ela tinha para o *slot* $\mathcal{S}_R$ deste *chip*.

iv) O contador $elem$ é incrementado para indicar que uma mensagem elementar foi enviada.

Um situação possível, porém não descrita pela ação $SendRese$, ocorre quando nenhuma tarefa tem reserva para o *slot* $\mathcal{S}_R$ do *chip*. Este cenário, no qual nenhuma mensagem é enviada durante $\mathcal{S}_R$, é contemplado na ação $NextTick$ (ver figura 3.13).

• ***RecvHard*** − Esta ação, apresentada na figura 3.9, descreve as regras que regem a recepção de uma mensagem crítica.

É importante lembrar que esta ação só é habilitada se o guarda $\exists\, m \in Shared.medium$ for satisfeito assim como foi visto na ação $Next$, apresentada na figura 3.5. De fato, a recepção de uma mensagem, quer seja ela crítica ou não, requer que exista uma mensagem no meio para ser recebida. Os demais guardas contidos na ação $RecvHard$ garantem que a mensagem seja crítica ($m.type = ''\mathsf{hard}''$) e que sua transmissão tenha terminado ($Shared.macTimer = 0$).

A construção "EXCEPT" é utilizada para atualizar o estado do meio ($Shared' = [Shared$ EXCEPT $!.medium = \{\}]$). Finalmente, para representar a possibilidade de falhas de omissão, distingui-se duas possibilidades para a atualização do estado das tarefas. A primeira usa a construção "$LET \ldots IN$" para definir o conjunto $noRecvSet$ das tarefas que não recebem a mensagem m. Além da tarefa emissora ($T[m.id]$), tarefas vítimas de falhas de omissão são incluídas neste conjunto. No exemplo da figura 3.9, é o caso de $T[3]$ no segundo *chip* de cada ciclo. Na segunda possibilidade, o operador TLA+ $@@$ é utilizado para construir a tupla $TaskState'$ a partir das duas tuplas mapeadas pelos conjuntos complementares $noRecvSet$ e $Task \setminus noRecvSet$. A primeira destas tuplas corresponde às tarefas que não

$$
\begin{array}{l}
RecvHard(m) \triangleq \\
\quad \land\ m.type = \text{“hard”} \\
\quad \land\ Shared.macTimer = 0 \\
\quad \land\ Shared' = [Shared \text{ EXCEPT } !.medium = \{\}] \\
\quad \land\ \text{LET } noRecvSet \triangleq \text{IF } Shared.chipCount = 2 \text{ THEN } \{m.id, 3\} \text{ ELSE } \{m.id\} \\
\quad\ \ \text{IN } TaskState' = \\
\qquad\qquad [i \in noRecvSet \mapsto TaskState[i]]\ @@ \\
\qquad\qquad [i \in Task \setminus noRecvSet \mapsto [TaskState[i] \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.msg = Append(@, m), \\
\qquad\qquad\qquad\qquad !.execTimer = \text{IF } Len(TaskState[i].msg) = 0 \text{ THEN } pi \text{ ELSE } @, \\
\qquad\qquad\qquad\qquad !.cons = \text{IF } m.res \neq \{-1\} \text{ THEN } @ + 1 \text{ ELSE } @, \\
\qquad\qquad\qquad\qquad !.res = \text{IF } m.res = \{-1\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } [j \in Task \mapsto \text{IF } j = m.id \text{ THEN } -1 \text{ ELSE } @[j]] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } [j \in Task \mapsto \text{IF } j \in m.res \text{ THEN } m.id \text{ ELSE } @[j]]]] \\
\quad \land\ \text{UNCHANGED } \langle ProcState, History \rangle
\end{array}
$$

Figura 3.9: A ação $RecvHard$

recebem m e para quais, portanto, $TaskState$ não é alterado. A segunda corresponde às tarefas que recebem m. Para estas, os campos da variável $TaskState[i]$ são atualizados da seguinte maneira:

i) A mensagem m é armazenada na lista msg de mensagens para serem processadas;

ii) Se a lista de mensagem msg é vazia, o temporizador $execTimer$ é definido para o valor π para representar o início do processamento da mensagem pela tarefa i. Senão, ele é mantido constante.

iii) Se m é uma mensagem elementar, ($m.res \neq \{-1\}$), o contador $cons$ é incrementado. Este contador é incrementado a cada recepção de uma mensagem elementar. A omissão de uma recepção implica, portanto, que $cons$ não é incremento. Neste caso, a tarefa é dita inconsistente e sua capacidade de reserva é limitada, assim como foi visto na descrição da ação 3.6;

iv) Se m é uma mensagem de reserva, a reserva do *slot* $\mathcal{S}_R$ atual feita pela tarefa emissora

$T[m.id]$, é cancelada, redefinindo o seu valor para -1 (se m chegou, é porque $T[m.id]$ consumiu sua reserva). Senão, m é uma mensagem elementar. Neste caso, cada lista de reserva é atualizada de acordo com as reservas carregadas por m.

3.4.8 O anel não-crítico

No anel não-crítico, a circulação do bastão é organizada em função da comunicação observada. A cada mensagem não-crítica recebida, um processo $P[j]$ incrementa seu contador $token$. Quando sua vez chega, isto é, quando $ProcState[j].token = j$, $P[j]$ transmite uma mensagem, quer seja da sua fila de mensagem em espera, ou uma mensagem de tamanho mínimo para consumir o bastão.

O anel não-crítico é especificado por duas ações principais $SendSoft$ e $RecvSoft$, cujas descrições são objetos desta seção.

• ***SendSoft*** $-$ Esta ação, apresentada na figura 3.10, descreve as regras que regem a emissão de uma mensagem não-crítica.

Os dois primeiros guardas da ação $SendSoft$ descrevem a isolamento temporal entre o anel não-crítico e o anel não crítico pelo TDMA. Ou seja, uma janela $\mathcal{W}_S$ só acontece quando $2 * delta \leq Shared.chipTimer \leq deltaChip$. O terceiro guarda garante que o meio está livre ($Shared.medium = \{\}$). Em seguida, a construção "LET ... IN" define várias constantes locais:

i) i é o índice do processo p (tal que $P = P[i]$) para o qual a ação é verdadeira;

ii) $lenTX$ é o tamanho da mensagem a ser enviada. Se a lista $ProcState[i].list$ for vazia, a função $lenMsg(i)$, apresentada na figura 3.11, define o tamanho mínimo $delta$ para a mensagem a ser enviada. Senão, o tamanho da primeira mensagem na lista é utilizado ($Head(ProcState[i].list).txTime$).

iii) d é o tempo de transmissão da mensagem.

iv) $NoMsg$ é uma disjunção verdadeira se o processo $P[i]$ não emite sua mensagem. Isto ocorre se $P[i]$ está falho ($i \in Failed$) ou se não houver tempo suficiente na atual janela $\mathcal{W}_S$ para que $P[i]$ envie sua mensagem ($d > deltaChip$).

$$
\begin{array}{l}
SendSoft(P) \triangleq \\
\quad \wedge\ 2 * delta \leq Shared.chipTimer \\
\quad \wedge\ Shared.chipTimer \leq deltaChip \\
\quad \wedge\ Shared.medium = \{\} \\
\quad \wedge\ \text{LET } i \triangleq procId(P) \\
\qquad\quad lenTX \triangleq lenMsg(i) \\
\qquad\quad d \triangleq Shared.chipTimer + lenTX \\
\qquad\quad NoMsg \triangleq i \in Failed \vee d > deltaChip \\
\qquad \text{IN } \wedge\ i = ProcState[i].token \\
\qquad\quad \wedge\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\quad !.macTimer = \text{IF } NoMsg \text{ THEN } Infinity \text{ ELSE } lenTX, \\
\qquad\qquad\quad !.medium = \text{IF } NoMsg \text{ THEN } @ \text{ ELSE } \{[id \mapsto i,\ type \mapsto \text{“soft”}]\}] \\
\qquad\quad \wedge\ ProcState' = [ProcState \text{ EXCEPT} \\
\qquad\qquad\quad ![i].token = \text{IF } NoMsg \text{ THEN } @ \text{ ELSE } (@ \ \%\ nProc) + 1, \\
\qquad\qquad\quad ![i].count = \text{IF } NoMsg \text{ THEN } @ \text{ ELSE } @ + 1, \\
\qquad\qquad\quad ![i].list = \text{IF } d > deltaChip \vee @ = \langle\rangle \text{ THEN } @ \text{ ELSE } Tail(@)] \\
\qquad\quad \wedge\ \text{UNCHANGED } \langle TaskState,\ History\rangle
\end{array}
$$

Figura 3.10: A ação *SendSoft*

Observa-se que o conjunto *Failed* dos processos falhos é definido arbitrariamente de acordo com o cenário de verificação de falhas desejado. No exemplo da figura 3.11, uma falha acontece no processo 3 nos *chips* 2, 3, 4 e 5, e no processo 5 nos *chips* 3 e 5. Lembrar que o modelo de falhas, apresentado na seção 3.3.1, considera apenas falhas silenciosas.

Depois do último guarda $Shared.chipCount = i$ que garante que $P[i]$ esteja em posse do bastão circulante, as ações envolvendo as variáveis *Shared* e *ProcState* são especificadas. Distingui-se dois casos, C1 e C2 dependendo se $P[i]$ envie sua mensagem ($NoMsg$ é falso), ou não ($NoMsg$ verdadeiro):

i) C1: O valor $lenTX$ é atribuído ao temporizador $macTimer$ para representar o tempo durante o qual o meio será ocupado pela transmissão da mensagem não-crítica.

C2: O temporizador $macTimer$ é desativado ($macTimer = Infinity$).

$$
\begin{array}{l}
Failed \triangleq \text{CASE } Shared.chipCount = 2 \rightarrow \{3\} \\
\qquad\qquad \square\ Shared.chipCount \in \{3,\,4\} \rightarrow \{3,\,5\} \\
\qquad\qquad \square\ Shared.chipCount = 5 \rightarrow \{3,\,5\} \\
\qquad\qquad \square\ Shared.chipCount \in \{1\} \cup 6\,..\,nTask \rightarrow \{\}
\end{array}
$$

$$
\begin{array}{l}
lenMsg(i) \triangleq \\
\qquad \text{IF } ProcState[i].list \neq \langle\rangle \text{ THEN } Head(ProcState[i].list).txTime \text{ ELSE } delta
\end{array}
$$

Figura 3.11: O conjunto $Failed$ e a função $lenMsg(i)$

ii) C1: $medium$ armazena a mensagem não-crítica enviada, cujo identificador é i e o tipo é "soft".

C2: O meio permanece livre.

iii) C1: O contador $token$ é incrementado, pois $P[i]$ consumiu sua vez.

C2: O contador $token$ não é incrementado.

iv) C1: O contador $count$, utilizado para contabilizar quantas mensagens são enviadas durante uma janela $\mathcal{W}_S$, é incrementado.

C2: $count$ não é incrementado.

Finalmente, a lista de mensagem de $P[i]$ é atualizada. Em ambos os casos, seja com $P[i]$ falho ou enviado uma mensagem, a primeira mensagem da lista, se esta não estiver vazia, é retirada. Senão, $P[i]$ não enviou sua mensagem porque faltava tempo na janela $\mathcal{W}_S$. Neste caso, a lista permanece a mesma.

Observar que tanto falhas como omissões por falta de tempo são representadas, na ação $SendSoft$, pela desativação do temporizador $macTimer$, atribuindo-lhe o valor infinito. Ademais, a ausência de mensagem, até o fim da janela $\mathcal{W}_S$, implica que o bastão permanece com o processo $P[i]$ até a próxima janela $\mathcal{W}_S$.

• ***RecvSoft*** − Esta ação, apresentada na figura 3.12, descreve as regras que regem a recepção de uma mensagem não-crítica.

$$
\begin{array}{l}
RecvSoft(m) \triangleq \\
\quad \land\ m.type = \text{“soft”} \\
\quad \land\ Shared.macTimer = 0 \\
\quad \land\ Shared' = [Shared \text{ EXCEPT } !.medium = \{\}] \\
\quad \land\ ProcState' = [j \in \{m.id\} \mapsto ProcState[j]] \ @@ \\
\qquad\qquad [j \in Proc \setminus \{m.id\} \mapsto [ProcState[j] \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.token = (@ \ \% \ nProc) + 1, \\
\qquad\qquad\qquad\qquad !.count = @ + 1]] \\
\quad \land \text{ UNCHANGED } \langle TaskState, History \rangle
\end{array}
$$

Figura 3.12: A ação $RecvSoft$

Assim como para a ação $RecvHard$ (ver figura 3.9), o primeiro guarda desta ação, $\exists\, m \in Shared.medium$, aparece na formulação da ação $Next$ (ver figura 3.5). Os dois demais guardas da ação $RecvSoft$ garantem que a mensagem a ser é recebida é não-crítica ($m.type = \text{''soft''}$) e que sua transmissão já tenha terminado ($Shared.macTimer = 0$). Em seguida, o estado do meio é atualizado ($Shared' = [Shared \text{ EXCEPT } !.medium = \{\}]$) e a variável $ProcState$ é atualizada, distinguindo dois casos. Se a mensagem foi enviada pelo próprio processo ($j \in \{m.id\}$), $ProcState[j]$ não é alterado. Senão, os campos $token$ e $count$ são incrementados, indicando que o bastão deve ir para o próximo processo do anel não-crítico e que mais uma mensagem não-crítica foi recebida.

3.4.9 A representação temporal

A ação *Tick* organiza o progresso do protocolo e é composta de duas ações principais: $NextTick$, que incrementa o tempo por passos discretos, e $NextChip$, que define a transição entre *chips* consecutivos.

- ***NextTick*** – Esta ação, apresentada na figura 3.13, organiza o fluxo linear do tempo durante a janela temporal de um *chip*.

Assim como foi visto nas apresentações do anel crítico e não-crítico, as cinco ações principais de emissão e recepção de mensagens só podem acontecer se o temporizador $macTimer$ for

$$
\begin{array}{l}
NextTick \triangleq \\
\quad \text{LET } noRese \triangleq \ \wedge\ Shared.medium = \{\} \\
\qquad\qquad\qquad\qquad\ \wedge\ Shared.chipTimer = delta \\
\qquad\qquad\qquad\qquad\ \wedge\ \forall\, i \in Task : TaskState[i].res[Shared.chipCount] \neq i \\
\qquad tmp \triangleq \ min(\{TaskState[i].execTimer : i \in Task\} \cup \\
\qquad\qquad\qquad\qquad \{deltaChip - Shared.chipTimer\}) \\
\qquad d \triangleq \ \text{IF } noRese \text{ THEN } min(\{delta,\ tmp\}) \text{ ELSE } \ min(\{Shared.macTimer,\ tmp\}) \\
\quad \text{IN } \wedge\ d > 0 \\
\qquad\ \wedge\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad !.chipTimer = @ + d, \\
\qquad\qquad !.macTimer = \text{IF } noRese \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } @ \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE IF } @ = Infinity \text{ THEN } Infinity \text{ ELSE } @ - d] \\
\qquad\ \wedge\ TaskState' = [i \in Task \mapsto [TaskState[i] \text{ EXCEPT} \\
\qquad\qquad !.msg = \text{IF } TaskState[i].execTimer - d = 0 \text{ THEN } Tail(@) \text{ ELSE } @, \\
\qquad\qquad !.execTimer = \text{IF } @ - d = 0 \\
\qquad\qquad\qquad\qquad\qquad \text{THEN IF } Len(TaskState[i].msg) > 1 \text{ THEN } pi \text{ ELSE } Infinity \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE IF } @ = Infinity \text{ THEN } @ \text{ ELSE } @ - d]] \\
\qquad\ \wedge\ \text{UNCHANGED } \langle ProcState,\ History \rangle
\end{array}
$$

Figura 3.13: A ação $NextTick$

nulo. Nas ações de emissão, esta condição é uma conseqüência do guarda $Shared.medium = \{\}$, pois, se o meio estiver livre, a última mensagem presente no meio foi recebida e nenhuma nova mensagem foi enviada. Se o campo $macTimer$ não for nulo, todas as ações de emissão e recepção são desabilitadas, assim como a ação principal $NextChip$, apresentada na figura 3.14, pois esta ação também tem o guarda $Shared.medium = \{\}$. Assim, a ação $NextTick$ deve ser habilitada, a não ser que uma situação de bloqueio tenha sido atingida. Resumidamente, se nenhum temporizador é nulo, então o tempo deve progredir.

Na ação $NextTick$, o incremento do tempo é definido com base nos valores dos temporizadores, de tal forma que o progresso temporal ocorra em intervalos de tempo maiores possíveis. O valor deste incremento, denotado d, corresponde ao intervalo de tempo necessário para que uma das cinco ações de emissão ou recepção, ou a ação $NextChip$ seja habilitada. Para determi-

nar o valor de d, distinguem-se os temporizadores crescentes e os temporizadores decrescentes.

No caso de $chipTimer$, o único temporizador crescente, o guarda associado é $chipTimer = deltaChip$ da ação $NextChip$ (c.f. figura 3.14). O valor de d correspondente, que torna este guarda verdadeiro, é portanto $deltaChip - Shared.chipTimer$. Este valor corresponde ao tempo durante o qual o meio permanece livre, antes de o *chip* terminar. Para os temporizadores decrescentes, eles apenas habilitam uma ação quando forem nulos. O incremento de tempo d correspondente a um temporizador t decrescente, valendo x, é portanto este valor x.

Determinar d consiste, portanto, em achar o mínimo entre os temporizadores decrescentes e o valor $deltaChip - Shared.chipTimer$. Isto é realizado pela construção "LET ... IN", utilizando a função $min(S)$ (não mostrada aqui) que acha o valor mínimo num conjunto S de valores. Para representar a possibilidade que nenhuma mensagem seja enviada num *slot* $\mathcal{S}_R$ de um *chip*, a construção "LET ... IN" também define o predicado $noRese$. Os dois primeiros guardas deste predicado determinam o início de $\mathcal{S}_R$ e o terceiro garante que nenhuma tarefa tenha uma reserva para este *slot*. Em seguida, a constante temporária tmp é definida para armazenar o menor valor entre os temporizadores decrescentes $execTimer$ e o tempo que falta para terminar o *chip*. Finalmente, a constante d é definida. Se $noRese$ for verdadeiro, o conjunto de busca do menor valor contém tmp e $delta$ e não contém $macTimer$ que é nulo. Senão, o conjunto de busca contém tmp e $macTimer$.

Considere-se, então, o temporizador t cujo valor d foi selecionado como incremento do tempo. Quando este incremento é realizado pela ação $NextTick$, o valor de t, se for um temporizador decrescente, é redefinido para "0". No caso de $chipTimer$, o seu valor é redefinido para $deltaChip$. Em ambos os casos, a ação respectiva, cujo guarda é $t = 0$ ou $chipTimer = deltaChip$, é habilitada. É importante observar aqui, que, o uso de um valor de d menor que m não permitiria habilitar nenhuma nova ação, e que apenas a ação $NextTick$ continuaria sendo habilitada. De fato, depois uma progressão de $d < m$, um temporizador decrescente teria o valor $m - d > 0$ e $chipTimer$ teria o valor $chipTimer + d < deltaChip$. Portanto, o mesmo temporizador continuaria sendo a cota para o incremento do tempo, e um novo passo $NextTick$ teria que acontecer, até que, em algum momento futuro, t chegasse ao valor "0" ou $chipTimer$ chegasse ao valor $deltaChip$.

Depois de ter determinado o valor do incremento do tempo d, o único guarda desta ação ($d > 0$) garante que, se algum temporizador for nulo, a ação associada deve acontecer antes

que o tempo seja incrementado. Em seguida, todos os temporizadores da especificação são atualizados.

Dois casos são distinguidos na atualização de $macTimer$. Se $noRese$ é verdadeiro, $macTimer$ não é alterado. Senão, $macTimer$ (se não for infinito) é decrementado de d. Por fim, se o valor de d não é associado ao fim do processamento de uma mensagem, $execTimer$ (se não for infinito) é decrementado de d. Senão, o valor de d corresponde ao fim de processamento de uma mensagem crítica por uma tarefa $T[i]$. Neste caso, esta mensagem deve ser retirada da lista $TaskState[i].msg$ e o temporizador $execTimer$ associado deve ser redefinido para o valor pi se houver mais alguma mensagem para ser processada. Caso contrário, este temporizador deve ser desabilitado.

• ***NextChip*** − Esta ação, apresentada na figura 3.14, organiza a transição de um *chip* para o próximo, assim como o fluxo circular do tempo.

$$
\begin{array}{l}
NextChip \triangleq \\
\quad \land \quad Shared.medium = \{\} \\
\quad \land \quad Shared.chipTimer = deltaChip \\
\quad \land \quad \text{LET } Overflow \triangleq \exists j \in Proc : Len(ProcState[j].list) > 14 \\
\qquad\qquad NextCycle \triangleq Shared.chipCount' = 1 \\
\qquad \text{IN} \quad \land Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.macTimer = 0, \\
\qquad\qquad\qquad\qquad !.chipCount = (@ \ \% \ nTask) + 1, \\
\qquad\qquad\qquad\qquad !.chipTimer = \text{IF } Overflow \text{ THEN } -1 \text{ ELSE } 0] \\
\qquad\qquad\qquad \land ProcState' = [j \in Proc \mapsto [ProcState[j] \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.token = \text{IF } ProcState[j].count = 0 \text{ THEN } (@ \ \% \ nProc) + 1 \text{ ELSE } @, \\
\qquad\qquad\qquad\qquad !.count = 0, \\
\qquad\qquad\qquad\qquad !.list = \text{IF } NextCycle \text{ THEN } @ \circ list(j) \text{ ELSE } @]] \\
\qquad\qquad\qquad \land \text{IF } NextCycle \\
\qquad\qquad\qquad\quad \text{THEN } History' = [elem \mapsto 0, rese \mapsto 0] \\
\qquad\qquad\qquad\quad \text{ELSE UNCHANGED } History \\
\quad \land \quad \text{UNCHANGED } TaskState
\end{array}
$$

Figura 3.14: A ação $NextChip$

Os dois guardas desta ação garantem que o meio está livre ($Shared.medium = \{\}$) e que o *chip* terminou ($Shared.chipTimer = deltaChip$). Após isso, a construção " LET ... IN " define os predicados $Overflow$ e $NextCycle$. O primeiro é utilizado para detectar se a lista de mensagens de algum processo ultrapassa um valor limite arbitrário, definido aqui como 14. O segundo, $NextCycle$, é utilizado para caracterizar uma mudança de ciclo de *DoRiS*.

Em seguida, as ações envolvendo as variáveis $Shared$ e $ProcState$ são especificadas:

i) O valor "0" é atribuído ao temporizador $macTimer$ para habilitá-lo novamente.

ii) O contador $chipCount$ é incrementado (módulo $nTask$).

iii) Caso o predicado $Overflow$ seja verdadeiro, a lista de algum processo cresceu acima do valor limite tolerado e o valor arbitrário -1 é atribuído ao temporizador $chipTimer$ de forma a permitir a detecção do erro, como será vista na seção 3.5. Caso contrário, o valor "0" é atribuído ao temporizador $chipTimer$, pois um novo *chip* irá iniciar.

iv) Se nenhuma mensagem não-crítica foi recebida durante $\mathcal{W}_S$ deste *chip* ($ProcState[i].count = 0$), o processo em posse do bastão está falho. Neste caso, o contador $token$ é incrementado. Senão, seu valor é mantido inalterado.

v) O contador de mensagens não-críticas recebidas durante a última janela $\mathcal{W}_S$ é redefinido para "0", pois um novo *chip* vai começar.

vi) Se um novo ciclo tiver começando, o campo $list$ é redefinido para o valor $@ \circ list(j)$, isto é a concatenação da lista de mensagens atuais com a lista arbitrária $list(j)$, já utilizada na definição de $Init$ (ver figura 3.4). Senão, seu valor é mantido inalterado.

Finalmente, se houver mudança de ciclo, os dois contadores $elem$ e $rese$ da variável $History$ são redefinidos para "0".

Duas observações devem ser ressaltadas aqui. Em primeiro lugar, o valor arbitrário -1, atribuído ao temporizador $chipTimer$, quando uma lista de mensagem cresce indevidamente, permite desabilitar todas as ações da especificação. Desta forma, garante-se a detecção imediata do erro pelo verificador de modelo TLC, como será visto na seção 3.5.

Em segundo lugar, a circularidade temporal da especificação é obtida pela redefinição de $chipTimer$ para "0" a cada fim de *chip*, juntamente com a utilização do módulo $nTask$ no incremento de $chipCount$, responsável pela mudança de ciclo.

3.5 VERIFICAÇÃO AUTOMÁTICA

Para poder verificar alguns modelos finitos do sistema com o verificador de modelo TLC (YU; MANOLIOS; LAMPORT, 1999), um arquivo de configuração é utilizado. Este arquivo contém o valor de todas as constantes do conjunto CONSTANTS e a lista de propriedades temporais a serem verificadas. Como foi visto na seção 3.4.4, um conjunto de valores utilizado para verificar *DoRiS* foi, por exemplo: $nTask = 8, nProc = 7, deltaChip = 300, delta = 6, pi = 111, maxTxTime = 122$. Os tempos de execução para tal modelo foram bastante razoáveis, porém nenhum estudo comparativo foi realizado com outras ferramentas. Por exemplo, para um modelo com 17 tarefas e 14 processos, e usando cenários de falhas e de comunicação não-crítica, similares àqueles apresentados na seção 3.4, TLC verificou a especificação de *DoRiS* e da suas propriedades temporais em menos de um minuto num processador Intel Core Duo 2 Ghz usando a máquina virtual java com uma pilha de 512M. O código completo da especificação e do arquivo de configuração utilizados para este teste estão apresentados no apêndice A.

Após a detecção de erros de sintaxe, TLC busca possíveis situações de bloqueio (*deadlock*). Em seguida, TLC verifica se a especificação implica nas fórmulas temporais que são listadas no arquivo de configuração. Sempre que ele detecta a violação de uma propriedade, TLC produz um comportamento, que é um contra-exemplo para esta propriedade, gerando uma seqüência de estados com os valores de todas as variáveis para cada estado. A análise destes dados, chamados de rastro, é uma ferramenta valiosa para identificar a causa do erro e corrigir o protocolo (ou a sua especificação). O que se segue é a descrição de algumas fórmulas utilizadas para verificar propriedades relevantes de *DoRiS*.

• ***TypeInvariance*** – Esta propriedade, apresentada na figura 3.15, permite verificar a invariância de tipo das variáveis, ou seja, que uma variável permanece no domínio do seu tipo durante um comportamento que satisfaz a especificação. Tal verificação, para cada variável da especificação, assegura a detecção de erros óbvios.

Para construir os domínios de tipo das variáveis, a definição deste invariante utiliza duas

$$HardMsg \triangleq Seq([id : Task, type : \{\text{“hard”}\}, res : \text{SUBSET } (\{-1\} \cup Task)])$$

$$MediumMsg \triangleq \{m : m \in [id : Proc, type : \{\text{“soft”}\}] \cup [id : Task, type : \{\text{“hard”}\}, res : \text{SUBSET } (\{-1\} \cup Task)]\}$$

$$\begin{aligned}
TypeInvariance \triangleq\ & \\
& \wedge Shared.chipCount \in Task \\
& \wedge Shared.chipTimer \in 0 \,..\, deltaChip \\
& \wedge Shared.macTimer \in 0 \,..\, maxTxTime \cup \{Infinity\} \\
& \wedge \forall\, m \in Shared.medium : m \in MediumMsg \\
& \wedge ProcState \in [Proc \rightarrow [token : Proc,\ count : 0 \,..\, 50, \\
& \qquad list : \{\langle\rangle\} \cup Seq([txTime : 0 \,..\, maxTxTime])]] \\
& \wedge TaskState \in [Task \rightarrow [msg : \{\langle\rangle\} \cup HardMsg,\ res : [Task \rightarrow \{-1\} \cup Task\ ], \\
& \qquad\qquad execTimer : 0 \,..\, pi \cup \{Infinity\},\ cons : Task]]
\end{aligned}$$

Figura 3.15: A propriedade $TypeInvariant$

construções de TLA+, ainda não explicadas. A primeira é o uso da construção "$[\ldots : \ldots]$", para definir conjuntos. Por exemplo, $[id : Task]$ é o conjunto de tuplas da forma $[id \mapsto i]$ com $i \in Task$, ou seja, $[id : Task] == \{[id \mapsto i] : i \in Task\}$, expressão na qual o símbolo "$:$" tem agora o significado "tal que" usual em matemática. A segunda é o uso do símbolo $\rightarrow$ para definir conjuntos de funções. Por exemplo, $[A \rightarrow B]$ é o conjunto das funções cujo domínio é A e a imagem é B.

Uma vez especificados os conjuntos de variação para cada variável, o invariante $TypeInvariance$ verifica que, em cada estado, cada variável pertence a este conjunto. Estas fórmulas, apesar de longas e complexas, não representam propriedades alguma do sistema. Portanto, não serão descritas em mais detalhes aqui. No entanto, a verificação deste invariante é recomendada, pois permite a detecção dos erros mais óbvios da especificação. A seguir, propriedades mais relacionadas ao protocolo *DoRiS* são apresentadas.

- ***CollisionAvoidance*** – Esta fórmula temporal, apresentada na figura 3.16, permite verificar que em nenhum comportamento, emissões de mensagens podem acontecer simultaneamente. Ou seja, se uma ação de emissão é habilitada num estado, então nenhuma outra ação de emissão é habilitada no mesmo estado.

$$
\begin{array}{l}
Send(Q) \triangleq \ \lor \ \land \ Q \in TaskSet \\
\qquad\qquad\quad \land \ (\text{ENABLED } SendElem(Q) \lor \text{ENABLED } SendRese(Q)) \\
\qquad\qquad \lor \ \land \ Q \in ProcSet \\
\qquad\qquad\quad \land \ \text{ENABLED } SendSoft(Q)
\end{array}
$$

$$
\begin{array}{l}
CollisionAvoidance \triangleq \\
\quad \forall\, P,\ Q \in TaskSet \cup ProcSet : \Box(\text{ENABLED } (Send(P) \land Send(Q)) \implies (P = Q))
\end{array}
$$

Figura 3.16: A propriedade $CollisionAvoidance$

Vale a pena mencionar que, a fim de produzir rastros dos comportamentos verificados para cada propriedade, a verificação realizada por TLC sempre deve ser executada duas vezes para cada propriedade. Primeiro, verifica-se a propriedade e, em seguida, a sua contraposição. Desta forma, pode-se verificar que TLC detecta a violação da propriedade, ou da sua contra-posição, numa das duas execuções, e apenas numa. A análise do comportamento produzido como contra-exemplo permite assim conferir que a propriedade especificada expressa realmente o que era desejado verificar.

Por exemplo, $NoCollisionAvoidance$ é a contraposição do predicado $CollisionAvoidance$ que ilustra o uso de tal metodologia. Os dois rastros produzidos por TLC, numa execução com apenas uma tarefa e um processo, são comentados no apêndice B.

$$
\begin{array}{l}
NoCollisionAvoidance \triangleq \\
\quad \exists\, P,\ Q \in TaskSet \cup ProcSet : \Diamond((P \neq Q) \land \text{ENABLED } (Send(P) \land Send(Q)))
\end{array}
$$

• ***HardRingCorrectnesss*** − Nesta fórmula, apresentada na figura 3.17, algumas propriedades do anel crítico são verificadas.

$$
\begin{array}{l}
HardRingCorrectness \triangleq \\
\quad \land \forall\, T \in TaskSet : \Box(Len(TaskState[taskId(T)].msg) \leq 3) \\
\quad \land \Box(\text{ENABLED } NextChip \implies History.elem = Shared.chipCount)
\end{array}
$$

Figura 3.17: A propriedade $HardRingCorrectness$

Em primeiro lugar, verificou-se que o tamanho da lista de mensagens para serem proces-

sadas permanece abaixo de um certo limite, definido aqui como 3, e que, portanto, não há possibilidade de esgotamento da memória da placa de rede. Com os valores das constantes utilizadas nesta verificação, duas mensagens críticas podem ser enviadas em cada *chip*. Estas duas mensagens devem ser processadas antes de um novo *chip* começar, portanto, as memórias não devem conter mais de duas mensagens, o que explica o valor 3 utilizado aqui.

Em seguida, a segunda linha verifica que, quando $NextChip$ acontece, a ação $ElemSlot$ sempre foi executada exatamente $nTask$ vezes. Vale a pena observar que, para especificar esta propriedade, foi necessário utilizar o observador $History$, cujo campo $elemSlot$ é utilizado para contabilizar as mensagens elementares enviadas em cada ciclo. Este contador, redefinido para "0" no início de cada ciclo, é incrementado quando $ElemSlot$ é verdadeira. Assim, no final de cada ciclo, $History.elemSlot$ deve ser igual a $nTask$ se todas as tarefas enviaram suas mensagens elementares. Em outras palavras, a ação $ElemSlot$ é regularmente habilitada. O fato que esta ação seja regularmente realizada é assegurada pelo guarda $i = ChipCount$, presente na fórmula da ação $SendElem$ (ver figura 3.6).

Lembrar que falhas de omissão foram especificadas na ação de recepção de mensagens críticas (ver figura 3.9). Isto implica que falhas de envio de mensagens e falhas de paradas também foram modeladas. Por exemplo, uma falha de omissão na emissão de uma mensagem pode ser vista como um conjunto de falhas de omissões na recepção, em todos as estações, e uma falha de parada de uma estação corresponde a uma falha permanente de omissão de emissão. Portanto, não foi necessário verificar cenários específicos de falhas de emissão, nem de falhas de paradas de tarefas.

• ***ReservationSafety*** $-$ Esta propriedade, apresentada na figura 3.18, garante que, se uma tarefa j pode enviar uma mensagem num *slot* $\mathcal{S}_R$ de um *chip*, então, neste *chip*: (i) a tarefa j tem uma reserva para $\mathcal{S}_R$, e (ii) cada uma das demais tarefas estão cientes desta reserva, ou não tem reserva nenhuma para aquele *slot*.

Em particular, esta fórmula implica que duas tarefas não podem ter reservas para o mesmo *slot* $\mathcal{S}_R$. Juntamente com o guarda $Reser[i][ChipCount] = i$ da ação $SendRese$ (ver figura 3.8), a especificação implica também que uma tarefa $T[i]$ só pode enviar uma mensagem de reserva num *slot* que ela reservou anteriormente.

$$
\begin{array}{l}
ReservationSafety \triangleq \\
\quad \Box \forall\, chip,\, j \in Task : \wedge \text{ENABLED } SendRese(T[j]) \\
\qquad\qquad\qquad\qquad \wedge Shared.chipCount = chip \\
\qquad \Longrightarrow \wedge TaskState[j].res[chip] = j \\
\qquad\qquad \wedge \forall\, i \in Task \setminus \{j\} : TaskState[i].res[chip] \in \{j,\ -1\}
\end{array}
$$

Figura 3.18: A propriedade $ReservationSafety$

• ***SoftRingFairness*** − Esta propriedade, apresentada na figura 3.19, permite verificar que todos os processos receberão o bastão circulante em algum momento futuro e que todas as listas de mensagens de processos não falhos serão esgotados.

$$
\begin{array}{l}
SoftRingFairness \triangleq \\
\quad \wedge \forall\, i \in Proc : \Box\Diamond(i = ProcState[i].token) \\
\quad \wedge \Box\Diamond(\forall\, i \in Proc \setminus Failed : Len(ProcState[i].list) = 0)
\end{array}
$$

Figura 3.19: A propriedade $ReservationSafety$

É importante observar que a segunda linha desta fórmula não é detectada como falsa por TLC em comportamentos nos quais a lista de mensagem dos processos crescem indefinidamente. Em tais comportamentos, TLC não consegue parar de criar novos estados, e portanto, não pode verificar, se, no futuro, um destes estados satisfará a condição. No entanto, tais comportamentos são detectados na ação $NextChip$, pelo predicado $Overflow$, assim como foi explicado na descrição da figura 3.14.

3.6 CONCLUSÃO

Neste capítulo, a descrição de *DoRiS*, um protocolo de comunicação baseado em Ethernet, e da sua especificação e verificação em TLA+, foram apresentados. *DoRiS* foi projetado para os sistemas de tempo real modernos, que exigem previsibilidade, tolerância a falha e flexibilidade. Para tal trabalho, a linguagem TLA+ mostrou ter uma capacidade de expressão poderosa, mantendo um nível de abstração elevado.

Como pôde ser visto, foi possível verificar propriedades interessantes do protocolo *DoRiS*. Em resumo, a verificação automática permitiu garantir que: (i) o protocolo provê isolamento das comunicações críticas e não-críticas, evitando colisões; (ii) cada tarefa elementar sempre envia uma mensagem por ciclo e a capacidade das memórias locais nunca é esgotada, (iii) o mecanismo de reservas é seguro e correto; (iv) a comunicação não-crítica satisfaz critérios de justiça; e (v) tolerância a falhas é garantida.

Do ponto de vista do desenvolvimento de *software*, a abordagem utilizada para conceber *DoRiS* mostrou o benefício que o uso de métodos formais pôde trazer. De fato, a utilização de TLA+ e das suas ferramentas permitiu uma metodologia de desenvolvimento interativa, na qual especificação e verificação foram realizados durante a fase de concepção das funcionalidades do protocolo.

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 necessária do mecanismo de controle do bastão circulante. No entanto, as modificações subseqüentes da especificação formal foram simples de escrever, devido ao aspecto modular de TLA+.

O produto final deste capítulo concretizou-se num expressivo aumento da maturidade e confiança dos desenvolvedores no comportamento do protocolo. Por conseguinte, o trabalho de implementação apresentado no próximo capítulo foi significativamente facilitado.

CAPÍTULO 4

PLATAFORMA OPERACIONAL

4.1 INTRODUÇÃO

Nos capítulos que precedem, o protocolo *DoRiS* foi descrito e os seus objetivos foram apresentados. Depois desta fase de definição, especificação formal e validação do protocolo *DoRiS*, é chegado o momento de apresentar um outro desafio deste projeto de desenvolvimento de um protocolo de comunicação de tempo real baseado em Ethernet. Isto é, a escolha de uma plataforma operacional de tempo real híbrida, de código livre, que possa ser utilizada tanto em computadores de uso geral quanto em dispositivos dedicados e que atende aos requisitos temporais necessários para a implementação do protocolo *DoRiS*.

É importante notar aqui que o desafio principal concentra-se em torno do uso dos computadores de uso geral. De fato, os dispositivos dedicados utilizam sistemas embarcados ou micro-controladores para oferecer um conjunto de funcionalidades definidas. Integrar tais dispositivos num sistema distribuído requer levar em conta as suas restrições específicas como, por exemplo, capacidade de processamento, taxa de transferência, consumo de energia, etc. No entanto, a qualidade dos serviços oferecidos por tais dispositivos é uma conseqüência do seu projeto. Conseqüentemente, uma vez especificados os requisitos do sistema, o dispositivo pode ser escolhido adequadamente de tal forma a satisfazer estes requisitos.

No caso dos computadores de uso geral, a situação é bastante diferente, pois o primeiro objetivo a ser alcançado é o desempenho geral do sistema. No entanto, as tecnologias modernas utilizadas nas arquiteturas de computadores atuais para aumentar o desempenho dos dispositivos de hardware são muitas vezes fontes de imprevisibilidade temporal. Isto é, por exemplo, o caso da memória *cache*, do acesso direto à memória (DMA), do co-processamento, da predição de instruções, das unidades *multicore* ou dos *pipelines*. Devem também ser mencionadas as funções de gerenciamento da energia, que, por afetar a freqüência de execução do processador, dificultam a estimativa dos piores casos dos tempos de execução dos programas. Tal comple-

xidade do hardware, embora torne os sistemas computacionais mais velozes, aumenta o grau de imprevisibilidade do sistema e dificulta a estimativa dos atributos necessários à teoria do escalonamento de Sistemas de Tempo Real (LIU; LAYLAND, 1973; AUDSLEY et al., 1993; TINDELL; BURNS; WELLINGS, 1994). No entanto, reduzir o conjunto de serviços oferecidos por um Sistema Operacional de Propósito Geral (SOPG) no patamar dos SOs que podem garantir previsibilidade equivale a negar as evoluções do hardware das últimas décadas. Para resolver este desafio, várias abordagens foram propostas ao longo do tempo para desenvolver plataformas operacionais determinísticas baseadas em SOPG.

Este capítulo apresenta algumas destas abordagens com o objetivo de escolher a plataforma operacional que oferecerá suporte à implementação de *DoRiS*. Como esta deverá ser de software livre, usar a família de sistemas Linux parece ser uma tendência natural. Inicialmente, uma explicação geral sobre as principais características de Linux é apresentada na seção 4.2. Como será visto, Linux oferece vários aspectos que impedem seu uso direto na implementação de *DoRiS*. Tais aspectos serão detalhados na seção 4.3. Em seguida, algumas das tendências que têm incorporado características de sistemas de tempo real em Linux serão descritas na seção 4.4. Por fim, resultados comparativos de algumas plataformas serão apresentados na seção 4.5 e a escolha de uma destas será justificada, de acordo com as necessidade de *DoRiS*.

4.2 LINUX: UM SISTEMA OPERACIONAL DE PROPÓSITO GERAL

Depois de uma breve justificativa da escolha do Linux para este trabalho, esta seção apresenta os principais elementos do sistema Linux que causam ou que são relacionados à existência de imprevisibilidade temporal na execução das aplicações.

4.2.1 Motivação para o uso de Linux

A escolha da plataforma operacional no contexto deste trabalho se deu pelas características do protocolo *DoRiS* e pelo contexto universitário do seu desenvolvimento. Resumidamente, os seguintes critérios foram adotados para determinar uma plataforma adequada para desenvolvimento de *DoRiS*:

i) ser de código livre e aberto (licença GPL);

ii) garantir desvios máximos da ordem da dezena de micro-segundos para as aplicações de controle via rede, conforme o padrão de qualidade de serviço oferecida pelos Sistemas Operacionais de Tempo Real (SOTR);

iii) permitir o uso de aplicações multimídia, banco de dados e outros componentes de software distribuídos tipicamente usadas em ambientes multi-usuários de Sistemas Operacionais de Propósito Geral (SOPG);

iv) prover suporte às aplicações embarcadas usando componentes de prateleira.

A primeira destas condições decorre do modelo de pesquisa e desenvolvimento defendido neste trabalho. Apesar de a ecologia política ser um assunto essencial aos olhos deste mesmo, foge do escopo deste trabalho apresentar os elementos referindo a este tópico. A leitura de André Gorz (GORZ, 1977) ou Milton Santos (SANTOS, 2000) deverá saciar a curiosidade dos mais interessados.

A segunda condição decorre dos requisitos temporais das aplicações de controle e corresponde também às margens de desvios necessárias para a implementação do protocolo *DoRiS*, conforme visto no capítulo 3. Finalmente, o terceiro e quarto itens são diretamente relacionados com as metas do protocolo *DoRiS*, que já foram amplamente discutidas nos capítulos 2 e 3.

Dentre as soluções de SOPG, o sistema Linux (L. TORVALDS et al., 2008; BOVET, 2005) é um software livre e de código aberto que se distingue pela originalidade da sua proposta de desenvolvimento sob licença GNU/GPL (GNU *General Public License*). Uma conseqüência direta deste modelo de desenvolvimento cooperativo é que o *kernel* Linux conta com uma das maior comunidade de desenvolvedores espalhada pelo mundo inteiro. Outra vantagem do *kernel* Linux é ser baseado no sistema UNIX e oferecer uma interface de utilização conforme o padrão POSIX *Portable Operating System Interface* (IEEE, 2004). Por fim, deve ser observado que Linux é bastante difundido nos ambientes de pesquisa acadêmica.

Por todas estas razões, a plataforma Linux apareceu como uma candidata pertinente para este projeto de pesquisa. No entanto, como será visto mais adiante, o *kernel* Linux por si só não oferece as garantias temporais definidas no item (iv) acima. Portanto, revelou-se necessário pesquisar soluções existentes para tornar o SOPG Linux determinista. Algumas destas soluções são apresentadas ao longo deste capítulo, assim como a extensão Linux de tempo real adotada no contexto deste trabalho.

Antes de abordar a caracterização das propriedades temporais do Linux e de algumas de suas extensões de tempo real, são apresentados brevemente alguns conceitos básicos de sistemas operacionais que são utilizados intensivamente ao longo deste capítulo.

4.2.2 Interrupções

Na classe do sistema do tipo UNIX (BACH, 1986), na qual se encontra o sistema Linux, o SO, ou simplesmente, *kernel*, executa num modo diferente dos demais processos: o modo "protegido". Diz-se que os processos, associados às aplicações, executam em modo "usuário". Estes dois modos são definidos no nível do hardware do processador e são utilizados para restringir o acesso aos dispositivos de hardware da máquina. Por este meio, garante-se que os únicos processos que podem obter acesso aos dispositivos de hardware são aqueles executando em modo protegido. Responsável pelo gerenciamento dos recursos, o *kernel* provê uma camada de abstração dos dispositivos de hardware aos processos usuários. A interface do Linux é oferecida através de trechos de código denominados "chamadas de sistema". Cada uma destas chamadas oferece uma API (Interface de Programação da Aplicação) padronizada pela norma POSIX (IEEE, 2004), para facilitar tanto o trabalho dos programadores, quanto a portabilidade dos códigos de software.

A organização da comunicação entre os dispositivos, o *kernel* e as aplicações é baseada no conceito de interrupção do processador. Tais interrupções são gerenciadas por um dispositivo de hardware específico, o PIC ou o APIC (*Advanced Programmable Interrupt Controller*) que é diretamente conectado ao processador. Na ocorrência de um evento de comunicação, ou seja, quando um dispositivo requer a atenção do processador, ele informa ao APIC via uma linha de interrupção (*IRQ lines*). Por sua vez, o APIC informa ao processador que uma interrupção precisa ser tratada.

Distinguem-se as interrupções síncronas e assíncronas. As primeiras são geradas pelo próprio processo em execução no final do ciclo de execução de uma instrução. Elas correspondem ao uso de instruções específicas pelo programa, tais como as chamadas de sistema da interface do *kernel*, ou podem ser causadas pela execução de uma instrução indevida. As interrupções assíncronas são geradas pelos dispositivos de hardware para informar ao processador a ocorrência de um evento externo e podem ser geradas a qualquer instante do ciclo do processador.

Quando o processador percebe uma interrupção, quer seja síncrona ou assíncrona, ele desvia sua execução e passa a executar o tratador de interrupção associado à linha de interrupção que solicitou o APIC inicialmente. Esta execução não é associada a processo algum, mas sim à ocorrência de um evento e, portanto, não tem contexto de execução próprio. O tratador simplesmente executa no contexto do último processo que estava executando no processador.

Para minimizar o impacto das interrupções sobre a execução dos processos regulares, um tratador de interrupção é geralmente subdividido em três partes. A primeira parte executa operações críticas que não podem ser atrasadas e que modificam estruturas de dados compartilhadas pelo dispositivo e o *kernel*. Tais operações são executadas imediatamente e com as interrupções desabilitadas. Também executado imediatamente pelo tratador, mas com as interrupções habilitadas, são as operações rápidas que modificam apenas as estruturas de dados do *kernel*, pois estas são protegidas por mecanismos de *locks*, assim como será vista na seção 4.2.4.1. Estes dois conjuntos de operações constituem a parte crítica do tratador, durante a qual a execução não pode ser suspensa. Finalmente, as operações não-críticas e não-urgentes são possivelmente adiadas e executadas com as interrupções habilitadas. Estas execuções são chamadas de `softirqs` ou `tasklets`.

Do tratamento eficiente das interrupções depende a capacidade do sistema para reagir a eventos externos. O *kernel* padrão garante esta eficiência, proibindo que a execução da parte crítica do tratador de interrupção seja suspensa, mas permitindo que a parte não-crítica, e geralmente mais demorada, seja suspensa para a execução da parte crítica de uma outra interrupção.

4.2.3 Tempo compartilhado

O principal objetivo de um Sistemas Operacional de Propósito Geral (SOPG), tal como Linux, é oferecer o melhor serviço possível para o uso compartilhado por vários usuários de recursos limitados, tal como processadores, memória, disco, placas de rede e outros dispositivos de *hardware* (BACH, 1986; TANENBAUM, 2001; OLIVEIRA; CARISSIMI; TOSCANI, 2001). Um usuário do sistema, compartilhando um conjunto de recursos, deverá ter a ilusão que está sozinho atuando naquele sistema. O SOPG deve, portanto, garantir o acesso dos usuários a todos os recursos que ele gerencia com latências imperceptíveis para um ser humano. A implementação de tal serviço, cuja qualidade depende altamente da subjetividade de cada usuário

e das aplicações que ele precisa executar, utiliza o mecanismo de compartilhamento temporal dos recursos. Para dar a impressão de exclusividade e continuidade dos serviços aos usuários, os recursos são alocadas sucessivamente às aplicações por fatias de tempos curtas, da ordem de alguns milisegundos. A alternância rápida destas alocações garante que cada aplicação ganhe o acesso ao recurso com uma freqüência suficiente para que não haja tempo ocioso perceptível do ponto de visto do usuário. O modelo de tempo compartilhado caracteriza assim os sistemas multi-programáveis e multi-usuários.

No Linux e em SOPG similares, o entrelaçamento das execuções dos processos ao longo do tempo é realizado da seguinte maneira. O tempo é dividido em intervalos de tamanho iguais. Para este efeito, o SO utiliza o PIT (*Programable Interrupt Timer*), baseado, nas arquiteturas PCs i386, num de hardware dedicado - o chip 8254 ou seu equivalente. O PIT é programado para gerar uma interrupção periódica, o *tick*, cujo período, chamado de $jiffy$, é configurável, variando entre 1 e $10\mu s$ em função das arquiteturas.

A cada *tick* uma interrupção ocorre. Esta provoca a atualização e execução eventual dos temporizadores do sistema assim como a chamada do escalonador quando isto se faz necessário. Conseqüentemente, quanto menor o $jiffy$, mais freqüentes serão as ativações de cada processo, melhorando assim a capacidade de reação do sistema. Por outro lado, com um $jiffy$ pequeno, a alternância entre os processos é mais freqüente, aumentando o tempo gasto em modo *kernel*, no qual o processador só executa tarefas de gerenciamento. Portanto, escolher a freqüência dos *ticks* constitui um compromisso entre o desempenho do sistema e a resolução desejada para escalonar os processos. Observa-se, em particular, que a implementação do tempo compartilhado por *ticks* de duração constante faz com que um processo não possa "dormir" por um tempo menor do que um *jiffy*.

Além disso, o algoritmo de gerenciamento dos temporizadores, chamado de "roda dos temporizadores", pode constituir uma outra fonte de sobrecarga nos sistemas que utilizam muitos temporizadores. Este algoritmo utiliza uma estrutura de armazenamento baseada em 5 faixas de $jiffies$ correspondendo a intervalos de tempo que crescem exponencialmente (BOVET, 2005; MOLNAR, 2005). Cada faixa armazena os temporizadores de acordo com os seus valores de instantes de expiração. A primeira faixa corresponde aos temporizadores com tempos de expiração contidos no intervalo indo de 1 a $256\,jiffies$. A segunda corresponde aos temporizadores expirando entre $257\,jiffies$ e $16384\,jiffies$ e assim por diante até a quinta faixa que corresponde

a todos os temporizadores expirando depois de $67108865\,jiffies$. Quando um temporizador é criado, ele é armazenado na faixa que contém o *jiffy* no qual ele deve expirar. A cada 256 $jiffies$, depois que todos os temporizadores da primeira faixa sejam eventualmente disparados, o sistema executa a rotina chamada "cachoeira" que atualiza as diferentes faixas, cascateando os temporizadores de faixa em faixa. Por exemplo, esta rotina determina quais são os temporizadores da segunda faixa que vão expirar nos próximos 256 $jiffies$ e os transferem para a primeira faixa. Similarmente, a rotina transfere os devidos temporizadores da terceira para a segunda faixa, da quarta para terceira e da quinta para a quarta.

Além de utilizar uma estrutura de dados de tamanho limitado, este algoritmo se torna muito eficiente quando os temporizadores são apagados antes de serem disparados. Isto ocorre, por exemplo, no caso de uma estação servidor Internet na qual a grande maioria dos temporizadores são cancelados rapidamente. Neste caso, a sobrecarga causada pela execução da rotina da cachoeira passa a ser insignificante, pois os temporizadores são cancelados antes de serem cascateados. Por outro lado, percebe-se que a diminuição da duração do *jiffy* aumenta a sobrecarga gerada por este algoritmo, pois a rotina da "cachoeira" acaba sendo executada mais freqüentemente. Além disso, sendo o tempo entre cada execução menor, o número de temporizadores ainda não cancelados é maior. Conseqüentemente, o número de transferências de faixa a ser realizado para cada temporizador, antes do seu cancelamento eventual, aumenta.

Na sua publicação inicial, o *kernel* 2.6 passou a usar um *jiffy* de $1ms$. Percebeu-se então que este valor gerou uma sobrecarga significativa no sistema, notadamente devido aos temporizadores utilizados pelas conexões TCP. Este fato explica em parte porque as versões do *kernel* posteriores a 2.6.13 vêm com o valor padrão do *jiffy* de $4ms$, e não mais de $1ms$.

4.2.4 Preempção

No contexto dos escalonadores baseados em prioridade, um processo de baixa prioridade pode ser suspenso no decorrer da sua execução para ceder o processador a uma processo de prioridade mais alta. Quando tal evento ocorre, diz-se que houve preempção do processo em execução pelo processo de mais alta prioridade. De forma geral, diz-se que houve preempção de um processo A por um processo B, quando o processo A deve interromper sua execução para ceder o processador ao processo B. No caso dos processos executando em modo usuário,

a preempção corresponde à alternância de processos que o *kernel* executa para a implementação do mecanismo de tempo compartilhado. Este procedimento se torna mais complexo quando se trata da preempção de processos executando em modo *kernel*.

Diz-se, de maneira simplificada, que o *kernel* é preemptivo se uma alternância de processos pode acontecer quando o processador está em modo protegido. Considere, por exemplo, um processo A executando um tratador de exceção associado a uma chamada de sistema. Enquanto A está executando, o tratador de uma interrupção de hardware acorda um processo B mais prioritário que A. Num *kernel* não preemptivo, o *kernel* completa a execução da chamada de sistema de A antes de entregar o processador para o processo B. No caso de um *kernel* preemptivo, o *kernel* suspende a execução da chamada de sistema para começar a executar B imediatamente. Eventualmente, depois da execução de B, o processo A será escalonado novamente e a chamada de sistema poderá ser finalizada.

O principal objetivo de tornar o SO preemptivo (BOVET, 2005) é diminuir o tempo de latência que o processo de mais alta prioridade pode sofrer antes de ganhar o processador. Este objetivo é de suma importância para que um SOPG tal como Linux possa oferecer as garantias temporais encontradas em STR.

4.2.4.1 Concorrência e sincronização Um dos desafios em tornar o *kernel* preemptivo é garantir a integridade dos dados, mesmo que vários caminhos de controle do *kernel* possam ter acesso aos mesmos dados de forma concorrente. Este é um problema fundamental e é comumente conhecido como problema da exclusão mútua (DIJKSTRA, 1965; LAMPORT, 1974; RAYNAL, 1986; MELLOR-CRUMMEY; SCOTT, 1991; LAMPORT; MELLIAR-SMITH, 2005). No entanto, já que as soluções adotadas pelo *kernel* fazem parte das principais causas de imprevisibilidade temporal do Linux padrão, vale a pena apresentar brevemente estas soluções.

Chama-se de região crítica qualquer recurso ou estrutura de dados que só pode ser utilizado por um processo (ou um conjunto de processos) de forma atômica. Isto é, se um processo P entra numa região crítica, nenhum outro processo pode entrar nesta mesma região crítica enquanto P não a liberou. Uma maneira simples de garantir a exclusão mútua num sistema monoprocessado é desabilitar a possibilidade de preempção durante a execução de uma região crítica. Esta solução, bastante utilizada nas versões do *kernel* não superiores à versão 2.4, tem dois inconvenientes relevantes. Primeiro, ela só funciona em sistemas monoprocessados

e, segundo, ela não impede o acesso da região crítica por tratadores de interrupção. Neste segundo caso, para garantir a exclusão mútua, um processo entrando numa região crítica que é compartilhada com tratadores de interrupções deve também desabilitar aquelas interrupções.

Nas suas versões mais recentes, a partir da versão 2.6, o *kernel* tenta reduzir ao máximo o uso de tais soluções, que comprometem o desempenho do sistema em termos de capacidade reativa. No entanto, existem situações nas quais o uso destes mecanismos é necessário. Para este efeito, a implementação da exclusão mútua em contextos multiprocessados e/ou em tratadores de interrupções utiliza duas primitivas básicas de sincronização: os semáforos e os *spin-locks*.

4.2.4.2 Semáforos Um semáforo (TANENBAUM, 2001; BOVET, 2005) é constituído de um contador, de duas funções atômicas `up` e `down` e de uma fila. Quando um processo quer entrar numa região crítica, ou de forma equivalente, quando este quer adquirir um recurso compartilhado que só pode ser adquirido simultaneamente por um número limitado de processos, ele chama a função `down` que decrementa o contador do semáforo. Se o valor resultante é positivo ou nulo, o processo adquiriu o semáforo. Senão, ele é suspenso depois de ter sido colocado na fila de espera. Após um processo terminar de usar o recurso, ele executa a função `up` que incrementa o valor do contador e acorda o primeiro processo da fila. O valor inicial n do contador define o número máximo de processos que podem adquirir este semáforo simultaneamente. No caso $n = 1$, o semáforo é simplesmente chamado de *mutex*.

Observa-se que o tempo que um processo fica suspenso, esperando por um semáforo, é imprevisível. Ele depende de quantos processos já estão esperando por aquele semáforo e do tempo que cada um deles permanecerá na região crítica. Além disso, um processo é autorizado a dormir enquanto ele está em posse de um semáforo. Estes fatos fazem com que os tratadores de interrupção que não são autorizados a dormir não possam usar semáforos.

Um outro ponto importante a ser observado é a ausência de "dono" do semáforo na implementação pelo *kernel* padrão. Isto é, quando um processo P adquire um semáforo, ele se torna "dono" deste. Mas esta informação não é disponível para os demais processos que possam tentar adquirir o semáforo enquanto P o detém. As conseqüências deste aspecto de implementação são discutidas na seção 4.3.4

4.2.4.3 Spin-lock Nos ambientes multiprocessados, o uso de semáforo nem sempre é eficiente. Imagine o seguinte cenário: um processo P_1 executando num processador Π_1 tenta adquirir um mutex S, que já foi adquirido pelo processo P_2 que executa num outro processador Π_2. Conseqüentemente, P_1 é suspenso e o *kernel* entrega no processador Π_1 para um outro processo P_3. Mas, se a estrutura de dados protegida por S for pequena, P_2 pode executar a função `up` antes mesmo que P_3 comece a executar. Se P_1 é mais prioritário que P_3, uma nova alternância será realizada pelo *kernel* para permitir que P_1 volte a executar no processador Π_1. Percebe-se então que, quando o tempo de execução de uma troca de contexto é maior do que o tempo de execução na região crítica, o uso de semáforos deve ser evitado. Nestes casos, a solução é utilizar os mecanismos de trancas e de espera ocupada fornecidos pelos *spin-locks*.

No *kernel* padrão, um *spin-lock* é uma variável booleana que é utilizada de forma atômica. Só um processo pode adquirir um *spin-lock* num dado instante. Quando um processo tenta adquirir um *spin-lock* que já está em posse de um outro processo, ele não é suspenso mas sim executa uma espera ocupada (*spin*), tentando periodicamente adquirir o *lock*. Isto evita as trocas de contextos do cenário acima. Como uma região crítica protegida por um *spin-lock* há de ser curta, um processo que adquire um *spin-lock* não pode ser suspenso. Portanto, a preempção é desabilitada enquanto um processo está em posse do *lock*. Além disso, quando um caminho de controle do *kernel* C utiliza um *spin-lock* que pode ser adquirido por um tratador de interrupção, ele precisa desabilitar as interrupções. Caso contrário, se uma interrupção ocorre enquanto C está em posse do *lock*, o tratador causa a preempção do processo C e fica em espera ocupada, tentando adquirir o *lock* que C detém. Mas, como o processador está ocupado pelo tratador, C não pode mais executar, e portanto, não pode devolver o *lock*, resultando no *deadlock* do sistema. Para impedir que tal cenário aconteça, o *kernel* adota a solução de desabilitar as interrupções durante um *spin-lock* que pode ser adquirido por um tratador de interrupções. Apesar de resolver o problema, esta solução pode aumentar significativamente o tempo de resposta do sistema na ocorrência de uma interrupção.

Observa-se que a implementação de *spin-locks* no *kernel* padrão não utiliza fila e que, assim como os semáforos, os *spin-locks* não têm "donos".

4.3 CARACTERIZAÇÃO DO COMPORTAMENTO TEMPORAL DO LINUX

Esta seção apresenta as diferentes métricas adotadas para caracterizar o comportamento temporal do SOPG Linux.

4.3.1 Precisão temporal de escalonamento

De acordo com (PICCIONI; TATIBANA; OLIVEIRA, 2001), o mecanismo de compartilhamento do tempo (discutido na seção 4.2.3) limita a resolução temporal disponível para escalonar os processos, utilizando o escalonador do *kernel*, a até duas vezes o valor do $jiffy$. Para observar este fato, definiu-se aqui o seguinte cenário ilustrado na figura 4.1. Seja ε e δ dois números arbitrários positivos e menores que $jiffy$. No instante T_1, um *tick* ocorre. Logo em seguida, no instante $t_1 = T_1 + \delta$, o processo corrente P executa a chamada de sistema `sleep` pedindo para dormir durante um intervalo de tempo $jiffy + \varepsilon$. O escalonador do *kernel*, que não trabalha com frações de $jiffy$, arredonda este valor para $2jiffy$, o múltiplo logo superior. Além disso, este tempo só começa a ser descontado a partir do instante do próximo *tick* T_2. Portanto, o temporizador associado ao `sleep` acorda P no instante $T_2 + 2jiffy$. Ao final, P dormiu $3jiffy - \delta$ ao invés de dormir $jiffy + \varepsilon$.

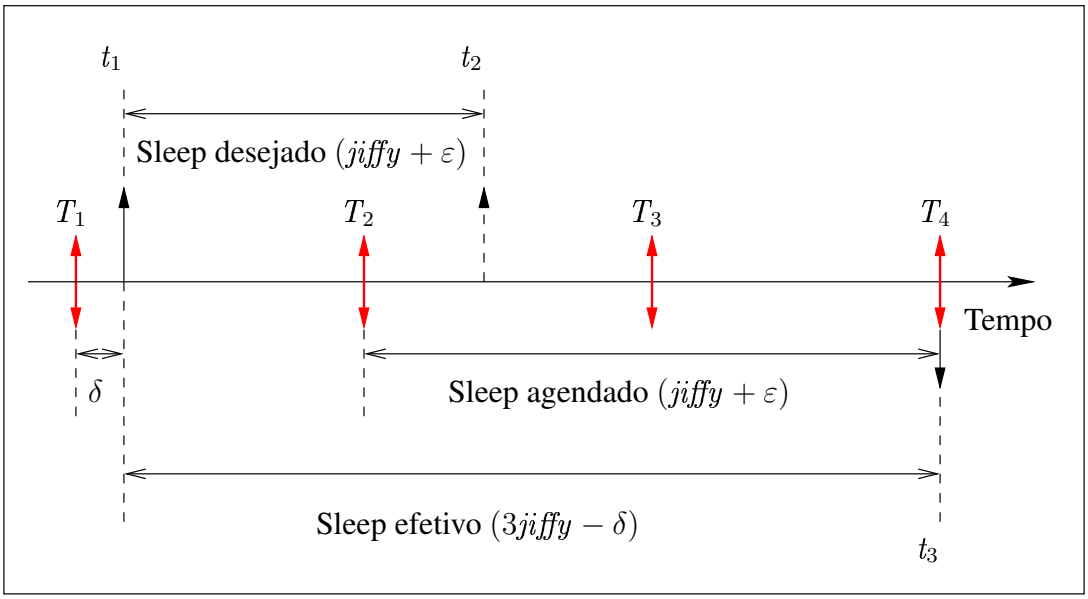


Figura 4.1: Latência causadas pela existência do *tick*

Para ilustrar o efeito da granularidade do *tick* na precisão de escalonamento (*scheduling jitter*), montamos um experimento com o *kernel* 2.6.19.7 com preempção do *kernel* habilitada (`CONFIG_PREEMPT`). O *tick* utilizado foi o valor padrão do kernel, $jiffy = 4ms$, correspondendo a uma freqüência de $250Hz$. Para o experimento, um processo executou sozinho e com a prioridade máxima em modo *single*, ou seja, com a carga mínima possível no sistema. Este processo foi programado para executar as operações seguintes, apresentadas sob forma de pseudo-código:

```
bef := read_tsc
while(1) {
     usleep(jiffy + ε)
     now := read_tsc
     write(now - bef)
     bef := now
}
```

Onde `read_tsc` é uma função que lê o valor do *Time Stamp Counter* (TSC), `bef` e `now` são duas variáveis que armazem o valor lido e $\varepsilon = 2ms$. A operação $write$ é efetuada na memória e requer um tempo muito inferior a $1ms$.

O resultado ótimo esperado para os valores das diferenças, se não fosse a existência do *tick*, seria de $6ms$. A figura 4.2 apresenta o tempo realmente observado entre duas chamadas sucessivas da chamada de sistema `usleep`. As duas execuções mostradas correspondem aos dois cenários de execução diferentes observados entre várias realizações deste experimento. Após o primeiro laço, os valores permanecem sempre iguais, portanto, só foram mostrados os resultados obtidos para os 5 primeiros laços.

Observa-se que, nas duas execuções, depois da primeira chamada, os processos sempre dormem $8ms$, apesar do pedido de $6ms$ passado para a chamada `usleep`. Isto é uma conseqüência direta do valor do *jiffy* de $4ms$, pois $8ms$ é o menor múltiplo do $jiffy$ maior que $6ms$. Observa-se também que na primeira chamada da primeira execução, o processo chega a dormir $11ms$ enquanto que na segunda execução, o processo dorme aproximadamente $7ms$. Este dois cenários diferentes ilustram a dependência do tempo de dormência no instante relativo no qual a primeira chamada `usleep` é efetuada em relação ao instante no qual o *tick* ocorre, conforme explicado no início desta seção.

É importante observar que a variabilidade discutida nesta seção caracteriza o comporta-

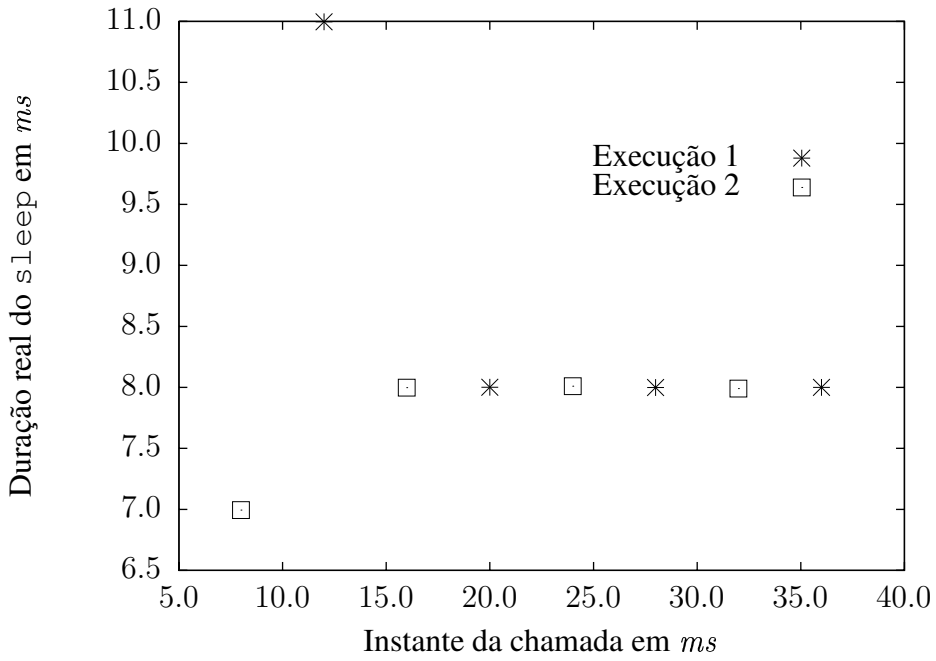


Figura 4.2: Tempo de dormência com chamadas `usleep` de 6 ms

mento do escalonador do Linux. No contexto deste trabalho, esta variabilidade não tem relevância, pois as plataformas de tempo real estudas utilizam um escalonador próprio, baseados em temporizadores de alta precisão. Portanto, a variabilidade de escalonamento não foi contemplada nos experimentos, pois não serve para efeito de comparação. No entanto, escolheu-se apresentar esta conseqüência da existência do *tick* devido à sua importância na implementação dos SO modernos.

4.3.2 Latência de interrupção

Como foi visto na seção 4.2.2, as interrupções de hardware são assíncronas e podem acontecer em qualquer momento do ciclo de execução do processador. Além disso, a execução da parte crítica de um tratador de interrupção requer eventualmente a desabilitação das interrupções para impedir o acesso concorrente a dados protegidos por *spin-locks* dentro de tratadores de interrupção.

Portanto, quando uma interrupção acontece, vários cenários de latência para a sua detecção e seu tratamento pelo processador são possíveis. Se as interrupções forem habilitadas, ela é

detectada no final do ciclo das instruções em execução. Senão, a interrupção pode acontecer enquanto as interrupções estão desabilitadas pela execução da parte crítica de um tratador de interrupção. Após o fim da execução deste tratador, as interrupções voltam a ser habilitadas novamente.

Em seguida à detecção da interrupção, o processador começa por gravar o contador de programa e alguns registradores da memória para poder retomar a execução do processo interrompido, depois do tratamento da interrupção. Observe que um tratador de interrupção executa no contexto do último processo executado. Conseqüentemente, a troca de contexto necessária para executar o tratador no processador é bastante rápida. Depois de executar mais algumas operações, tal como ler o vetor de interrupção no controlador de interrupção e carregar as informações necessárias no seus registradores, o processador finalmente começa a executar o tratador da interrupção que aconteceu. O tempo decorrido entre o instante no qual a interrupção aconteceu e o início da execução do tratador associado é chamado de **latência de interrupção** (*interrupt latency*).

A latência de interrupção caracteriza a capacidade do sistema para reagir a eventos externos. Portanto, esta grandeza foi contemplada como métrica para efeito de comparação das plataformas estudas (ver seção 4.5).

4.3.3 Latência de ativação

Quando uma interrupção ocorre, quer seja porque um temporizador expirou ou porque um evento de hardware ocorreu, o tratador da interrupção executa imediatamente a parte crítica. Lembrar que a palavra crítica faz aqui referência às primitivas de sincronização e as estruturas de dados utilizadas, no contexto de um sistema preemptível e com acesso concorrente aos dados (ver seção 4.2.2). A parte não-crítica do tratador é executada num *softirq* logo após o retorno da parte crítica do tratador. No entanto, entre o instante no qual a interrupção ocorre e o instante no qual o *softirq* começa a executar, outras interrupções podem acontecer, provocando um possível atraso na execução da parte não-crítica.

Nas plataformas de tempo real, eventos de temporizadores ou de *hardware* são utilizados para disparar tarefas, num modelo similar aos *softirqs*. Tal tarefa, muitas vezes periódica, tem um contexto próprio e fica suspensa, na espera de um evento. Quando o evento ocorre, a

interrupção associada aciona o seu tratador, que, por sua vez, acorda a tarefa. O intervalo de tempo entre os instantes no qual o evento ocorre e o início da execução da tarefa associada é chamada de **latência de ativação**. Assim como no caso dos *softirqs*, a latência de ativação pode ser aumentada pela ocorrência de interrupções. Além disso, a execução de outros *softirqs* pode ser escalonada com alguma política (ex: FIFO, prioridade fixa), o que pode também gerar interferências na latência de ativação.

Um outro aspecto importante diz respeito à implementação dos temporizadores utilizados para agendar as tarefas de tempo real. Mais especificamente, a obtenção de uma precisão de micro-segundos nos eventos disparados por temporizadores requer a utilização de relógios de alta precisão, distintos daqueles usados pelo *kernel* padrão. Desta forma, a latência de ativação passa a ser independente do escalonador de processos do Linux e do valor do `jiffy`.

Assim como a latência de interrupção, a latência de ativação caracteriza a capacidade de um sistema em reagir aos eventos externos. Portanto, esta grandeza foi também contemplada com métrica para efeito de comparação das plataformas estudas (ver seção 4.5).

4.3.4 Latência causada pela inversão de prioridade

Para organizar o compartilhamento dos recursos, os processos utilizam as primitivas de *locks* que foram descritas na seção 4.2.4.1. Quando um processo quer obter o acesso a um recurso, ele adquire o *lock* associado. Uma vez em posse do *lock*, um processo utiliza o recurso e, em algum momento futuro, devolve o *lock* quando ela não precisa mais do recurso.

Este mecanismo de reserva pode causar latência de ativação em contradição com as políticas de prioridades definida no sistema. Considere-se, primeiramente, um cenário envolvendo dois processos P_A e P_B, o primeiro de alta prioridade e o segundo de baixa prioridade. Suponha que P_B adquire um recurso compartilhado R, e que, enquanto P_B está utilizando R, uma interrupção de *hardware* acorda P_A. Então, P_A causa a preempção de P_B e adquire o processador. Possivelmente, P_A tenta adquirir o *lock* do recurso R. Porém, P_B não liberou o *lock* ainda e, conseqüentemente, P_A tem que esperar que P_B ganhe o processador novamente e complete a sua execução, pelo menos até devolver o *lock*. Só então P_A pode adquirir o processador e conseguir o recurso R. Percebe-se que, neste cenário, o processo de mais alta prioridade acaba esperando pelo processo de mais baixa prioridade.

Esta situação simples pode se tornar ainda pior no seguinte caso. Suponha agora que um (ou mais) processo P_M de prioridade média, mais alta que a prioridade de P_B e mais baixa que a prioridade de P_A comece a executar enquanto P_B está em posse do recurso R. P_A não pode executar enquanto P_B não libera o recurso R e P_B não pode executar enquanto P_M não libera o processador. Portanto, o processo de mais alta prioridade pode ficar esperando um tempo indefinido, enquanto tiver processos de prioridade intermediária ocupando o processador.

Duas soluções para este problema, chamado de inversão de prioridade, foram propostas por Sha et al em 1990 num trabalho pioneiro (SHA; RAJKUMAR; LEHOCZKY, 1990). A primeira utiliza o conceito de herança de prioridade. Resumidamente, enquanto um processo de baixa prioridade utiliza um recurso, ele herda a prioridade do processo de maior prioridade que está esperando por aquele recurso. Desta forma, e na ausência de bloqueios encadeados, o tempo máximo que um processo de alta prioridade terá de esperar é o tempo máximo que um processo de mais baixa prioridade pode bloquear o recurso. A segunda solução consiste em determinar em tempo de projeto, a mais alta prioridade dos processos que compartilham um certo recurso. Esta prioridade teto será então atribuída a qualquer um destes processos durante sua utilização deste recurso. Apesar de não impedir a inversão de prioridade, estas soluções permitem limitar o tempo máximo de espera de um processo de mais alta prioridade no sistema, aumentando, portanto, o grau de previsibilidade do sistema. Outras soluções são baseadas em protocolos sem *locks* ou na replicação dos recursos (TANENBAUM, 2001).

A ocorrência de inversão de prioridade depende altamente das aplicações e do uso que elas fazem dos recursos compartilhados. Além disso, as latências por inversão de prioridade sofridas pelas aplicações resultam também das latências de interrupção e de ativação. Ambos os fatos dificultam a elaboração de experimentos de medição. Portanto, esta grandeza não foi utilizada como métrica no contexto deste capítulo.

4.4 SOLUÇÕES DE SOTR BASEADAS EM LINUX

Nesta seção, apresentaremos os princípios de três abordagens diferentes para tornar o Sistema Operacional Linux de Tempo Real. A primeira, descrita na seção 4.4.1, consiste em tornar o *kernel* Linux inteiramente preemptível. Desta forma, é possível limitar as latências máximas de interrupção e de ativação. Uma outra abordagem, descrita na seção 4.4.2, consiste em or-

ganizar a ativação das tarefas de tempo real a partir dos tratadores de interrupção, criando uma interface de programação acessível em modo usuário. Finalmente, uma terceira abordagem, baseada em *nanokernel*, será descrita na seção 4.4.3. Esta proposta utiliza uma camada intermediária entre o *kernel* e o hardware, o *nanokernel*, que oferece serviços de tempo real para as aplicações. Apesar de existir em outras propostas de SOTR baseadas no Linux (ex: (BARABANOV, 1997; SRINIVASAN et al., 1998; CALANDRINO et al., 2006)), estas três abordagens são bastante representativas para permitir a exposição dos princípios fundamentais destinados ao aumento da previsibilidade de um SOPG tal como Linux.

4.4.1 O *patch* PREEMPT-RT

Há alguns anos, Ingo Molnar, juntamente com outros desenvolvedores do *kernel* Linux (MCKENNEY, 2005; ROSTEDT; HART, 2007), têm trabalhando ativamente para que o próprio *kernel* possa oferecer serviços de tempo real confiáveis. Este trabalho resultou na publicação do *patch* do *kernel* chamado "PREEMPT-RT" em abril de 2006 e cujo código está disponível na internet (I. MOLNAR et al., 2008).

Para resolver o problema da precisão temporal do escalonador baseado em *ticks* descrito na seção 4.3.1, o *patch* PREEMPT-RT utiliza uma nova implementação dos temporizadores de alta resolução desenvolvida por Thomas Gleixner e documentada no próprio código do *kernel* Linux (L. TORVALDS et al., 2008). Baseados no registrador *Time Stamp Counter* (TSC) da arquitetura Intel ou em relógios de alta resolução, esta implementação oferece uma API que permite obter valores temporais com uma resolução de micro-segundos. Desde a versão do *kernel* 2.6.21, esta API faz parte da linha principal do *kernel*. De acordo com resultados apresentados (ROSTEDT; HART, 2007), os tempos de latência de ativação obtidos usando esta API são da ordem de algumas dezenas de micro-segundos e não dependem mais da freqüência do *tick*. Com PREEMPT-RT, o *tick* continua sendo estritamente periódico. No entanto, com a proposta do KURT-Linux (SRINIVASAN et al., 1998), que disponibiliza o *patch* $UTIME$, é possível utilizar um *tick* aperiódico e programável com uma resolução de alguns micro-segundos.

O segundo problema diz respeito aos tempos de latência de interrupção e de preempção. Além de utilizar temporizadores de alta precisão, o *patch* PREEMPT-RT comporta várias modificações para tornar o *kernel* totalmente preemptível. Este objetivo é essencial para garantir

que, quando um processo de mais alta prioridade acorda, ele consiga adquirir o processador com uma latência mínima, sem ter que esperar o fim da execução de um processo de menor prioridade, mesmo que este esteja executando em modo *kernel*. Como foi visto nas seções 4.2.4.1 e 4.3.2, a utilização das primitivas de sincronização que permitem garantir a exclusão mútua de regiões críticas introduz possíveis fontes de latência e indeterminismo. Para eliminar estas fontes de imprevisibilidade, PREEMPT-RT modifica estas primitivas de maneira a permitir a implementação de um protocolo complexo, baseado em herança de prioridade. Por exemplo, um *spin-lock* (ou um *mutex*) agora possui um dono, uma fila de processos em espera e pode sofrer preempção, ou seja, um processo possuindo um *spin-lock* pode ser suspenso. Os atributos dono e fila são necessários para a implementação do protocolo baseado em herança de prioridade (SHA; RAJKUMAR; LEHOCZKY, 1990), como foi visto na seção 4.3.

Uma outra modificação importante diz respeito ao tratamento das interrupções. No *kernel* padrão, quando uma interrupção acontece, a parte crítica do tratador da interrupção é executada logo que a interrupção é detectada pelo processador, podendo, conseqüentemente, atrasar a execução de um outro tratador ou processo de maior prioridade. Para diminuir esta causa de latência, o *patch* PREEMPT-RT utiliza *threads* de interrupções. Quando uma linha de interrupção é inicializada, um *thread* é criado para gerar as interrupções associadas a esta linha. Na ocorrência de uma interrupção, o tratador associado simplesmente mascara a interrupção, acorda o *thread* da interrupção e volta para o código interrompido. Desta forma, a parte crítica do tratador de interrupção é reduzida ao seu mínimo e a latência causada pela sua execução, além de ser breve, é determinística. O *thread* acordado será eventualmente escalonado, de acordo com a sua prioridade e os demais processos em execução no processador.

De acordo com os resultados obtidos (ROSTEDT; HART, 2007; SIRO; EMDE; MCGUIRE, 2007), o *patch* PREEMPT-RT permite reduzir as latências do *kernel* padrão para valores da ordem de algumas dezenas de micro-segundos. Portanto, usando códigos confiáveis, respeitando as regras de programação do *patch* e alocando os recursos de acordo com os requisitos temporais, a solução PREEMPT-RT tem a vantagem de oferecer o ambiente de programação do sistema Linux, dando acesso às bibliotecas C e ao conjunto de software disponível para este sistema.

4.4.2 Uma caixa de areia em espaço usuário

A proposta de caixa de areia (QI; PARMER; WEST, 2004; FRY; WEST, 2007) tem o objetivo de permitir a extensão do *kernel* Linux padrão para oferecer uma interface de programação para tarefas de tempo real executadas em modo usuário. A idéia fundamental aplicada aqui é criar uma extensão do espaço de memória de todos os processos executados no sistema. Esta implementação utiliza o gerenciamento por páginas da memória virtual e não requer nenhum dispositivo de hardware específico. Basicamente, uma caixa de areia corresponde a uma ou duas páginas da memória virtual que são adicionadas a todas as áreas de memória dos processos do sistema. Desta forma, qualquer código da caixa de areia pode ser executado em modo usuário no contexto de qualquer processo.

Para oferecer as extensões da caixa de areia, o *kernel* é modificado através de módulos carregados, cujas funcionalidades são utilizadas via *ioctls*, de uma forma semelhante aos controladores de dispositivos. Através da interface de programação, um processo P pode registrar serviços na caixa de areia. Para utilizar estes serviços, por exemplo, durante a execução de um tratador de interrupção, o *kernel* usa uma função de *upcall* que acorda um *thread* previamente criado pelo processo P. Este *thread* executa em modo usuário, no contexto do último processo que estava executando no processador. Como este processo, possivelmente diferente de P, tem a caixa de areia na sua área de memória virtual, o *thread* acordado tem acesso a todas as funcionalidades registradas nesta área de memória compartilhada.

O tamanho da caixa de areia é arbitrário, mas deve ser suficiente para comportar uma pilha de execução dos *thread* e para conter uma versão pequena da biblioteca C padrão. Para isto, duas páginas de $4Mb$ são utilizadas. Uma página, chamada "pública", dá direito de leitura e execução, tanto em modo usuário quanto em modo *kernel*. Esta página contém as funções da interface de programação e as funções registradas pelos processos quando eles são criados. A outra página, dita "protegida", dá direito de leitura e escrita em modo *kernel*, mas só pode ser escrita por um *thread* executando em modo usuário durante um *upcall*. Isto garante que os dados de um processo, contidos na caixa de areia, não podem ser alterados por outros processos.

Nesta implementação, os serviços da caixa de areia são obtidos a partir da parte não crítica do tratador de interrupção (*softirq*). Apesar de esta parte dos tratadores ser executada com certa imprevisibilidade comparativamente com a parte crítica, esta escolha é justificada pelos autores

pelo fato de permitir aos códigos da caixa de areia fazerem chamadas de sistemas bloqueantes. Resultados apresentados mostram que os tempos de latência de interrupção são da ordem de dezenas de micro-segundos, inclusive no caso de interrupções geradas pela placa de rede na recepção de mensagens Ethernet. No entanto, no caso de eventos de redes, os autores produzem desvios padrões da mesma ordem de grandeza que as latências medidas. Apesar deste fato, o uso da caixa de areia é advogado pelos autores (FRY; WEST, 2007) para sistemas de tempo real não-críticos. No contexto deste trabalho, esta solução não foi contemplada, pois *DoRiS* tem requisitos de tempo real críticos.

4.4.3 *Nanokernel*

As soluções de implementação para SOTRs baseadas em *nanokernel*, sendo as mais divulgadas RT-Linux (BARABANOV, 1997; V. YODAIKEN et al., 2008), *Real Time Application Interface* (RTAI) (DOZIO; MANTEGAZZA, 2003; P. MANTEGAZZA et al., 2008) e Xenomai (GERUM, 2005; P. GERUM et al., 2008) são as únicas, até o momento, que alcançam latências da ordem do micro-segundos e dão suporte a sistemas críticos. Estas soluções utilizam uma camada de indireção das interrupções, chamada camada de abstração do hardware (HAL), localizada entre o *kernel* e os dispositivos de hardware e disponibilizam uma interface de programação para serviços de tempo real. Observa-se que os códigos fontes do Xenomai e RTAI são disponíveis sob licença GNU/GPL. No caso do RT-Linux, uma versão profissional é desenvolvida sob licença comercial. Uma outra versão livre é disponibilizada, sob licenças específicas, com uma interface de utilização restrita e sem suporte para as versões do *kernel* posteriores a 2.6.9.

Do ponto de vista da implementação, os *nanokernel* de Xenomai, RTAI e RT-Linux utilizam o mecanismo de virtualização das interrupções, também chamada de indireção de interrupção, introduzido na técnica de "proteção otimista das interrupções" (STODOLSKY; CHEN; BERSHAD, 1993). No contexto da interação do *nanokernel* com Linux, esta técnica pode ser resumida da seguinte maneira. Quando uma interrupção acontece, o *nanokernel* identifica se esta é relativa a uma tarefa de tempo real ou se a interrupção é destinada a um processo do *kernel* Linux. No primeiro caso, o tratador da interrupção é executado imediatamente. Caso contrário, a interrupção é enfileirada e, em algum momento futuro, entregue para o *kernel* Linux quando nenhuma tarefa de tempo real estiver precisando executar. Quando o *kernel* Linux precisa

desabilitar as interrupções, o *nanokernel* deixa o *kernel* Linux acreditar que as interrupções estão desabilitadas. No entanto, o *nanokernel* continua a interceptar qualquer interrupção de *hardware*. Nesta ocorrência, a interrupção é tratada imediatamente se for destinada a uma tarefa de tempo real. Caso contrário, a interrupção é enfileirada, até que o *kernel* Linux habilite suas interrupções novamente.

4.4.3.1 RT-Linux No caso da versão livre do RT-Linux, tanto a camada HAL quanto API de programação são fornecidas em um único *patch* que modifica o código fonte do *kernel* Linux. Este *patch* permite então co-existência do *nanokernel* RT-Linux que oferece garantias temporais críticas para as tarefas de tempo real e do *kernel* Linux padrão. O modelo de programação das tarefas de tempo real é baseado na inserção de módulos no *kernel* em tempo de execução. Portanto, estas tarefas devem necessariamente executar em modo protegido, o que restringe a interface de programação fornecida pelo RT-Linux.

Para realizar a virtualização das interrupções, as funções que manipulam as interrupções `local_irq_enable`, antigamente `sti` (*set interruption*) e `local_irq_disable`, antigamente `cli` (*clear interruption*), são modificadas. Para dar o controle à camada HAL, as novas funções não alteram mais a máscara real de interrupção, mas utilizam uma máscara virtual. Enquanto esta máscara virtual indicar que as interrupções são desabilitadas, depois de uma chamada `local_irq_disable` pelo *kernel* Linux, o *nanokernel* intercepta as interrupções que não são de tempo real e trata as interrupções de tempo real imediatamente. Quando o *kernel* chama a função `local_irq_disable`, a máscara virtual de interrupção é modificada e o *nanokernel* entrega as interrupções pendentes para Linux.

Resumindo, no RT-Linux, o *kernel* Linux é uma tarefa escalonada pelo *nanokernel* RT-Linux como se fosse a tarefa de mais baixa prioridade no sistema.

4.4.3.2 Adeos, RTAI e Xenomai No caso dos projetos RTAI (DOZIO; MANTEGAZZA, 2003) e Xenomai (GERUM, 2005), a camada HAL é fornecida pelo *Adaptative Domain Environment for Operating Systems* (Adeos), desenvolvida por Karim Yaghmour (YAGHMOUR, 2001). Adeos, fornecida por um *patch* separado, tem os seguintes objetivos:

- permitir o compartilhamento dos recursos de hardware entre diferentes sistemas operaci-

onais e/ou aplicações específicas;

- contornar a padronização dos SO, isto é, flexibilizar o uso do hardware para devolver o controle aos desenvolvedores e administradores de sistema;
- oferecer uma interface de programação simples e independente da arquitetura.

Para não ter que iniciar a construção de um sistema operacional de tempo real completo, o *nanokernel* Adeos utiliza Linux como hospedeiro para iniciar o hardware. Logo no início, a camada Adeos é inserida abaixo do *kernel* Linux para tomar o controle do hardware. Após isto, os serviços de Adeos podem ser utilizados por outros sistemas operacionais e/ou aplicações executando conjuntamente ao *kernel* Linux.

A arquitetura Adeos utiliza dois conceitos principais, domínio e canal hierárquico de interrupção. Um domínio caracteriza um ambiente de execução isolado, no qual se pode executar programas ou até mesmo sistemas operacionais completos. Domínios diferentes são associados ao SO Linux e às aplicações mais específicas como RTAI ou Xenomai. Um domínio enxerga Adeos mas não enxerga os outros domínios hospedados no sistema.

O canal hierárquico de interrupção, chamado *ipipe*, serve para priorizar a entrega de interrupções entre os domínios. Quando um domínio se registra no Adeos, ele é colocado numa posição no *ipipe* de acordo com os seus requisitos temporais. Adeos utiliza então o mecanismo de virtualização das interrupções para organizar a entrega hierárquica das interrupções, começando pelo domínio mais prioritário e seguindo com os menos prioritários. Funções apropriadas (*stall/unstall*) permitem bloquear ou desbloquear a transmissão das interrupções através de cada domínio.

Para prover serviços de tempo real, as interfaces RTAI ou Xenomai utilizam o domínio mais prioritário do *ipipe*, chamado "domínio primário". Este domínio corresponde, portanto, ao núcleo de tempo real no qual as tarefas são executadas em modo protegido. Como as interrupções são entregues começando pelo domínio primário, o núcleo de tempo real pode escolher atrasar, ou não, a entrega das interrupções para os demais domínios registrados no *ipipe*, garantindo, desta forma, a execução das suas próprias tarefas. Módulos podem ser utilizados para carregar as tarefas, de forma semelhante ao RT-Linux.

Nas plataformas Xenomai e RTAI, o "domínio secundário" corresponde ao *kernel* Linux. Neste domínio, o conjunto de bibliotecas e software usual do Linux está disponível. Em con-

trapartida, as garantias temporais são mais fracas, dado que o código pode utilizar as chamadas de sistemas bloqueantes do Linux.

Para oferecer o serviço de tempo real em modo usuário chamado LXRT, RTAI utiliza o mecanismo de associação (*shadowing*) de um *thread* do núcleo de tempo real com um processo usuário executando no domínio Linux. De acordo com sua prioridade, este *thread*, também chamado de "sombra" do processo, executa o escalonamento rápido do seu processo associado.

O projeto Xenomai se distingue do RTAI/LXRT. Além de não utilizar o mecanismo de associação (*shadowing*), Xenomai tem por objetivo privilegiar o modelo de programação em modo usuário. O modelo de tarefas executando no modo protegido só está sendo mantido para dar suporte às aplicações legadas. A implementação dos serviços de tempo real em modo usuário se baseia nas seguintes regras:

- A política de prioridade utilizada para as tarefas de tempo real, e para o escalonador associado, é comum aos dois domínios.
- As interrupções de hardware não podem impedir a execução de uma tarefa prioritária enquanto ela estiver no domínio secundário.

A primeira destas garantias utiliza um mecanismo de herança de prioridade entre o domínio primário e o domínio secundário. Quando uma tarefa do domínio primário migra para o secundário, por exemplo, porque ela efetua uma chamada de sistema do Linux, esta tarefa continua com a mesma prioridade, maior que a de qualquer processo do Linux. Este mecanismo garante notadamente que a preempção de uma tarefa de tempo real executando no domínio secundário não possa ser causada por uma tarefa de menor prioridade executando no domínio primário. A segunda garantia é obtida por meio de um domínio intermediário, chamado de "escudo de interrupção", registrado no *ipipe*, entre o primeiro domínio (o *nanokernel*) e o segundo domínio (o *kernel* Linux). Enquanto uma tarefa de tempo real está executando no segundo domínio, o "escudo de interrupção" é utilizado para bloquear as interrupções de hardware destinadas ao Linux. No entanto, aquelas interrupções destinadas à tarefa em execução são entregues imediatamente.

Para completar esta arquitetura, as chamadas de sistema executadas por uma tarefa enquanto está no domínio primário e secundário são interceptadas por Adeos que determina a função a

ser executada, seja ela do Linux padrão ou da API do Xenomai. Isto permite que uma tarefa executando no domínio secundário possa obter os serviços providos por Xenomai.

Observa-se que quando uma tarefa está no domínio secundário, ela pode sofrer uma latência de escalonamento devido à execução de alguma sessão não preemptiva do Linux. Portanto, Xenomai se beneficia do esforço de desenvolvimento do *patch* PREEMPT-RT, que tem por objetivo tornar o *kernel* inteiramente preemptível. Apesar de ainda constituir um *patch* separado, várias propostas do grupo de desenvolvedores deste *patch* já foram integrados na linha principal do *kernel* 2.6, melhorando significativamente suas capacidades de preempção.

4.5 METODOLOGIA EXPERIMENTAL

Em geral, realizar medições precisas de tempo no nível dos tratadores de interrupção pode não ser tão simples. De fato, o instante exato no qual uma interrupção acontece é de difícil medição, pois tal evento é assíncrono e pode ser causado por qualquer dispositivo de *hardware*. Para obter medidas das latências de interrupção e ativação confiáveis com alto grau de precisão, aparelhos externos, tais como osciloscópios ou outros computadores, são necessários. No entanto, como o objetivo do presente trabalho não foi medir estas latências de forma precisa, mas caracterizar e comparar o grau de determinismo das plataformas operacionais estudadas, adotou-se uma metodologia experimental simples e efetiva, que pode ser reproduzida facilmente em outros contextos.

4.5.1 Configuração do experimento

O dispositivo experimental utilizou três estações: (1) a estação de medição E_M, na qual os dados foram coletados e onde temos uma tarefa de tempo real τ à espera de eventos externos; (2) a estação de disparo E_D, que foi utilizada para enviar pacotes Ethernet com uma freqüência fixa à estação E_M; e (3) a estação de carga E_C, utilizada para criar uma carga de interrupção na estação E_M. As estações de disparo e carga foram conectadas à estação de medição por duas redes Ethernet distintas, conforme ilustrado no diagrama da figura 4.3.

As chegadas em E_M dos pacotes enviados por E_D servem para disparar uma cascata de eventos na estação E_M, permitindo a simulação de eventos externos via a porta paralela (PP).

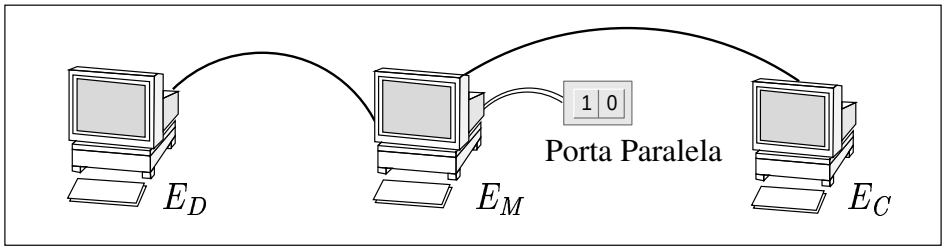


Figura 4.3: Configuração do experimento.

Mais explicitamente, cada chegada de um pacote Ethernet na placa foi utilizada para disparar uma interrupção na PP, escrevendo no pino de interrupção desta porta. Esta escrita foi realizada pelo próprio tratador T_{eth} de interrupção da placa de rede de E_M. O instante t_1 de escrita no pino de interrupção da PP pelo tratador T_{eth} constitui então o início da seqüência de eventos utilizados para medir as latências de interrupção (Lat_{irq}) e de ativação (Lat_{ativ}).

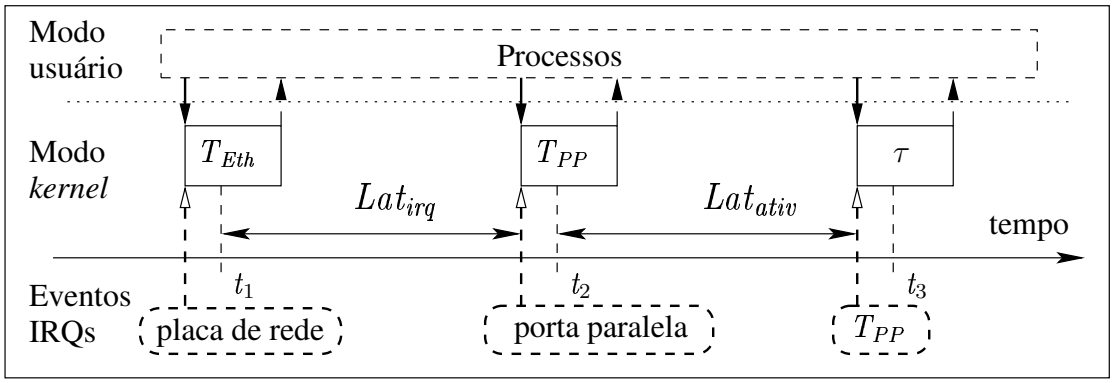


Figura 4.4: Cálculo das latências de interrupção e ativação na estação E_M.

As medidas de latência foram realizadas pela consulta do contador do processador, chamado *Time Stamp Counter* (TSC), permitindo uma precisão da ordem de micro-segundos. Concretamente, um módulo específico (M) foi desenvolvido na estação E_M para exportar as funcionalidades seguintes:

- Ler os 64 bits do TSC e armazenar o valor lido na memória.
- Escrever na porta de entrada e saída 0x378 da porta paralela para gerar interrupções de hardware.
- Criar uma tarefa τ que, num sistema real, executaria algum código útil precisando de

garantias de tempo real crítico. Neste experimento, τ simplesmente grava o valor do TSC.

- Requisitar a captura das interrupções na linha 7 associada à porta paralela e registrar o tratador de interrupção T_{PP} associado.
- Definir o tratador T_{PP} que executa as duas operações seguintes: (1) gravar o TSC; (2) acordar tarefa τ.
- Criar um canal de comunicação FIFO assíncrono para a transferência dos dados temporais coletados em modo protegido para o espaço usuário.

Parte deste conjunto de funções foi disponibilizado sob forma de `ioctl` para os processos executando em modo usuário através de um arquivo especial `/dev/M` do sistema de arquivos. A outra parte foi diretamente exportada, sob a forma de serviços do *kernel* podendo ser utilizados somente por *threads* do *kernel*.

As medidas seguiram o seguinte roteiro, ilustrado pela figura 4.4:

- A estação E_D envia pacotes Ethernet para a estação E_M, provocando interrupções assíncronas em relação às aplicações executando em E_M.
- A interrupção associada à chegada de um pacote provoca a preempção da aplicação executando pelo tratador de interrupção T_{eth}.
- T_{eth} escreve no pino de interrupção da PP e o instante t_1 é armazenado na memória. Este valor t_1 corresponde ao valor lido no relógio local, no instante na escrita da PP, logo após a chegada de uma pacote Ethernet.
- A interrupção associada à escrita no pino de interrupção da PP provoca a preempção da aplicação executando pelo tratador de interrupção T_{PP}.
- T_{PP} grava o instante t_2 e acorda a tarefa τ. Este valor t_2 corresponde ao valor do relógio local, logo após o início de T_{PP}.
- No momento que a tarefa τ acorda, ela grava o instante t_3 e volta a ficar suspensa até a próxima interrupção na PP. O valor de t_3 corresponde ao tempo no qual a tarefa τ começa

a executar, no final da cascata de eventos provocada pela chegada de um pacote na placa de rede.

Assim como representado na figura 4.4, Lat_{irq} corresponde a diferença $t_2 - t_1$ e Lat_{ativ} a diferença $t_3 - t_2$.

No decorrer do experimento, a transferência das medições da memória para o sistema de arquivos é efetuada por um processo usuário (P) (não representado na figura). Antes de iniciar a fase de medidas, P abre o arquivo especial */dev/M* para poder ter acesso às funções *ioctl* disponibilizadas pelo módulo M. P entra então num laço infinito no qual ele executa a chamada de sistema `read` para ler os dados do canal FIFO. Enquanto não há dados, P fica bloqueado. Em algum momento depois da escrita de dados pela tarefa τ, P acorda, lê o canal e escreve os dados num arquivo do disco rígido. Tal procedimento permitiu impedir qualquer interferência entre a aquisição dos dados e seu armazenamento no sistema de arquivos, pois a execução do processo P em modo usuário, não interfere na execução dos módulos do experimento, executados em modo *kernel*.

O mecanismo do disparo das interrupções na porta paralela pelos eventos de chegada de pacotes na placa de rede foi utilizado para garantir que estas interrupções ocorressem de forma assíncrona em relação aos processos executando na estação de medição. É interessante observar que a interrupção na porta paralela sempre acontece logo após a execução de T_{Eth}. Portanto, a medida da latência de interrupção Lat_{irq} deve ser considerada apenas como indicativa. No entanto, ela pôde ser utilizada para o objetivo principal destes experimentos, isto é, comparar as diferentes plataformas estudadas.

Como foi visto na seção 4.3.3, a latência de ativação Lat_{ativ} caracteriza o tempo necessário para ativar uma tarefa após a ocorrência do seu evento de disparo. O dispositivo apresentado aqui permitiu obter resultados de acordo como esperado, como será visto na seção 4.7. Além disso, o efeito do aumento da atividade na estação de medição pôde ser investigado, pois o instante de disparo da interrupção na porta paralela era totalmente independente da atividade dos processos na estação de medição.

4.5.2 Cargas de I/O, processamento e interrupção

Num primeiro momento, realizaram-se experimentos com uma carga mínima no processador da estação de medição (modo *single*). Desta forma, observou-se o comportamento temporal das três plataformas em situação favorável. Em seguida, dois tipos de cargas foram utilizados simultaneamente para sobrecarregar a estação de medição. Tais sobrecargas tiveram por objetivo avaliar a capacidade de cada plataforma em garantir uma latência determinista no tratamento das interrupções e na ativação de tarefas de tempo real, apesar da existência de outras atividades não-críticas. As cargas de I/O e processamento foram realizadas executando as instruções seguintes:

```
while "true"; do
      dd if=/dev/hda2 of=/dev/null bs=1M count=1000
      find / -name "*.c" | xargs egrep include
      tar -cjf /tmp/root.tbz2 /usr/src/linux-xenomai
      cd /usr/src/linux-preempt; make clean; make
done
```

Um outro estresse de interrupção foi criado utilizando uma comunicação UDP entre a estação E_M configurada como servidor e a estação E_C configurada como cliente. Para isolar esta comunicação da comunicação entre E_M e E_D, utilizou-se uma segunda placa de rede em E_M, assim com ilustrado pelo diagrama da figura 4.4. Durante o experimento, o cliente transmitiu pequenos pacotes de 64 bytes na freqüência máxima permitida pela rede, ou seja, com uma freqüência superior a $200\ kHz$ (um pacote a cada $10\mu s$). Desta forma, mais de 100.000 interrupções por segundos foram geradas pela segunda placa de rede de E_M. Esta placa de rede foi registrada na linha de interrupção 18 cuja prioridade é menor que a prioridade da porta paralela. Portanto, as interrupções geradas nesta placa de rede não deveriam, a princípio, interferir nas latências de interrupção mensuradas.

Nos experimentos com cargas, os dois tipos de estresses foram aplicados simultaneamente e as medições só foram iniciadas alguns segundos depois.

4.6 AVALIAÇÃO DE LINUX$^{\mathbf{PRT}}$, LINUX$^{\mathbf{RTAI}}$ E LINUX$^{\mathbf{XEN}}$

4.6.1 Configuração

Os experimentos foram realizados em computadores Pentium 4 com processadores de 2.6 Ghz e 512 Mb de memória, com o objetivo de ilustrar o comportamento temporal das quatro plataformas seguintes:

- **Linux$^{\mathbf{Std}}$:** Linux padrão - *kernel* versão 2.6.23.9 (opção *low-latency*);
- **Linux$^{\mathbf{Prt}}$:** Linux com o *patch* PREEMPT-RT (rt12) - *kernel* versão 2.6.23.9.
- **Linux$^{\mathbf{Rtai}}$:** Linux com o *patch* RTAI - versão "magma" - *kernel* versão 2.6.19.7;
- **Linux$^{\mathbf{Xen}}$:** Linux com o *patch* Xenomai - versão 2.4-rc5 - *kernel* versão 2.6.19.7;

Utilizou-se a configuração Linux$^{\mathbf{Std}}$ para realizar experimentos de referência para efeito de comparação com as duas plataformas de tempo real Linux$^{\mathbf{Prt}}$ e Linux$^{\mathbf{Xen}}$. A versão estável do *kernel* 2.6.23.9, disponibilizada em dezembro de 2007 foi escolhida para o estudo de Linux$^{\mathbf{Prt}}$, pois este *patch* tem evoluído rapidamente desde sua primeira versão publicada há dois anos. No entanto, utilizou-se a versão do *kernel* 2.6.19.7 para o estudo das versões "magma" de RTAI e 2.4-rc5 de Xenomai. De fato, considerou-se desnecessário atualizar a versão do *kernel*, pois RTAI e Xenomai são baseadas no Adeos (ver seção 4.4.3.2) e, portanto, as garantias temporais oferecidas para as aplicações executando no primeiro domínio dependem apenas da versão de RTAI ou Xenomai e do *patch* Adeos associado, e não, da versão do *kernel* Linux.

Utilizou-se uma freqüência de disparo dos eventos pela estação E_D de 20Hz. Para cada plataforma, dois experimentos de 10 minutos foram realizados. O primeiro sem carga nenhuma do sistema e o segundo aplicando os estresses apresentados na seção 4.5.2. Para a plataforma Linux$^{\mathbf{Xen}}$, o experimento com estresses foi repetido por uma duração de 12 horas.

Para as duas plataformas Linux$^{\mathbf{Rtai}}$ e Linux$^{\mathbf{Xeno}}$, os diferentes testes de latências fornecidos foram utilizados, dando resultados menores que $10\mu s$ no pior caso para a latência de interrupção, conforme os padrões da arquitetura Intel Pentium 4. Em ambas as plataformas, desabilitaram-se as interrupções de gerenciamento do sistema (*SMI*), conforme as recomendações dos desenvolvedores (P. MANTEGAZZA et al., 2008; P. GERUM et al., 2008). Isto cancelou

uma latência periódica (a cada 32 segundos) de aproximadamente $2.5ms$ que aparecia nos testes. No caso das plataformas Linux$^{\mathbf{Std}}$ e Linux$^{\mathbf{Prt}}$, estas interrupções foram também desabilitadas, porém, não foi constatada nenhuma alteração das medidas realizadas.

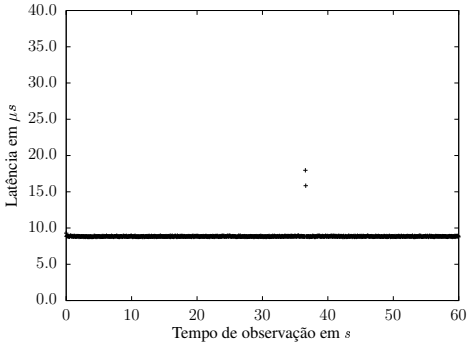
4.6.2 Resultados experimentais

Os resultados experimentais são apresentados nas figuras 4.5 e 4.6, onde o eixo horizontal representa o instante de observação variando de 0 a 60 segundos e o eixo vertical representa as latências medidas em micro-segundos. Apesar de cada experimento ter durado no mínimo uma hora, escolheu-se apresentar apenas resultados para um intervalo de $60s$, pois este intervalo é suficiente para observar o padrão de comportamento de cada plataforma. Neste intervalo, o total de eventos por experimentos é 1200, pois a freqüência de chegada de pacotes utilizada foi de $20Hz$.

Abaixo de cada figura, os seguintes valores são indicados: Valor Médio (VM), desvio padrão (DP), valor mínimo (Min) e valor máximo (Max). Estes valores foram obtidos considerando a duração de uma hora de cada experimento. Na medida do possível, utilizou-se uma mesma escala vertical para todos os gráficos. Conseqüentemente, alguns valores altos podem ter ficado fora das figuras. Tal ocorrência foi representada por um triângulo próximo do valor máximo do eixo vertical.

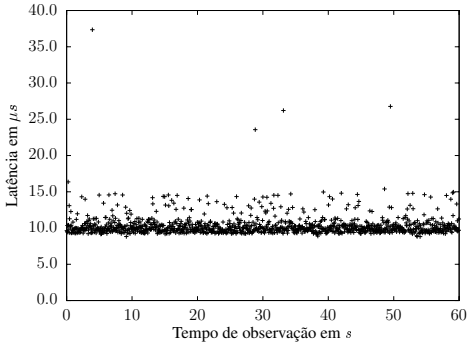
4.6.2.1 Latência de interrupção A figura 4.5 apresenta as latências de interrupção medidas, com e sem estresse do sistema. Como pode ser observado, sem carga, o Linux$^{\mathbf{Std}}$, Linux$^{\mathbf{Rtai}}$ e Linux$^{\mathbf{Xen}}$ têm comportamentos parecidos. Com carga, observa-se uma variação significativa do Linux$^{\mathbf{Std}}$, como esperado.

Com relação ao Linux$^{\mathbf{Prt}}$, dois resultados chamam atenção. Primeiro, o comportamento do sistema sem carga exibe latências da ordem de $20\mu s$. Isto é causado pela implementação dos *threads* de interrupção vista na seção 4.4.1. Segundo, contradizendo as expectativas, a aplicação do estresse teve um impacto significativo, provocando uma alta variabilidade das latências. De fato, entre o instante no qual o tratador T_{PP} acorda o *thread* de interrupção e o instante no qual este *thread* acorda efetivamente, uma ou várias interrupções podem ocorrer. Neste caso, a execução dos tratadores associados pode provocar o atraso da execução de T_{PP}.



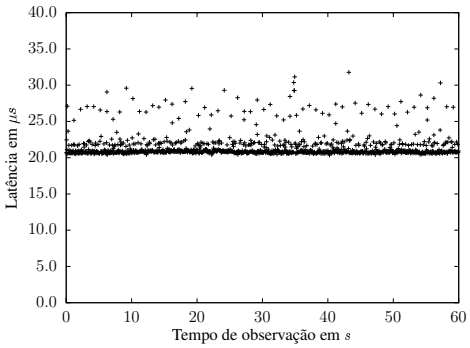
(a) **Linux$^{\mathbf{Std}}$ - Sem carga**

VM: 8.9, DP: 0.3, Min: 8.7, Max: 18.4



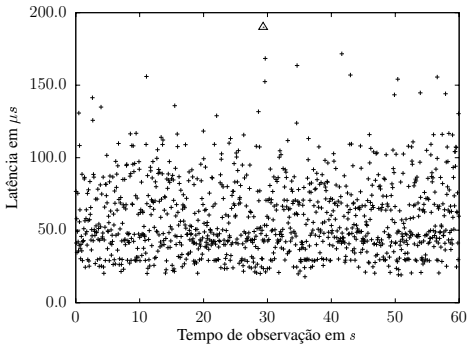
(b) **Linux$^{\mathbf{Std}}$ - Com carga**

VM: 10.4, DP: 1.9, Min: 8.8, Max: 67.7



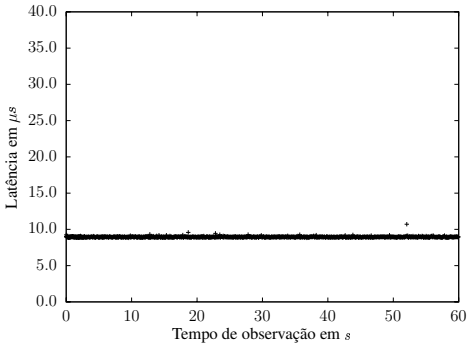
(c) **Linux$^{\mathbf{Prt}}$ - Sem carga**

VM: 21.5, DP: 1.7, Min: 20.3, Max: 45.1



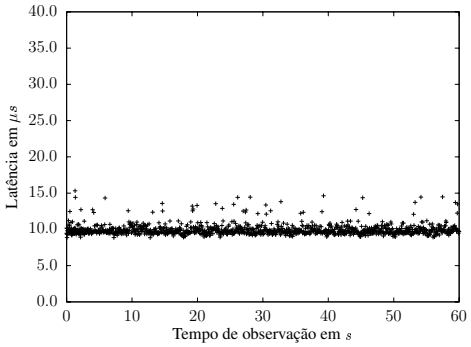
(d) **Linux$^{\mathbf{Prt}}$ - Com carga**

VM: 58.5, DP: 26.4, Min: 17.2, Max: 245.9



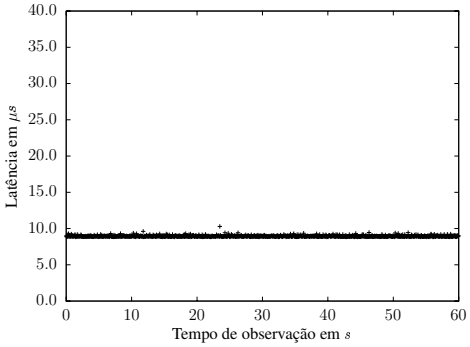
(e) **Linux$^{\mathbf{Rtai}}$ - Sem carga**

VM: 8.7, DP: 0.1, Min: 8.8, Max: 10.7



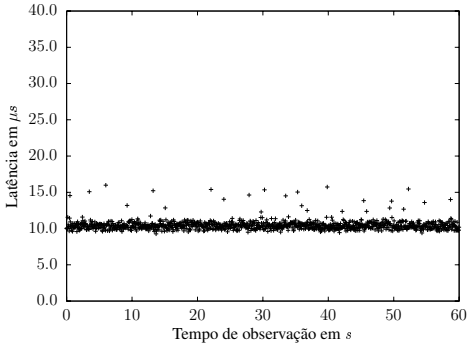
(f) **Linux$^{\mathbf{Rtai}}$ - Com carga**

VM: 10.0, DP: 0.8, Min: 8.8, Max: 20.8



(g) **Linux$^{\mathbf{Xen}}$ - Sem carga**

VM: 9.0, DP: 0.1, Min: 8.8, Max: 11.1



(h) **Linux$^{\mathbf{Xen}}$ - Com carga**

VM: 10.4, DP: 0.7, Min: 8.9, Max: 20.8

Figura 4.5: Latência de interrupção com freqüência de escrita na PP de $20 Hz$.

Para cancelar esta variabilidade indesejável, é possível usar o Linux$^{\mathbf{Prt}}$ sem utilizar a implementação de *threads* de interrupção. Para tanto, usa-se a opção `IRQF_NODELAY` na requisição da linha de interrupção. Utilizando esta opção na requisição da linha de interrupção da porta paralela, o comportamento do Linux$^{\mathbf{Prt}}$ passa a ser semelhante ao Linux$^{\mathbf{Std}}$.

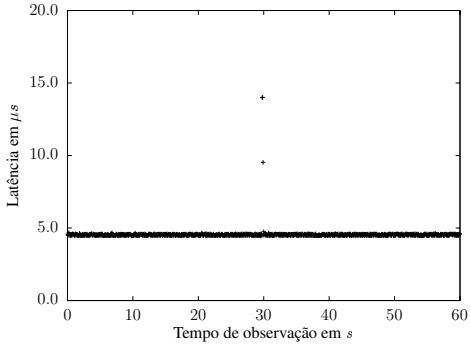
4.6.2.2 Latência de ativação A figura 4.6 apresenta os resultados para as latências de ativação sem estresse e com estresse do processador. Como pode ser observado, o comportamento de Linux$^{\mathbf{Std}}$ é inadequado para atender os requisitos de tempo real. Linux$^{\mathbf{Prt}}$, Linux$^{\mathbf{Rtai}}$ e Linux$^{\mathbf{Xen}}$, por outro lado, apresentam valores de latências dentro dos padrões esperados. Vale a pena notar o comportamento destes sistemas com carga. Constata-se que o valor médio encontrado para Linux$^{\mathbf{Xen}}$ $(8,7\mu s)$ é superior ao do Linux$^{\mathbf{Prt}}$ $(3,8\mu s)$ e ao Linux$^{\mathbf{Rtai}}$ $(4,4\mu s)$. No entanto, o desvio padrão de Linux$^{\mathbf{Xen}}$ é significativamente menor que este de Linux$^{\mathbf{Prt}}$, característica desejável para sistemas de tempo real críticos. Por outro lado, acredita-se que o fato de Linux$^{\mathbf{Xen}}$ ter um valor médio superior ao Linux$^{\mathbf{Rtai}}$ é devido à sobrecarga introduzida pelo projeto Xenomai para poder oferecer uma interface disponível em modo usuário.

É interessante ainda observar o comportamento de Linux$^{\mathbf{Prt}}$ sem utilizar o contexto de *threads* de interrupção, isto é, com a opção `IRQF_NODELAY`, comentada anteriormente. Como pode ser observado na figura 4.7, apesar de as latências de ativação sem estresse apresentar uns bons resultados em comparação ao Linux$^{\mathbf{Prt}}$, seus valores com estresse indicam um comportamento menos previsível que o Linux$^{\mathbf{Xen}}$.

4.7 CONCLUSÃO

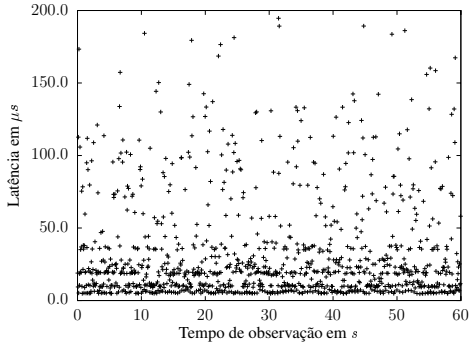
Neste capítulo, os conceitos de sistemas operacionais de propósito geral e de sistemas operacional de tempo real foram apresentados, assim como as principais abstrações que são necessárias para a descrição das suas funcionalidades. Em seguida, as principais causas de imprevisibilidade de um SOPG tal como Linux foram identificadas.

Para descrever os desafios que devem ser resolvidos para tornar um SOPG mais previsível, três soluções baseadas em Linux foram descritas em detalhes. Experimentos foram realizados com o objetivo de comparar as três soluções de SOTR baseadas em Linux consideradas para a implementação de *DoRiS*.



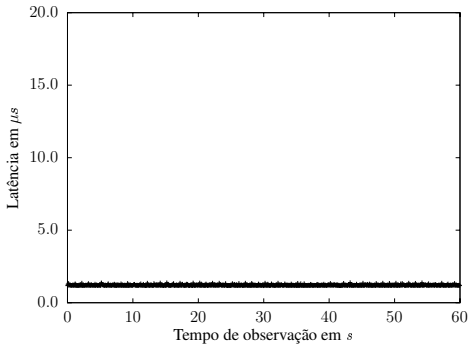
(a) **Linux$^{\mathbf{Std}}$ - Sem carga**

VM: 4.6, DP: 0.4, Min: 4.4, Max: 16.2



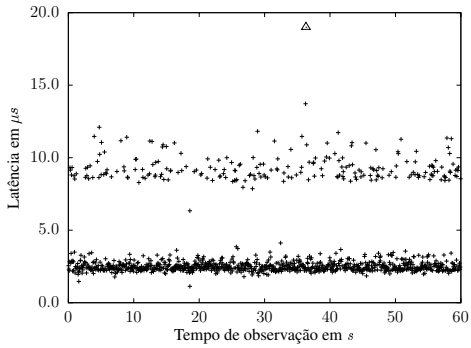
(b) **Linux$^{\mathbf{Std}}$ - Com carga**

VM: 37.3, DP: 48.2, Min: 4.6, Max: 617.5



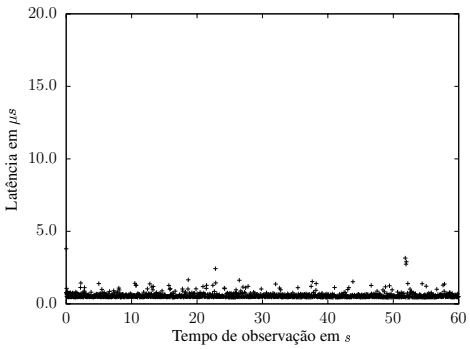
(c) **Linux$^{\mathbf{Prt}}$ - Sem carga**

VM: 2.1, DP: 0.2, Min: 1.2, Max: 9.4



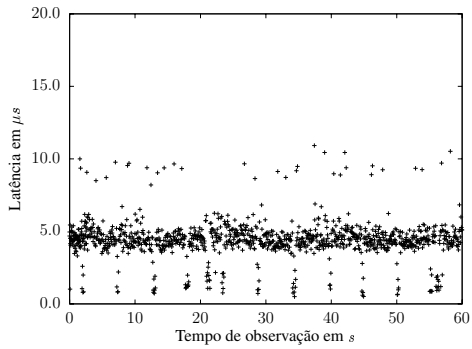
(d) **Linux$^{\mathbf{Prt}}$ - Com carga**

VM: 3.8, DP: 2.8, Min: 1.1, Max: 27.4



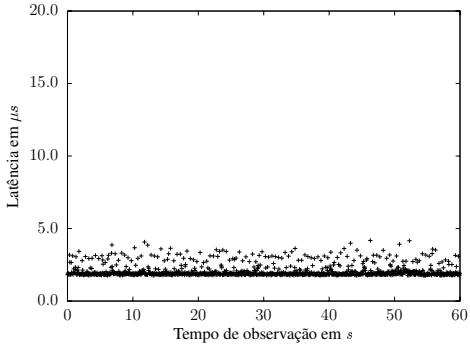
(e) **Linux$^{\mathbf{Rtai}}$ - Sem carga**

VM: 0.6, DP: 0.3, Min: 0.4, Max: 3.8



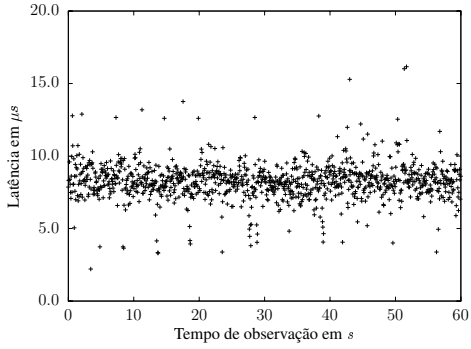
(f) **Linux$^{\mathbf{Rtai}}$ - Com carga**

VM: 4.4, DP: 0.7, Min: 0.5, Max: 14.7



(g) **Linux$^{\mathbf{Xen}}$ - Sem carga**

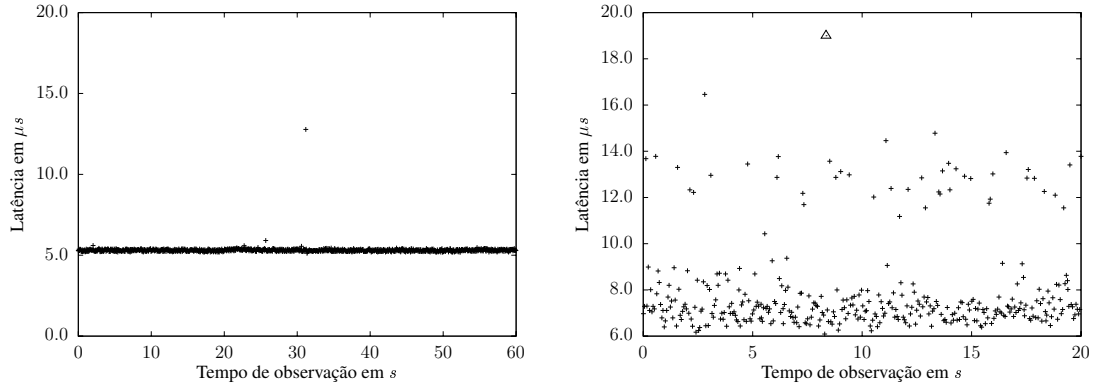
VM: 2.1, DP: 0.5, Min: 1.8, Max: 8.4



(h) **Linux$^{\mathbf{Xen}}$ - Com carga**

VM: 8.7, DP: 0.3, Min: 1.8, Max: 18.7

Figura 4.6: Latência de ativação com freqüência de escrita na PP de $20Hz$.



(a) **Linux$^{\mathbf{Prt}}$ - Sem carga**
VM: 5.3, DP: 0.3, Min: 5.0, Max: 13.1

(b) **Linux$^{\mathbf{Prt}}$ - Com carga**
VM: 8.0, DP: 2.0, Min: 5.2, Max: 31.0

Figura 4.7: Latência de ativação do Linux$^{\mathbf{Prt}}$ desabilitando o *thread* associado as interrupções da PP (opção `IRQF_NODELAY`).

A metodologia experimental permitiu medir as latências de interrupção e de ativação, em situações de carga variável, tanto do processador quanto de eventos externos tratados por interrupção. Linux padrão apresentou latências no pior caso acima de $100\mu s$, enquanto as plataformas Linux$^{\mathbf{Prt}}$, Linux$^{\mathbf{Rtai}}$ e Linux$^{\mathbf{Xen}}$ conseguiram prover garantias temporais com uma precisão abaixo de $20\mu s$. No entanto, para se conseguir este comportamento em relação ao Linux$^{\mathbf{Prt}}$, foi necessário desabilitar *threads* de interrupção, tornando o sistema menos flexível. Com tais *threads*, o comportamento de Linux$^{\mathbf{Prt}}$ sofreu considerável degradação da sua previsibilidade temporal.

Observa-se que os experimentos foram realizados numa arquitetura específica e que resultados quantitativamente diferentes teriam provavelmente sido encontradas em outras configurações de máquinas. No entanto, os resultados obtidos aqui são qualitativamente consistentes com os resultados apresentados em trabalhos similares (ROSTEDT; HART, 2007; SIRO; EMDE; MCGUIRE, 2007; BENOIT; YAGHMOUR, 2005; DOZIO; MANTEGAZZA, 2003).

Mais especificamente, o presente trabalho apresentou resultados de latência de interrupção que confirmam os resultados obtidos em (BENOIT; YAGHMOUR, 2005), numa avaliação bastante abrangente do *patch* Adeos, publicada apenas na Internet. Já os resultados encontrados aqui para Linux$^{\mathbf{Prt}}$, sem a opção `IRQF_NODELAY`, diferiram dos apresentados por (BENOIT; YAGHMOUR, 2005), pois uma degradação das garantias temporais por esta plataforma foi observada, tal como visto na seção 4.6.2.1. Em relação às latências de ativação, não temos conhecimento de nenhum outro trabalho comparativo.

A proposta do projeto Xenomai se destacou pelo fato de oferecer um ambiente de programação em modo usuário e ao mesmo tempo conseguir tempos de latência típicos das soluções baseadas em *nanokernel*.

CAPÍTULO 5

AMBIENTE DE IMPLEMENTAÇÃO E CONFIGURAÇÃO DE *DORIS*

5.1 INTRODUÇÃO

Neste capítulo, a implementação do protocolo de código aberto *DoRiS* é apresentada. A plataforma de tempo real escolhida foi 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. Como foi visto no capítulo 4, Xenomai oferece garantias temporais da ordem de dezenas de micro-segundos (nos equipamentos utilizados no nosso laboratório), suficiente para a implementação de *DoRiS*. Além destas garantias temporais, a escolha do Xenomai apresenta duas vantagens significativas. Em primeiro lugar, Xenomai oferece uma interface de programação chamada RTDM (*Real Time Driver Model*) cujo 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 as suas implementações, possibilitando a concentração de esforços 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 5.2. Em seguida, a camada de rede RTnet e sua interface RTDM com a plataforma Xenomai são descritos na seção 5.3. A seção 5.4 é dedicada à apresentação da implementação de *DoRiS* e da sua integração na camada RTnet do Xenomai.

No final desta mesma seção, alguns resultados são apresentados. Por fim, algumas conclusões são discutidas na seção 5.5.

5.2 AS CAMADAS DE REDE E ENLACE DO LINUX

Esta seção apresenta apenas alguns elementos da implementação, pelo *kernel* Linux, dos protocolos IP na camada de rede e Ethernet 802.3 na camada de enlace, pois somente estas duas camadas são utilizadas pela implementação de *DoRiS*. Estas duas camadas utiliza principalmente duas estruturas para armazenar os dados necessários à transmissão e recepção de pacotes. A primeira, chamada `net_device`, é associada 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 acesso à memória circular definida na zona DMA (*Direct Memory Access*) utilizadas pelo controlador da placa de rede para armazenar os pacotes sendo transmitidos e recebidos. 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 que são necessárias ao 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 local da placa de rede, o dispositivo copia o pacote na memória DMA prevista para este efeito. Para fins de otimização, o dispositivo pode ser configurado para operar com vários pacotes, ao invés de apenas um. Tal procedimento não muda o modo de operação, pois tratar vários pacotes de uma vez é equivalente a tratar um pacote de tamanho maior. Portanto, ilustra-se-á aqui o processo de recepção de apenas um pacote. Logo que o pacote se torna disponível na memória DMA, o *kernel* precisa ser informado da sua presença 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 o controlador da placa de rede termina a operação de DMA, o dispositivo da placa de rede interrompe o processador

para informá-lo da presença do pacote a ser recebido. No tratador desta interrupção, o *kernel* armazena as informações relevantes para localizar o pacote e cria um `softirq` (ver seção 4.2.2) 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 seguintes operações: (i) alocação dinâmica do espaço de memória de um `sk_buff`; (ii) cópia do pacote armazenado na memória DMA neste `sk_buff`; e (iii) encaminhamento do `sk_buff` (via apontadores) para a camada IP. Esta abordagem tem a seguinte limitação. Quando a taxa de chegadas de pacotes alcança um certo patamar, a chegada de um novo pacote acontece antes que o tratador do pacote anterior termine de executar. À 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, então, monopolizado sem que haja possibilidade alguma de executar os `softirqs` escalonados ou qualquer outro processo. Em algum momento, a memória DMA fica cheia e novos pacotes 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 há algum pacote em espera para ser recebido. Se este for o caso, o *kernel* processa parte ou todos dos pacotes que estiverem 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 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 chega 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 as vantagens de ambos os 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, referentes a apenas um dispositivo específico. Na chegada de um pacote, a seguinte seqüência de eventos é executada:

i) o dispositivo copia o pacote na memória circular DMA. Na ausência de espaço nesta memória, o pacote pode ser tanto descartado quanto copiado no lugar do mais velho pacote já presente na memória local.;

ii) se as interrupções da placa de rede forem habilitadas, o dispositivo interrompe o processador para informá-lo que há pacote em espera na memória DMA. Senão, o dispositivo continua a receber pacote e transferi-los para a memória DMA, sem interromper o processador;

iii) na ocorrência de uma interrupção da placa de rede, o tratador começa por desabilitar as interrupções provenientes 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 na memória DMA. Se o número de pacotes para serem processados é maior que o limite configurado, o `softirq` agenda-se para processá-los posteriormente. Em seguida, o *kernel* executa outros `softirqs` que estão eventualmente em espera, antes de escalonar o `softirq` de consulta da placa de rede novamente;

v) quando a memória DMA não contém 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 constatado, esta solução impede o cenário de *livelock* graças à 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 desnecessária do processador.

Do ponto da visto das latências, esta solução tem os seguintes defeitos. 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`) para armazenar os pacotes. Esta alocação pode também levar um tempo imprevisível, pois a chamada de sistema `malloc` pode falhar, na ausência de memória disponível.

As seções a seguir mostram como estes problemas são resolvidos pela camada RTnet da plataforma Xenomai.

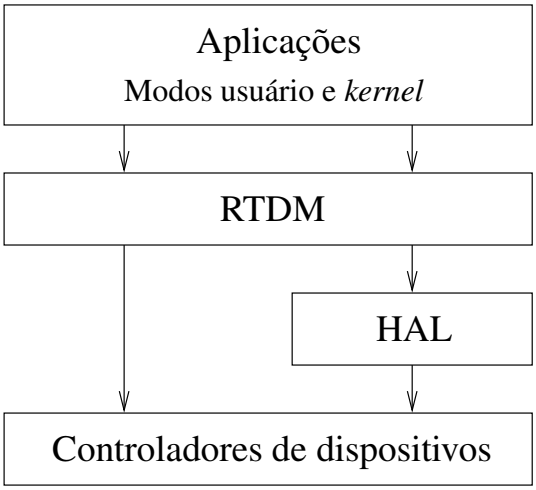


Figura 5.1: A interface do RTDM (reproduzida de (KISZKA, 2005))

5.3 A CAMADA DE REDE DO XENOMAI: RTNET

5.3.1 RTDM

Antes de apresentar o projeto RTnet em detalhes, precisa-se descrever brevemente a interface de programação na qual ele se baseia, isto é a interface RTDM (*Real Time Driver Model*) (KISZKA, 2005), que vem sendo desenvolvida 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 constitui 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 5.1. O RTDM foi inicialmente especificado e desenvolvido na plataforma Xenomai. No entanto, em vista dos benefícios trazidos por esta API, ela também foi adotada pelo RTAI.

A interface do RTDM oferecida para as aplicações pode ser dividida em dois conjuntos de funções. O primeiro conjunto é constituído das funções que dão suporte aos dispositivos de entrada e saída e aos dispositivos associados à serviços. O segundo conjunto é constituído das funções associadas aos serviços de tempo real básicos, independentes do hardware.

- ***Serviços de entrada e saída e troca de mensagens*** – Para este conjunto de funções, 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`. Os dispositivos associados a protocolos, dedicados à troca de mensagens, 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 usuais do *kernel*, o sufixo `_rt` ou `_nrt` é adicionado ao seu nome. Do ponto de vista 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`. Caso contrário, a função `sendmsg_nrt` será executada.

• ***Serviços do* nanokernel** – Este segundo conjunto de funções diz respeito à abstração dos serviços do *nanokernel* de tempo real. Elas contemplam notadamente os serviços de relógios de alta precisão e dos 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 principais, vários outros utilitários são disponibilizados, tais como, 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 ao projeto Xenomai (P. GERUM et al., 2008) realizada em dezembro de 2007, enumerou-se um pouco mais de 100 funções definidas por esta API. Desta forma, o uso do RTDM para a implementação de *DoRiS* apareceu como uma escolha interessante. Além disso, o RTDM também é disponível na plataforma RTAI, permitindo o uso dos produtos de *software* baseados nesta interface, tanto na plataforma Xenomai quanto no RTAI.

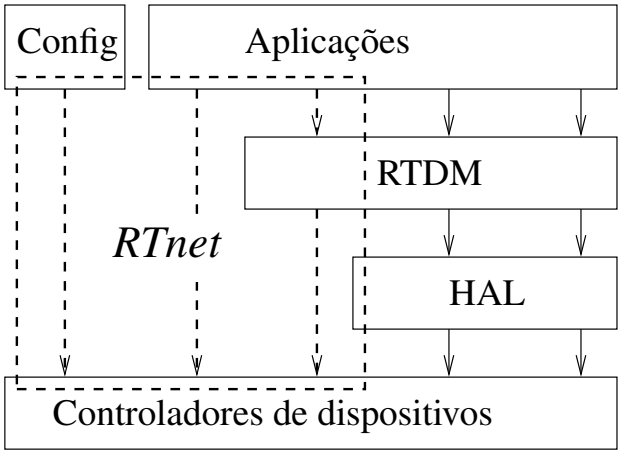


Figura 5.2: Localização do RTnet.

5.3.2 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. 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 5.2. Como pode ser observado, RTnet utiliza os dispositivos de *hardware* existentes e introduz uma camada de software para aumentar o determinismo dos serviços de comunicação. Do ponto de vista da sua interface com as aplicações, RTnet utiliza o RTDM (*Real Time Driver Model*) (KISZKA, 2005). Em relação ao hardware, RTnet utiliza tanto a camada HAL, provida pelo Xenomai, quanto controladores de dispositivos próprios. Para a implementação de tais controladores, capazes de prover garantias temporais, o código original dos controladores de dispositivos do Linux é modificado, conforme a descrição disponível na documentação do RTnet (J. KISZKA et al., 2008). O objetivo principal destas modificações é remover do código do dispositivo qualquer chamada às funções bloqueantes do *kernel* Linux.

O detalhe dos componentes da pilha RTnet é apresentado na figura 5.3. Como pode ser observado, RTnet se inspira na organização em camada da pilha de rede do Linux. No entanto, a implementação atual do RTnet só oferece serviços de comunicação baseados em UDP/IP e não fornece suporte ao protocolo TCP/IP.

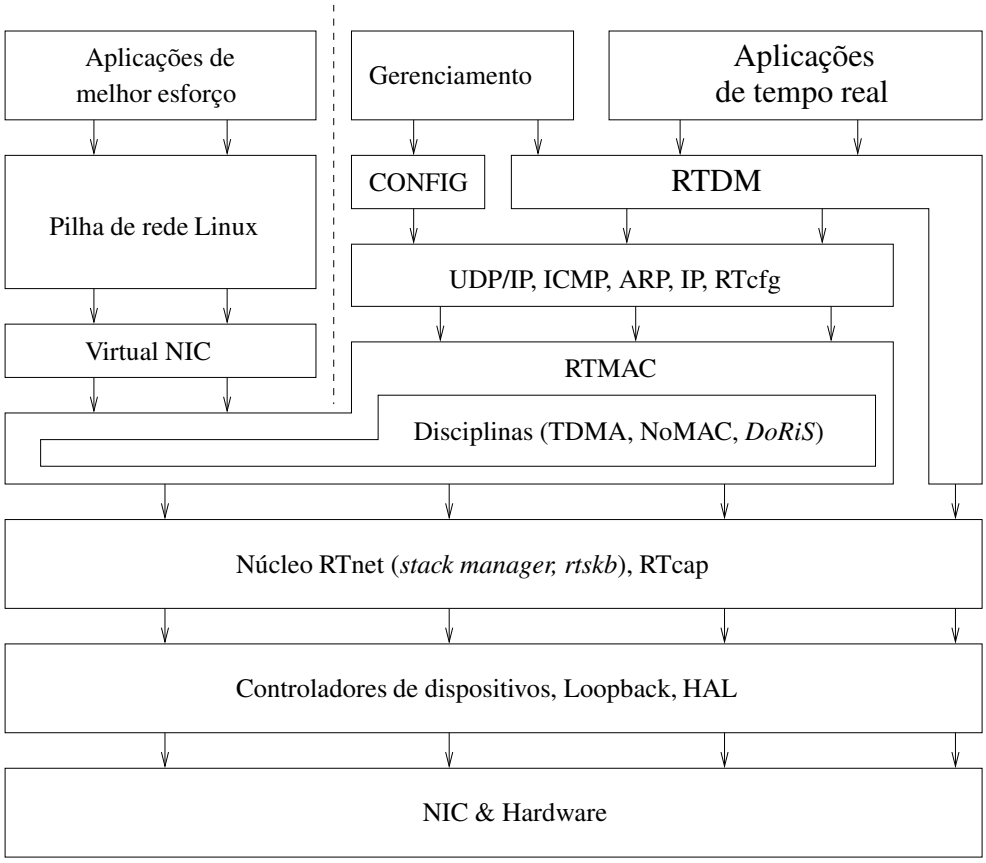


Figura 5.3: Estrutura em camada do RTnet.

Acompanhando a figura 5.3 de cima para baixo, pode-se ver 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. Encaixados na interface abaixo do RTDM, os componentes da camada de rede provêem implementações específicas dos protocolos UDP, ICMP, e ARP. A seção 5.3.3.3 mostra algumas das soluções adotadas para aumentar o determinismo dos protocolos UDP e ARP. O protocolo ICMP, baseado no protocolo IP, é dedicado às funções de controle e gerenciamento da rede. A sua implementação é bastante específica e de pouca relevância para este trabalho, e portanto, não será apresentada aqui.

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 ilustrados na figura 5.3. O núcleo contém os serviços de emissão e recepção dos paco-

tes, baseados na modificação dos controladores de dispositivos. Detalhes desta implementação serão descritos na seção 5.3.3. 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 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 como a interface RTmac, na seção 5.3.4. 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 5.4.

Além destes componentes principais, 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 de tempo real em tempo de execução. Os segundos permitem coletar dados temporais sobre os pacotes transmitidos. Estes serviços são opcionais, e não serão descritos aqui.

Ao lado da pilha RTnet, aplicações que utilizam os serviços de melhor esforço da pilha de rede do Linux podem fazê-lo através de interfaces virtuais, cujo mecanismo será apresentado na seção 5.3.3, juntamente com a descrição do formato dos pacotes RTnet.

5.3.3 RTnet: principais componentes

5.3.3.1 Gerenciamento de memória Foi visto na seção 5.2 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 pelo gerenciamento da memória virtual para que ele libere algumas páginas não utilizadas 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, relativos aos processos de emissão ou recepção, deve criar uma reserva de estruturas `rtskb`. Tal estrutura, similar à estrutura `sk_buff` do Linux padrão, é utilizada

para armazenar as informações e os dados de um pacote, desde sua chegada na memória DMA de recepção, até sua entrega à aplicação destino. Cada `rtskb` tem um tamanho fixo, suficiente para armazenar um pacote de tamanho máximo. Como o tamanho de cada reserva de estruturas `rtskb` é fixo, um mecanismo de troca é utilizado. Em outras palavras, 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 referências, tais operações de troca são quase instantâneas em comparação ao tempo que levaria a cópia do conteúdo destas estruturas.

5.3.3.2 Emissão e Recepção de pacotes A emissão de um pacote é uma operação que acontece no contexto da execução seqüencial de um processo. Portanto, é um evento síncrono e as operações subseqüentes são realizadas com a prioridade da tarefa que executa a operação de emissão. Conseqüentemente, as garantias temporais associadas a uma emissão dependem exclusivamente da plataforma operacional e da utilização correta dos seus serviços.

No caso da operação de recepção de uma mensagem, a situação é diferente, pois esta operação 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 chamada "gerente da pilha" (*stack manager*) tem a maior prioridade no sistema. Desta forma, garante-se que se o pacote for destinado a uma tarefa crítica, ele será encaminhado o mais rapidamente possível. O início do processo de recepção é parecido com este do Linux (ver seção 5.2). Após ter copiado um pacote na memória 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 começa a executar imediatamente, pois tem a maior prioridade do sistema, determina se o pacote é de tempo real ou não. Se for de tempo real, ele efetua as operações de recepção necessárias até entregar o pacote para a aplicação de destino. Caso contrário, o "gerente" coloca o pacote na fila dos pacotes de melhor esforço para serem recebidos, acorda o processo de baixa prioridade dedicado ao processamento desta fila e retorna.

5.3.3.3 A camada de rede Os principais problemas de latência para serem resolvidos na camada de rede são devido ao uso do protocolo ARP (*Address Resolution Protocol*) e aos mecanismos de fragmentação de pacote do protocolo UDP/IP.

Em relação à fragmentação de pacote, o RTnet oferece uma opção de configuração que permite transmitir pacotes 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 descarte de cadeias incompletas, antes que o conjunto de coletores se esgote. De fato, a perda de um pacote impede a compleição da cadeia associada e impossibilita a liberação da memória utilizada por esta cadeia.

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 raramente acontece.

Em relação ao protocolo ARP, os protocolos de redes convencionais o utilizam dinamicamente para construir a tabela ARP que associa os números IP e os endereços de roteamento Ethernet (KUROSE; ROSS, 2005). Para determinar o endereço Ethernet correspondente 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 estiver no mesmo segmento Ethernet, ela manda uma resposta informando o seu endereço Ethernet. Caso contrário, o roteador encarregado da sub-rede associada àquele IP informa do seu endereço Ethernet. Quando a resposta chega, a tabela ARP da estação de origem é atualizada. 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, realizada em tempo de configuração.

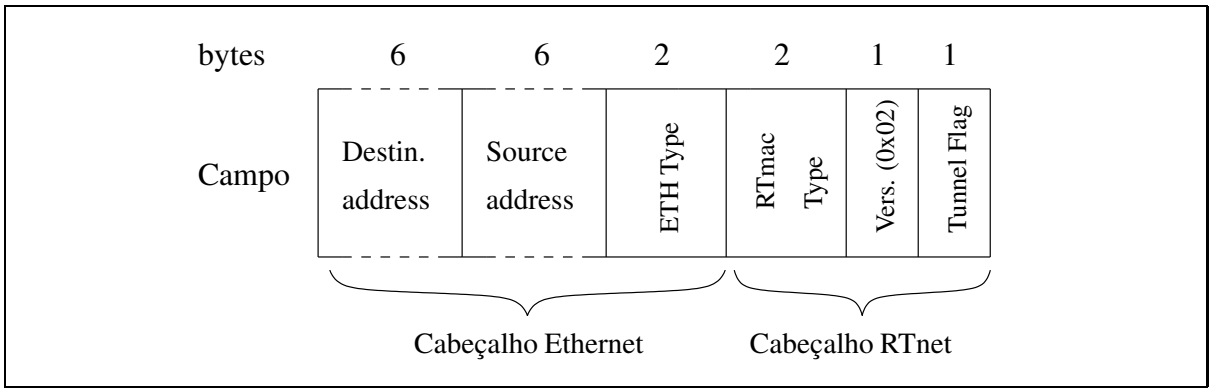


Figura 5.4: Cabeçalhos do RTnet.

5.3.3.4 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 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 à 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 necessária para que a comunicação RTnet se estabeleça. 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 5.4. Estes campos são utilizados quando o valor do campo *type* do cabeçalho Ethernet (`0x9021`) indica que o pacote é destinado à camada RTmac. Isto acontece, por exemplo, quando se trata de um pacote de configuração, ou 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` no momento do encapsulamento. Desta forma, a camada RTmac consegue determinar qual é a aplicação para a 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 virtual, cha-

mada VNIC (*Virtual NIC*). Esta interface é configurada com o comando `ifconfig` usual do Linux. 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, formam o esqueleto 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.

5.3.4 RTnet: As disciplinas TDMA e NoMAC

Apesar de ser opcional, o uso de alguma disciplina de acesso ao meio pode ser necessário para prover determinismo, em particular, quando o meio tem uma política de acesso probabilística, como é o caso de Ethernet (ver seção 2.1.2). Fiel ao seu modelo modular e hierárquico de desenvolvimento, RTnet fornece a interface RTmac para prover uma disciplina 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 falhas. 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 provenientes das aplicações executadas no *kernel* Linux via interface VNIC .

A disciplina NoMAC, como seu nome indica, não é 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 esqueleto de implementação é disponibilizado para facilitar o desenvolvimento de novas disciplinas de acesso ao meio.

5.4 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 de acordo com a interface RTmac. Para isto, adotou-se a seguinte metodologia. Usou-se como base de desenvolvimento a disciplina NoMAC e aproveitaram-se os exemplos de implementação mais elaborados encontrados no código da disciplina TDMA. De maneira geral, concentraram-se todas as novas funcionalidades necessárias na disciplina *DoRiS*. Desta forma, *DoRiS* pôde seguir as regras de instalação do RTnet.

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

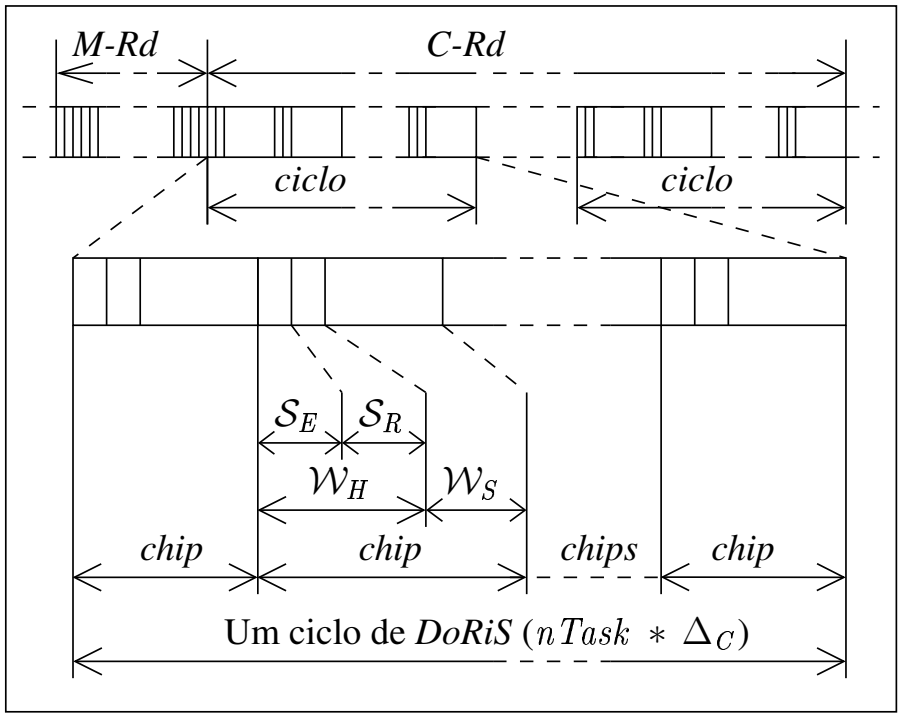


Figura 5.5: O esquema de divisão temporal de *DoRiS*

i) a composição do grupo de estações que participam do segmento *DoRiS* é configurada em tempo de projeto.;

ii) cada estação do segmento hospeda um único servidor *DoRiS*. Portanto, o número de estações do segmento é igual ao número de servidores, denotado $nServ$, valor conhecido por todos os servidores;

iii) para fins de simplificação, os identificadores dos servidores são regularmente alocados, indo de 1 a $nServ$.

A primeira restrição poderia ser relaxada com o uso de um protocolo independente de configuração dinâmica em tempo de execução na fase $M - Rd$ da figura 3.1. Esta figura, apresentada no capítulo 3, 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 trabalho, escolheu-se uma implementação sem o mecanismo de reserva. No entanto, as estruturas necessárias foram previstas e a implementação deste mecanismo deverá ocorrer num futuro próximo.

Vale também observar que a constante $nTask$, utilizada na especificação de *DoRiS*, apresentada no capítulo 3, corresponde ao número de servidores $nServ$. Isto é devido ao fato que a especificação inicial foi escrita considerando que uma tarefa correspondia a um servidor.

Além da própria disciplina *DoRiS*, cujos detalhes da implementação serão descritos na seção 5.4.5, os mecanismos de configuração e sincronização utilizados serão apresentados nas seções 5.4.2, 5.4.3 e 5.4.4. Antes de descrever estes mecanismos, a implementação do modelo de comunicação um-para-todos será brevemente descrita na seção 5.4.1

5.4.1 Comunicação um-para-todos

Uma das diferenças entre o protocolo *DoRiS* e o protocolo TDMA utilizado pelo RTnet é o modelo de comunicação. Ao invés de usar o modo de comunicação ponto-a-ponto dos *sockets*, *DoRiS* utiliza, para a implementação do anel crítico, 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.

No entanto, a interface do RTDM utilizada para abrir canais de comunicações segue o padrão de comunicação ponto-a-ponto dos *sockets*. Portanto, precisou-se identificar as mensagens com o identificador do servidor emissor. Para tal efeito, considerou-se aqui que um *byte* era suficiente, 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 de cada estação.

Mais explicitamente, o número IP (*Internet Protocol*) de uma estação E é configurado em tempo de projeto de tal forma que o seu último *byte* coincide com o identificador único do servidor *DoRiS* de E. Esta implementação permitiu aproveitar os códigos baseados em IP do RTnet, deixando a possibilidade de se 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.

5.4.2 Configuração e sincronização do anel crítico

Na fase inicial, quando um servidor quer se inserir num segmento de comunicação *DoRiS*, ele começa por observar a comunicação já existente durante três ciclos. Vale lembrar que um ciclo dura exatamente $nServ * \Delta_C$. Depois destes três ciclos de comunicação, se o servidor não percebe nenhuma mensagem, ele deduz que ninguém está enviando mensagens ainda e escolhe qualquer instante para transmitir uma mensagem elementar. Para evitar que duas estações decidam simultaneamente enviar suas primeiras mensagens elementares, provocando eventualmente uma colisão, espera-se um tempo para dar início a uma nova estação depois da primeira. Desta forma, a segunda estação tem a possibilidade de observar as mensagens da primeira e pode assim se sincronizar.

Após o início de um servidor S_i de identificador i, um outro servidor S_j que queira participar da comunicação deve observar três mensagens elementares enviadas por S_i durante três ciclos consecutivos de observação. S_j pode então deduzir o seu instante de transmissão t_j, utilizando o identificador carregado pela mensagem elementar enviada por S_i e o instante t da chegada desta mensagem:

$$t_j = t + ((nTask + j - i) \,\%\, nTask) * \Delta_C \; - \Delta_E \tag{5.1}$$

Nesta equação, a quantidade $(nTask + j - i) \,\%\, nTask$ representa o número de *chip* entre o *chip* corrente e o *chip* no qual S_j deve emitir. A subtração do termo Δ_E é devido ao fato de t ser o instante de recepção da mensagem. Portanto, o tempo Δ_E do *slot* elementar precisa ser subtraído.

A única informação desconhecida na formula (5.1) é justamente esta duração Δ_E de um *slot* elementar. Esta duração é a resultante de várias latências de diferentes naturezas:

i) a latência na estação emissora, que, por sua vez, pode ser decomposta em três termos: 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 devido à transmissão e à propagação da mensagem no meio físico, cujos valores dependem da taxa da banda e do comprimento do cabo conectando os nós;

iii) a latência na estação recebedora, que pode, também, ser decomposta em dois termos: a latência de recepção correspondente ao tempo necessário para copiar o pacote na memória DMA de recepção e a latência de interrupção, já amplamente discutida no capítulo 4.

Observa-se que estas diferentes fontes de latências têm uma variabilidade associada, pelo menos no que diz respeito aos itens (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 específica foi desenvolvida para estimar o valor total de Δ_E usando medidas realizadas durante a execução do protocolo.

5.4.3 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, o instante de recepção desta mensagem por todas as outras estações é o mesmo. Ou seja, considera-se que as latências de propagações e de recepções de uma mensagem são as mesmas para todas as estações. Esta hipótese pode ser resumida pelas duas suposições seguintes:

i) As diferenças dos tempos de propagação entre quaisquer duas estações são desprezíveis.

ii) As diferenças entre as latências nas estações recebedoras são desprezíveis.

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 à primeira suposição, sabe-se que a velocidade de propagação das mensagens na rede é um pouco inferior à velocidade da luz no vácuo. Pode-se assumir, para fins de estimativa, o valor de $2.510^8 m/s$, o qual é próximo da velocidade da luz num meio material. 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 4.5).

A segunda suposição estabelece que, em todas as estações, os tempos entre a chegada do pacote na placa de rede e o tratamento da interrupção decorrente do processador sejam iguais. Esta hipótese pode ser garantida usando dispositivos de *harware* com o mesmo comportamento temporal e uma plataforma operacional determinista. No caso de existirem diferenças significativas entre os dispositivos de *harware* do segmento, o segundo ponto pode ser relaxado, por mais que o comportamento dos dispositivos seja determinista. Neste caso, fatores corretivos devem ser estimados em tempo de projeto para compensar o efeito da disparidade dos dispositivos, e permitir assim a integração de nós lentos e rápidos num mesmo segmento *DoRiS*.

No dispositivo experimental utilizado neste trabalho, fatores corretivos não foram necessários, pois os três participantes do segmento tinham o mesmo dispositivo de *hardware* (placa de rede RealTek 8139) e utilizavam a mesma plataforma operacional.

Portanto, as suposições (i) e (ii) foram consideradas válidas e as propriedades da comunicação um-para-todos foram utilizadas para calcular o valor da latência Δ_E. Para ilustrar este procedimento, consideram-se três estações A, B e C e o cenário de trocas de mensagens representadas na figura 5.6.

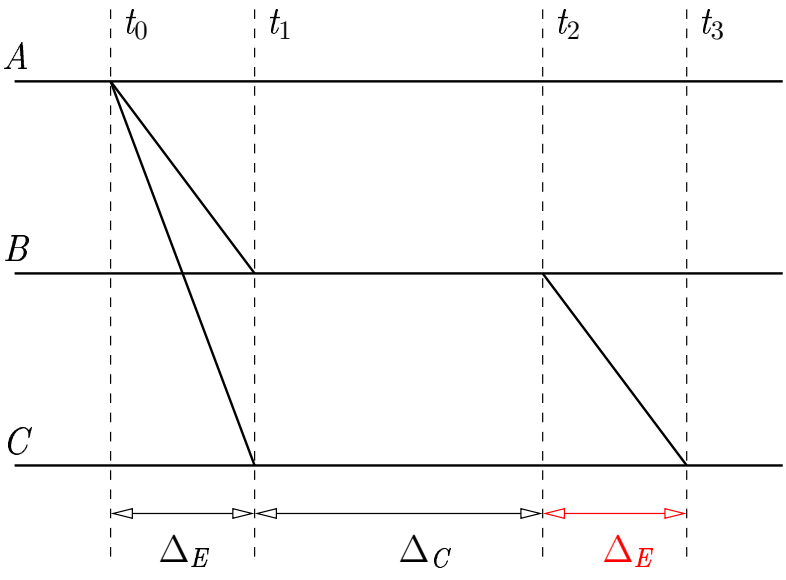


Figura 5.6: Cálculo de Δ_E

No instante t_0, a estação A envia uma mensagem elementar (de tamanho 64 bytes) recebida por B e C num mesmo instante t_1. B espera então um tempo Δ_C, conhecido por C, para

enviar a sua mensagem elementar. Devido ao determinismo da plataforma operacional, os desvios devido às latências internas são menores que $10\mu s$, valor muito menor que Δ que é superior a $500\mu s$. Portanto, este desvios não são representados aqui. 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:

$$\Delta_E = t_3 - t_1 - \Delta_C$$

Este procedimento foi utilizado por todas as estações de tal forma que, a cada recepção de duas mensagens elementares consecutivas, cada estação pôde calcular o valor de Δ_E . O valor médio deduzido destas observações pôde ser então aproveitado para configurar *DoRiS* em tempo de projeto.

5.4.4 Configuração do anel não-crítico

No decorrer da implementação do anel não-crítico, tal como especificado no capítulo 3, percebeu-se que o protocolo *DoRiS* poderia ser melhorado em relação ao seguinte aspecto: quando uma estação não tem mensagens não-críticas para enviar, o servidor *DoRiS*, quando adquire o bastão, deve obrigatoriamente emitir uma mensagem de tamanho mínimo para que o bastão possa circular. Suponha então que não haja nenhum processo querendo comunicar, todos os servidores enviam mensagens obrigatórias, na taxa máxima da banda, provocando uma sobrecarga significativa em todas as estações do segmento.

Para evitar tal sobrecarga, um mecanismo de configuração dinâmica do anel não-crítico foi desenvolvido, aproveitando o determinismo da comunicação crítica. Para tanto, utilizou-se o campo $type$ do quadro Ethernet para carregar o pedido de participação de um servidor no anel não-crítico. A convenção adotada usa quatro tipos diferentes, permitindo caracterizar:

- Se a mensagem elementar é obrigatória, ou se ela foi enviada no contexto de uma aplicação. No primeiro caso, os três primeiros dígitos do $type$, expresso em formato hexadecimal, valem 902, e, no segundo caso, 901.

- Se o emissor da mensagem participa do anel não-crítico. Neste caso, o último dígito tem o valor 2. Caso contrário, ele vale 1.

Em cada instante, a composição do anel não-crítico é armazenada por cada servidor numa estrutura de dados, chamada $proc$, de dimensão $nServ$. Cada elemento i desta estrutura contém as informações sobre o servidor S_i: se S_i participa ou não do anel não-crítico, e, se ele participa, quem são seu sucessor e seu predecessor no anel.

A fim de ilustrar o mecanismo de configuração dinâmica, considere o caso de um servidor S_i que ainda não participava do anel não-crítico e que queira se inserir neste anel. S_i começa por observar a comunicação ocorrendo durante as janelas $\mathcal{W}_S$. No final de uma janela vazia, isto é, uma janela $\mathcal{W}_S$ durante a qual S_i não recebe nenhuma mensagem, S_i redefine o anel não-crítico, retirando qualquer participante presente na estrutura $proc$. Quando a sua vez de enviar uma mensagem elementar chega, no *chip* i, S_i se insere na estrutura $proc$, e usa um dos tipos `0x9012` ou `0x9022` para indicar que, depois desta mensagem, ele deve ser considerado membro do anel não-crítico. Quando este mesmo servidor não quer mais participar do anel não-crítico, ele volta a utilizar os dois tipos `0x9011` ou `0x9021` no cabeçalho Ethernet da suas mensagens elementares.

Quando um servidor S_j recebe uma mensagem elementar enviada por um outro servidor S_i, ele efetua as ações seguintes:

- Ele atualiza o valor de $proc$ de acordo com o tipo da mensagem recebida. Se este valor termina com 2, ele insere S_i na estrutura $proc$. Caso contrário, S_j retira S_i do conjunto $proc$.
- Ele define o instante de início da próxima janela $\mathcal{W}_S$ e de fim desta mesma.
- Ele redefine o contador de mensagens não-críticas para 0.

Observar que se o anel não-crítico estiver vazio, um servidor S_i só pode começar a enviar mensagens não-críticas durante o *chip* i. Desta forma, se dois servidores S_i e S_j estiverem em fase de inserção simultaneamente, aquele cujo o *chip* acontecerá antes do outro, suponha i, irá se inserir primeiro. Em seguida, S_j receberá a mensagem elementar de S_i, ainda durante o *chip* i, informando que S_i é membro do anel crítico. Conseqüentemente, S_j deverá inserir S_i no anel não-crítico.

Considere agora o mecanismo de inserção de um servidor S_j no anel não-crítico, quando este anel tiver pelo menos um participante. Neste caso, S_j deve esperar até receber uma men-

sagem não-crítica m. Quando isto ocorre, S_j determina o valor do bastão circulante a partir do identificador m. A cada recepção de mensagens não-críticas, S_j atualiza o valor do bastão. Finalmente, quando o seu *chip* chega, S_j envia uma mensagem elementar informando que ele entra no anel não-crítico. Em algum momento futuro, o bastão chegará a S_j que poderá então começar a enviar mensagens não-críticas.

Para completar a definição deste mecanismo de configuração dinâmica, considerou-se os dois cenários de falhas seguintes. O primeiro corresponde à possibilidade de um servidor com o bastão não enviar mensagem alguma, ou porque ele falhou, ou porque ele perdeu o bastão. O segundo corresponde a perdas eventuais de mensagens elementares.

No primeiro cenário, quando o bastão se perde, o mecanismo apresentado aqui garante que o bastão será redefinido em algum momento futuro. Efetivamente, se o bastão não circula mais, as janelas $\mathcal{W}_S$ ficam vazias. Porém, como foi visto acima, na ocorrência de uma janela $\mathcal{W}_S$ vazia de mensagens, todos os servidores redefinem o anel crítico para o conjunto vazio. Portanto, a geração de um novo bastão se torna possível, permitindo a recuperação da falha.

Para tolerar o segundo cenário de falhas, isto é, falha de omissões de mensagens elementares, um servidor S_i utiliza o contador $cons$ definido na seção 3.4.4. Lembrar que este contador é incrementado a cada mensagem elementar recebida e redefinido para 0 pelo servidor S_i em cada *chip* i. Portanto, se este contador não vale $nServ$ quando o *chip* i começa, isto significa que S_i perdeu alguma mensagem elementar. Nesta caso, S_i deve esperar o próximo ciclo para poder se inserir no anel não-crítico.

Observar que este mecanismo é baseado na seguinte suposição. Um servidor de uma estação hospedando processos nunca deixa de perceber a ausência de mensagem não-críticas numa janela $\mathcal{W}_S$. Dito de forma positiva, um servidor de tarefas e processos sempre percebe a presença de uma mensagem não-crítica numa janela $\mathcal{W}_S$. No caso de uma estação hospedando apenas tarefas, como dispositivos de sensoriamento ou atuação, o servidor não precisa observar a comunicação não-crítica, portanto, esta restrição não se aplica. Por outro lado, as estações que hospedam processos têm capacidade de processamento suficiente para que esta suposição seja considerada verdadeira.

A especificação formal e a verificação automática da correção deste mecanismo de configuração dinâmica foram realizados e se encontram no apêndice C.

5.4.5 A implementação da disciplina *DoRiS*

Em cada estação, a disciplina *DoRiS* utiliza três tarefas de tempo real para organizar o acesso ao meio Ethernet. A primeira tarefa cuida da recepção das mensagens, a segunda é destinada à emissão das mensagens críticas e a terceira é responsável pela emissão de mensagens não-críticas.

A tarefa que cuida da recepção assíncrona de mensagens corresponde ao gerente da pilha (*stack manager*) do RTnet, descrito na seção 5.3.3.2. Ela tem a maior prioridade do sistema. As duas tarefas de tempo real encarregadas de gerenciar as operações de emissão de mensagens de um servidor S_i são denotadas CE_i (*Critical Emission*) e SE_i (*Soft Emission*). A tarefa CT_i é associada ao anel crítico e a tarefa SE_i é responsável pela transmissão de mensagens em $\mathcal{W}_S$. Além do caráter seqüencial das operações de emissões nos dois anéis, a prioridade de CE_i é maior que a de SE_i, o que garante que a comunicação não-crítica não interfira na emissão de mensagens de tempo real crítico.

5.4.5.1 Recepção A principal modificação da tarefa de recepção foi 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* foi inserido no tratamento da interrupção de recepção do pacote. A execução destas operações no tratador de interrupção permitiu descartar os pacotes que não são destinados à estação antes de acordar o "gerente da pilha".

As principais operações de gerenciamento realizadas por um servidor S_i são:

- Num evento de recepção de uma mensagem elementar:
  - Atualizar o valor do instante de emissão da próxima mensagem elementar de S_i;
  - Atualizar o valor do próximo instante de emissão de uma mensagem elementar;
  - Atualizar os contadores $chipCount$ e $cons$;
  - Atualizar a estrutura de dados $proc$, de acordo com as ações apresentadas na seção 5.4.4.
  - Quando em posse do bastão do anel não-crítico, acordar a tarefa de emissão das

mensagens não-críticas.

- Num evento de recepção de uma mensagem não-crítica:
  - Atualizar o valor do bastão do anel não-crítico, de acordo com as informações contidas na estrutura $proc$.
  - Atualizar o contador de mensagens não-críticas utilizado para detecção de janelas $\mathcal{W}_S$ vazias;
  - Quando em posse do bastão, 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 necessários para a análise temporal do protocolo podem eventualmente ser armazenados.

Depois destas operações de gerenciamento do protocolo *DoRiS*, dentro do próprio tratamento da interrupção de chegada de um pacote, o servidor descarta a mensagem se ela não for destinada a alguma aplicação que ele hospeda. Caso contrário, o servidor coloca o pacote na fila de recepção do gerente da pilha, antes de acordá-lo. Assim que ele acorda, o gerente determina a aplicação que está esperando pelo pacote, entregando-o. Ou ele o faz chamando a função de recepção do próprio Linux, se mensagem for não-crítica, ou chamando a função de recepção definida por *DoRiS*, se ela for crítica. Neste caso, a mensagem é encaminhada para o *socket* de tempo real, previamente aberto por alguma aplicação de tempo real com a interface RTDM.

Percebe-se que o gerente da pilha deve ter a maior prioridade do sistema para garantir que uma mensagem de tempo real seja entregue com a latência mínima possível.

5.4.5.2 Emissão de mensagens críticas Para controlar as operações de emissão, os servidores utilizam temporizadores e condições lógicas, assim como foi visto na apresentação de *DoRiS* no capítulo 3. No caso do anel crítico, um servidor S_i deve conhecer, com a melhor precisão possível, o instante t_i do início do próximo *chip* i no qual a tarefa CE_i deve enviar uma mensagem elementar. Considere, por exemplo, o servidor S_1. Suponha que o início do *chip* 1 acontecerá no instante t_1. Quando este instante chega, o temporizador τ_1 de S_1 acorda CE_1 que: (i) envia uma mensagem elementar; e (ii) define o instante t_1' de começo do próximo

chip 1 para o valor:

$$t'_1 = t_1 + nTask * \Delta_C \tag{5.2}$$

Suponha agora que, logo em seguida, S_1 programa o temporizador τ_1 com este valor de t'_1. Entre o instante desta programação, que é quase igual a t_1 e t'_1, várias mensagens elementares irão chegar, e, a cada chegada, S_1 irá atualizar o valor de t'_1, utilizando a equação (5.1). A última destas atualizações de t'_1 acontecerá, na ausência de falha, na chegada da mensagem elementar enviada pelo servidor de identificador $nServ$. Portanto, utilizar esta última atualização de t'_1 para programar τ_1 permite: (i) minimizar o impacto do desvio local do relógio da estação 1; e (ii) compensar eventuais desvios acumulados nas transmissões anteriores das mensagens elementares.

Porém, este valor de t'_1 não é conhecido no instante t_1, pois só será conhecido durante o *chip* anterior a t'_1. Para contornar esta dificuldade, no instante t_1, o servidor S_1 programa τ_1 para acordar CE_1 no meio do *chip* que precede t'_1, sendo o valor deste instante t''_1 estimado pela seguinte formula:

$$t''_1 = t_1 + \left(nTask - \frac{1}{2} \right) * \Delta_C \tag{5.3}$$

Quando a tarefa CE_1 acorda, ela programa τ_1 novamente, utilizando a mais recente atualização de t'_1. Neste instante t'_1, τ_1 finalmente acorda a tarefa CE_1 uma segunda vez, no início exato do seu *slot* elementar.

Observar que quando a fila de mensagens críticas para serem enviadas for vazia, mensagens obrigatórias são criadas e enviadas conforme a especificação de *DoRiS*. Nesta implementação, prioridades não são definidas para as mensagens críticas.

5.4.5.3 Emissão de mensagens não-críticas As emissões de mensagens não-críticas são regidas pela circulação do bastão e por uma condição temporal que garante que estas mensagens sejam enviadas durante uma janela $\mathcal{W}_S$. Foi visto na seção 5.4.5.1 que o contador $token$ é incrementado a cada recepção de mensagens não-críticas e que a tarefa SE_j é acordada quando o servidor S_j adquire o bastão circulante, isto é, quando $token = j$. Nesta ocorrência, SE_j estima o tempo ainda disponível na janela $\mathcal{W}_S$. Se este tempo for maior que o tamanho da

mensagem em espera, SE_j a envia. Caso contrário, SE_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 $\mathcal{W}_S$.

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

5.4.6 Resultados experimentais

O objetivo desta seção é apresentar resultados experimentais obtidos com o protótipo de *DoRiS* desenvolvido. Os cenários de comunicação utilizados foram simples, pois só utilizaram três computadores idênticos. Apesar disto, estes experimentos permitiram testar o protótipo de *DoRiS* e mostrar a sua capacidade em garantir entrega de mensagens com garantias temporais da ordem de alguns micro-segundos.

A configuração dos experimentos envolveu três computadores Pentium IV ($nServ = 3$), com processadores de 2.4 Ghz e 512 MB de memória. A rede utilizada foi constituída de um comutador Ethernet 100Mbps dedicado e de cabos de comprimentos menores que $5m$. O valor de Δ_C foi definido para $500\mu s$.

Inicialmente, verificou-se que os serviços do anel crítico garantiam trocas de mensagens entre duas aplicações de tempo real T_1 e T_2, escritas com a interface RTDM, utilizada em modo usuário. Este cenário permitiu medir o valor de Δ_E, seguindo a metodologia apresentada na seção 5.4.3. Constatou-se que este valor alcançava $25\mu s$ em média e que os desvios máximos deste valor foram de $3\mu s$, na ausência de sobrecarga dos processadores. Apesar de o tempo de transmissão de uma mensagem de 64 bytes ser um pouco menor que $6\mu s$ num barramento 100Mbps, as latências da plataforma Xenomai, da ordem de $10\mu s$, assim como foi visto no capítulo 4, e de transmissão do comutador explicam este valor observado de $25\mu s$. Este primeiro experimento confirmou que *DoRiS* pode oferecer uma taxa de transmissão de $8 * 64 * 2000 = 1Mbps$ para as tarefas de tempo real. Além disso, a variabilidade observada ficou abaixo de 1.2% em valor relativo ($6\mu s$ de desvio máximo para um período de

Em seguida, realizou-se o mesmo experimento, instalando-se em paralelo uma comunicação não-crítica entre o servidor de T_1, e o terceiro servidor ainda não envolvido na comunicação

(mas obviamente executando *DoRiS*). Constatou-se que:

- a comunicação crítica entre T_1 e T_2 não sofreu nenhuma alteração devida à atividade na rede. Em particular, os intervalos de tempo entre a chegada de mensagens críticas em T_1 e T_2 não apresentaram diferenças observáveis com os resultados do experimento anterior, sem comunicação não-crítica;

- a taxa de transferência da comunicação não-crítica alcançou um pouco menos de um terço da taxa da banda, ou seja, aproximadamente $46Mbps$. Mais precisamente, 10.000 mensagens de tamanho 1496 bytes levaram em média $2.6s$ para serem transmitidas. Não se observou nenhuma perda de mensagem não-crítica, em mais de 500.000 enviadas.

Em experimentos similares, realizados com a camada RTnet, carregando simplesmente a política NoMAC, apresentada na seção 5.3.4, obtiveram-se os seguintes resultados. No caso do primeiro experimento, na ausência de comunicação não-crítica, a comunicação crítica de período de $500\mu s$, não sofreu atrasos observáveis. Assim como no experimento com *DoRiS*, os intervalos de tempo entre as chegadas de mensagens críticas foram constantes, apresentando uma variabilidade menor que $3\mu s$, ou seja, abaixo de 1.2% em valor relativo. Isto era esperado, pois o serviço de comunicação RTnet oferece garantias temporais críticas em situações na qual o controle de acesso ao meio não é necessário.

O segundo experimento, no entanto, apresentou resultados significativamente diferentes de *DoRiS*, pois a ausência de disciplina resultou numa variabilidade de $160\mu s$ nos intervalos de tempos entre chegadas de mensagens críticas, ou seja, mais de 10% em valor relativo. Em relação à comunicação não-crítica, a taxa de transmissão da comunicação não-crítica se aproximou de 75Mbps. Porém, observaram-se perdas de mensagens esporádicas, variando de 0 a 10 em cada 10.000 mensagens enviadas.

Estes resultados confirmaram a capacidade deste protótipo de *DoRiS* organizar o acesso ao meio de comunicação Ethernet, para suportar várias aplicações com requisitos temporais críticos e não-críticos, de forma concorrente e determinista. No entanto, experimentos e testes mais complexos deverão ainda ser realizados no contexto do desenvolvimento de uma versão de produção do protocolo *DoRiS*.

5.5 CONCLUSÃO

Neste capítulo, a implementação do protocolo *DoRiS* foi apresentada em detalhes. Descreveu-se, em particular, o grau de integração deste protocolo com a camada RTnet da plataforma de tempo real Xenomai / Linux. Apesar dos testes e experimentos realizados, o protótipo de *DoRiS* está ainda em fase de desenvolvimento. No entanto, as mais de duas mil linhas de código já escritas permitiram confirmar a validade da proposta do protocolo *DoRiS*.

A implementação de código em modo *kernel* representou um grande desafio que não poderia ter sido alcançado se não fossem os inúmeros exemplos de programas fornecidos, tanto pelos desenvolvedores de Xenomai como pelos de RTnet. Deve ser observado, notadamente, que a arquitetura modular do RTnet facilitou significativamente a implementação da disciplina *DoRiS*.

Do ponto de visto do trabalho de implementação, a especificação formal serviu de referência para guiar este trabalho. No entanto, o fato de integrar *DoRiS* como uma disciplina da camada RTnet já existente impediu um desenvolvimento completamente baseado na especificação. Ainda assim, constatou-se que a fase de especificação se revelou um passo importante na compreensão do protocolo e de suas propriedades. Este conhecimento profundo facilitou o presente trabalho, em particular quando se tratou de modificar algum mecanismo.

Por ter sido desenvolvido durante a fase final deste trabalho, a especificação formal do mecanismo de configuração dinâmico do anel não-crítico não foi apresentada no capítulo 3. No entanto, a sua especificação em TLA+ e sua verificação com TLC foram realizadas com sucesso e serviram de base para a implementação. O apêndice C apresenta esta nova versão da especificação.

CAPÍTULO 6

CONCLUSÃO

Nesta dissertação, foi descrito um novo protocolo baseado em dois anéis lógicos que organiza a coexistência de comunicações com requisitos temporais críticos e não-críticos, num mesmo barramento Ethernet.

Graças à utilização da abordagem TDMA para estruturar o anel crítico, *DoRiS* garante previsibilidade para as aplicações com requisitos temporais críticos, enquanto a utilização da abordagem com bastão circulante implícito serviu para controlar eficientemente o acesso ao meio referente ao anel não-crítico. Esta combinação permitiu garantir, por um lado, a confiabilidade do protocolo pelo isolamento temporal dos dois modos de comunicação e, por outro, otimizar o uso da banda.

O protocolo *DoRiS* foi especificado formalmente em TLA+ (*Temporal Logic of Actions*) e várias propriedades temporais do protocolo foram verificadas pelo verificador de modelos TLC (*Temporal Logic Checker*). Especificar o protocolo em TLA+ permitiu adquirir uma compreensão precisa do sistema e das suas funcionalidades, antes de partir para a fase de implementação. Esta fase se beneficiou da especificação formal, que foi utilizada como base para o trabalho de codificação. Apesar de a escolha de usar uma plataforma já existente ter impossibilitado, em parte, um melhor aproveitamento da especificação, pois a codificação do protocolo teve que seguir o padrão da plataforma, chegou-se à conclusão que a fase de especificação facilitou várias decisões importantes tomadas durante a fase de implementação do protocolo.

Para o desenvolvimento deste projeto, uma plataforma de Tempo Real baseada no Sistema Operacional de Propósito Geral - Linux foi escolhida. As principais motivações para tal escolha foram a boa divulgação deste sistema de código aberto nas comunidades de pesquisa e a existência de extensões de tempo real para Linux. Algumas destas extensões foram apresentadas e experimentos foram realizados para medir as latências de interrupção e ativação de tarefas de tempo real. Os resultados obtidos permitiram confirmar a viabilidade do uso destes ambientes

operacionais para desenvolver *DoRiS*. Em particular, as plataformas de código aberto RTAI e Xenomai / Linux apresentaram propriedades temporais satisfatórias, conseguindo garantir o escalonamento de tarefas de tempo real com tempos de latências abaixo de 20 microsegundos.

A implementação de *DoRiS* foi então realizada na plataforma Xenomai / Linux, podendo ser facilmente transportada para a plataforma RTAI / Linux. A pilha de rede determinista RTnet, já desenvolvida para estas duas plataformas, foi aproveitada, e *DoRiS* foi inserido, sob a forma de uma nova disciplina desta pilha. Devido à complexidade do desenvolvimento de um protocolo de rede determinista num ambiente operacional tal como Xenomai / Linux, a implementação foi limitada a uma versão simplificada de *DoRiS*, sem o mecanismo de reserva. No entanto, um mecanismo de configuração dinâmica do anel não-crítico foi desenvolvido para diminuir a sobrecarga gerada pela transmissão de mensagens de controle.

O protótipo de *DoRiS*, com sua estrutura operacional em dois anéis, foi utilizado para realizar alguns experimentos que confirmaram a capacidade do protocolo *DoRiS* atender requisitos de sistemas híbridos, compostos de aplicações com requisitos temporais críticos e não-críticos. Em particular, os experimentos mostraram que *DoRiS* consegue garantir determinismo para as mensagens críticas e taxas de transmissões altas para a comunicação não-crítica. Os mesmos experimentos realizados com a pilha RTnet, sem disciplina de acesso ao meio, provocaram desvios significativos nos tempos de recepção das mensagens críticas e perdas de mensagens não-críticas.

No decorrer deste trabalho, vários aspectos foram abordados de maneira superficial. Outros foram deixados para trabalhos futuros dado o horizonte temporal para concluir a presente dissertação. Trabalhos futuros poderão:

- Completar o estudo experimental do comportamento de *DoRiS* com cenários de comunicação de redes industriais. Em particular, cenários de falhas deverão ser utilizados para testar as propriedades de confiabilidade de *DoRiS*.

- Estudar os protocolos de gerenciamento da composição e reconfiguração dinâmica dos grupos, e propor soluções adequadas para aumentar a confiabilidade e a flexibilidade de *DoRiS*.

- Acrescentar a implementação de *DoRiS* com o mecanismo de reserva descrito na sua especificação.

- Definir políticas de escalonamento das mensagens, baseadas no mecanismo de reserva e em possíveis extensões deste mecanismo. O aumento do número de *slots* de reserva por *chip* poderá, por exemplo, ser utilizado para aumentar a flexibilidade do protocolo.

Finalmente, por ser um protocolo distribuído, descentralizado e determinista, baseado num único barramento compartilhado, com capacidades de tolerância a falhas de omissão de mensagens e de paradas de estações, *DoRiS* apresenta características interessantes para ser adaptado no contexto de redes sem fios. No entanto, tal aplicação depende da implementação de um protocolo eficiente de gerenciamento e reconfiguração dinâmica da composição dos anéis críticos e não-críticos.

Acredita-se que este trabalho tenha trazido contribuições para o campo de pesquisa de redes para sistemas de tempo real. Alguns conceitos aqui apresentados, como o mecanismo de reserva e a adoção de métodos formais na fase inicial de concepção do protocolo de comunicação, deverão certamente influenciar trabalhos de pesquisa futuros nesta área.

APÊNDICE A

ESPECIFICAÇÃO FORMAL DE DORIS EM TLA+

Um exemplo de arquivo de configuração é apresentado na figura A.1. Este arquivo é utilizado pelo verificador de modelo e permite de definir os parâmetros da verificação sem alterar a especificação.

Em seguida figura o módulo completo da especificação de *DoRiS*.

Neste módulo, as barras horizontais são puramente estéticas. Elas são utilizadas para separar as ações principais e outros segmentos de códigos. Também para facilitar a leitura, escolheu-se de colocar cada ação numa página. Os textos escritos em fontes menores sob fundos cinzas são comentários.

Observar também que as funções ou ações precedidas de "A_" ou "B_" não são utilizadas nesta especificação. Elas apenas estão colocadas para ilustrar as mudanças de cenário de verificação possíveis.

Na primeira linha de código, o módulo carrega quatro outros módulos de TLA+, usando a palavra chave EXTENDS. Esta palavra permite utilizar as funções definidas nos módulos citados.

This is a TLC configuration file for testing *DoRiS*

This statement tells TLC that it is to take formula *Spec* as the specification it is checking.

$SPECIFICATION\ \ Spec$

This statement defines the type invariants of formula *Spec*

$INVARIANTS\ \ TypeInvariance$

This statement tells TLC to check that the specification implies the listed properties. In TLA+, a specification is also a property.

$PROPERTIES$

$EmptyMedium$
$CollisionAvoidance$
$HardRingCorrectness$
$ReservationSafety$
$SoftRingFairness$
$Omission$
$Failure$

This 3 properties are false and are used to generate counter-examples

$NoCollisionAvoidance$
$NoReservationSafety$
$NoOmission$

This statement defines the constants of the *Spec*

CONSTANTS

$nTask = 11$
$nProc = 7$
$deltaChip = 300$
$delta = 6$
$pi = 111$
$maxTxTime = 122$

Figura A.1: Arquivo de configuração da especificação de *DoRiS*

MODULE $DoRiS$

EXTENDS $Naturals$, $Reals$, TLC, $Sequences$

VARIABLES $Shared$, Distributed vision shared by tasks and processes
$TaskState$, Tasks local state
$ProcState$, Processes local state
$History$ An observer variable

CONSTANTS $nTask$, Number of hard tasks
$nProc$, Number of soft processes
$deltaChip$, Duration length of $DoRiS$ Period,
$delta$, Duration length of hard message slot
pi, Slot message processing time
$maxTxTime$ Maximum soft message sizes

Miscellaneous definitions

$min(Set) \triangleq$ CHOOSE $m \in Set : \forall\, y \in Set : m \leq y$

$Task \triangleq 1 \,..\, nTask$ Tasks indices
$Proc \triangleq 1 \,..\, nProc$ Processes indices

$Ti \triangleq [i \in Task \mapsto \langle \text{“T”}, i \rangle]$ Tasks tuple
$Pj \triangleq [j \in Proc \mapsto \langle \text{“P”}, j \rangle]$ Processes tuple

$TaskSet \triangleq \{Ti[i] : i \in Task\}$ Tasks identifiers set
$ProcSet \triangleq \{Pj[j] : j \in Proc\}$ Processes identifiers set

$procId(P) \triangleq$ CHOOSE $j \in Proc : P = Pj[j]$
$taskId(T) \triangleq$ CHOOSE $i \in Task : T = Ti[i]$

$vars \triangleq \langle Shared, ProcState, TaskState, History \rangle$

Two alternative soft message list fabric.

$$
\begin{aligned}
list(j) \triangleq\ & \text{CASE } j \in \{1\} \rightarrow [i \in 1 \,..\, 4 \mapsto [txTime \mapsto maxTxTime]] \\
& \square\ j \in \{2\} \rightarrow \langle\rangle \\
& \square\ j \in \{7, 14\} \rightarrow [i \in 1 \,..\, 2 \mapsto [txTime \mapsto maxTxTime]] \\
& \square\ \text{OTHER} \rightarrow 1 :> [txTime \mapsto maxTxTime]
\end{aligned}
$$

$A\_list(j) \triangleq 1 :> [txTime \mapsto maxTxTime]$

(*Continue*)

$$
\begin{array}{l}
B\_list(j) \triangleq \text{IF } j = 1 \\
\quad \text{THEN } 1 :> [txTime \mapsto maxTxTime] \mathbin{@@} 2 :> [txTime \mapsto maxTxTime] \mathbin{@@} \\
\qquad 3 :> [txTime \mapsto maxTxTime] \mathbin{@@} 4 :> [\, txTime \mapsto 50 \,] \\
\quad \text{ELSE IF } j = 2 \text{ THEN } \langle\rangle \text{ ELSE } 1 :> [txTime \mapsto maxTxTime] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbin{@@} 2 :> [\, txTime \mapsto maxTxTime \,]
\end{array}
$$

Initializations of the variables.

$$
\begin{array}{l}
Init \triangleq \\
\quad \land Shared = [chipTimer \mapsto 0,\ chipCount \mapsto 1,\ macTimer \mapsto 0,\ medium \mapsto \{\}] \\
\quad \land TaskState = [i \in Task \mapsto [msg \mapsto \langle\rangle,\ execTimer \mapsto Infinity, \\
\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto [j \in Task \mapsto -1],\ cons \mapsto nTask - i + 1]] \\
\quad \land ProcState = [j \in Proc \mapsto [token \mapsto 1,\ list \mapsto list(j),\ count \mapsto 0]] \\
\quad \land History = [elem \mapsto 0,\ rese \mapsto 0]
\end{array}
$$

The elementary and reservation slot action.
$reservation(i)$ is an arbitrary reservation function that states that tasks wants to reserve as much slots as possible.
"pi" is the time the message will need to be processed by others tasks.

$$
\begin{array}{l}
reservation(i) \triangleq \\
\quad \text{IF } TaskState[i].cons = nTask \\
\quad \text{THEN } \{j \in Task : TaskState[i].res[j] = -1\} \\
\quad \text{ELSE } \{(((i-1) + (nTask - 1)) \ \%\ nTask) + 1\}
\end{array}
$$

$$
\begin{array}{l}
SendElem(T) \triangleq \\
\quad \land Shared.medium = \{\} \\
\quad \land Shared.chipTimer = 0 \\
\quad \land \text{LET } i \triangleq taskId(T) \\
\qquad \text{IN } \land Shared.chipCount = i \\
\qquad\quad \land \text{LET } resSet \triangleq reservation(i) \\
\qquad\qquad \text{IN } \land Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.macTimer = delta, \\
\qquad\qquad\qquad\qquad !.medium = \{[id \mapsto i,\ type \mapsto \text{“hard”},\ res \mapsto resSet]\}] \\
\qquad\qquad\qquad \land TaskState' = [TaskState \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad ![i].res = [j \in Task \mapsto \text{IF } j \in resSet \text{ THEN } i \text{ ELSE } @[j]], \\
\qquad\qquad\qquad\qquad ![i].cons = 1] \\
\qquad\qquad \land History' = [History \text{ EXCEPT } !.elem = @ + 1] \\
\qquad\qquad \land \text{UNCHANGED } ProcState
\end{array}
$$

(*Continue*)

```
SendRese(T) ≜
    ∧ Shared.medium = {}
    ∧ Shared.chipTimer = delta
    ∧ LET i ≜ taskId(T)
      IN ∧ TaskState[i].res[Shared.chipCount] = i
         ∧ Shared' = [Shared EXCEPT
               !.macTimer = delta,
               !.medium = {[id ↦ i, type ↦ "hard", res ↦ { − 1}]}]
         ∧ TaskState' = [j ∈ Task ↦ [TaskState[j] EXCEPT
               !.res[Shared.chipCount] = − 1]]
         ∧ History' = [History EXCEPT !.rese = @ + 1]
         ∧ UNCHANGED ProcState
```

The soft window action. *lenTX* is the message transmission time. The *Failed* set has to be define according to which processes failure scenario is chosen.

```
Failed ≜ CASE Shared.chipCount = 2 → {3}
          □ Shared.chipCount ∈ {3, 4} → {3, 5}
          □ Shared.chipCount = 5 → {3, 5}
          □ OTHER → {}

lenMsg(i) ≜ IF ProcState[i].list ≠ ⟨⟩ THEN Head(ProcState[i].list).txTime ELSE delta

SendSoft(P) ≜
    ∧ 2 ∗ delta ≤ Shared.chipTimer
    ∧ Shared.chipTimer ≤ deltaChip
    ∧ Shared.medium = {}
    ∧ LET i ≜ procId(P)
          lenTX ≜ lenMsg(i)
          d ≜ Shared.chipTimer + lenTX
          NoMsg ≜ i ∈ Failed ∨ d > deltaChip
      IN ∧ i = ProcState[i].token
         ∧ Shared' = [Shared EXCEPT
               !.macTimer = IF NoMsg THEN Infinity ELSE lenTX,
               !.medium = IF NoMsg THEN @ ELSE {[id ↦ i, type ↦ "soft"]}]
         ∧ ProcState' = [ProcState EXCEPT
               ![i].token = IF NoMsg THEN @ ELSE (@ % nProc) + 1,
               ![i].list = IF d > deltaChip ∨ @ = ⟨⟩ THEN @ ELSE Tail(@),
               ![i].count = IF NoMsg THEN @ ELSE @ + 1]
         ∧ UNCHANGED ⟨TaskState, History⟩
```

(*Continue*)

The hard and soft messages receiving actions. We may introduce omission failures in the hard message reception action. *noMsgReceptionSet* defines the set of nodes that shall not receive a message. When an omission failure happens, the "Reserv" and "Consistency" sequence are not updated.

$$NoRecvSet(m) \triangleq \text{IF } Shared.chipCount \in \{2, 4\} \text{ THEN } \{m.id, 3\} \text{ ELSE } \{m.id\}$$

$$A\_NoRecvSet(m) \triangleq \{m.id\}$$

$$\begin{aligned}
&RecvHard(m) \triangleq \\
&\quad \land\ m.type = \text{“hard”} \\
&\quad \land\ Shared.macTimer = 0 \\
&\quad \land\ Shared' = [Shared \text{ EXCEPT } !.medium = \{\}] \\
&\quad \land\ TaskState' = \\
&\qquad [i \in NoRecvSet(m) \mapsto TaskState[i]]\ @@ \\
&\qquad [i \in Task \setminus NoRecvSet(m) \mapsto [TaskState[i] \text{ EXCEPT} \\
&\qquad\quad !.msg = Append(@, m), \\
&\qquad\quad !.execTimer = \text{IF } Len(TaskState[i].msg) = 0 \text{ THEN } pi \text{ ELSE } @, \\
&\qquad\quad !.cons = \text{IF } m.res \neq \{-1\} \text{ THEN } @ + 1 \text{ ELSE } @, \\
&\qquad\quad !.res = \text{IF } m.res = \{-1\} \\
&\qquad\qquad\qquad \text{THEN } [j \in Task \mapsto \text{IF } j = m.id \text{ THEN } -1 \text{ ELSE } @[j]] \\
&\qquad\qquad\qquad \text{ELSE } [j \in Task \mapsto \text{IF } j \in m.res \text{ THEN } m.id \text{ ELSE } @[j]]]] \\
&\quad \land\ \text{UNCHANGED } \langle ProcState, History \rangle
\end{aligned}$$

$$\begin{aligned}
&RecvSoft(m) \triangleq \\
&\quad \land\ m.type = \text{“soft”} \\
&\quad \land\ Shared.macTimer = 0 \\
&\quad \land\ Shared' = [Shared \text{ EXCEPT } !.medium = \{\}] \\
&\quad \land\ ProcState' = [j \in \{m.id\} \mapsto ProcState[j]]\ @@ \\
&\qquad\qquad [j \in Proc \setminus \{m.id\} \mapsto [ProcState[j] \text{ EXCEPT} \\
&\qquad\qquad\qquad !.token = (@ \ \%\ nProc) + 1, \\
&\qquad\qquad\qquad !.count = @ + 1]] \\
&\quad \land\ \text{UNCHANGED } \langle TaskState, History \rangle
\end{aligned}$$

NextTick and *NextChip* are the only actions that update timers according to passage of time. *NextTick* search the best next time increment and updates all timers *NextChip* reset the *ChipTimer* and increment *ChipCount* modulo the number of task. It sets the arbitrary soft sequence of message of each process during every cycle. Alltogether, this 2 actions allow the time circularity of the model. The count counter allows to check if a soft message has been sent in the previous chip. If it is not the case, all processes increment the token value by 1 at the end of the chip.

(*Continue*)

```
NextTick ≜
    LET noRese ≜  ∧ Shared.medium = {}
                  ∧ Shared.chipTimer = delta
                  ∧ ∀ i ∈ Task : TaskState[i].res[Shared.chipCount] ≠ i
        tmp ≜  min({TaskState[i].execTimer : i ∈ Task} ∪
                   {deltaChip − Shared.chipTimer})
        d ≜  IF noRese THEN min({delta, tmp}) ELSE  min({Shared.macTimer, tmp})
    IN  ∧ d > 0
        ∧ Shared′ = [Shared EXCEPT
              !.chipTimer = @ + d,
              !.macTimer = IF noRese
                           THEN @
                           ELSE  IF @ = Infinity THEN Infinity  ELSE  @ − d]
        ∧ TaskState′ = [i ∈ Task ↦ [TaskState[i] EXCEPT
              !.msg = IF TaskState[i].execTimer − d = 0  THEN Tail(@) ELSE  @,
              !.execTimer = IF @ − d = 0
                            THEN IF Len(TaskState[i].msg) > 1 THEN pi ELSE  Infinity
                            ELSE  IF @ = Infinity THEN @ ELSE  @ − d]]
        ∧ UNCHANGED ⟨ProcState, History⟩

NextChip ≜
    ∧  Shared.medium = {}
    ∧  Shared.chipTimer = deltaChip
    ∧  LET Overflow ≜ ∃ j ∈ Proc : Len(ProcState[j].list) > 14
           NextCycle ≜ Shared.chipCount′ = 1
       IN  ∧ Shared′ = [Shared EXCEPT
                 !.macTimer = 0,
                 !.chipCount = (@ % nTask) + 1,
                 !.chipTimer = IF Overflow THEN  − 1 ELSE  0]
           ∧ ProcState′ = [j ∈ Proc ↦ [ProcState[j] EXCEPT
                 !.token = IF ProcState[j].count = 0 THEN  (@ % nProc) + 1 ELSE  @,
                 !.count = 0,
                 !.list = IF NextCycle THEN @ ∘ list(j) ELSE  @]]
           ∧ IF NextCycle
             THEN History′ = [elem ↦ 0, rese ↦ 0]
             ELSE  UNCHANGED History
    ∧  UNCHANGED TaskState
```

(Continue)

Next defines the set of possible actions that doesn't alterate the timers. *Init* is the initial state, *Next* is the disjunct of steps choices, and *Tick* describes the passage of time.

$$Tick \triangleq NextTick \lor NextChip$$

$$\begin{aligned} Next \triangleq\ & \lor \exists\, T \in TaskSet : SendElem(T) \lor SendRese(T) \\ & \lor \exists\, P \in ProcSet : SendSoft(P) \\ & \lor \exists\, msg \in Shared.medium : RecvHard(msg) \lor RecvSoft(msg) \end{aligned}$$

$$Liveness \triangleq \Box\Diamond Tick$$

$$Spec \triangleq Init \land \Box[Next \lor Tick]_{vars} \land Liveness$$

The following temporal properties are checked when specified in the *DoRiS.cfg* configuration file.
"*TypeInvariance*" checks the variables type invariance.
"*CollisionAvoidance*" checks that the token can be hold by only one task or process at once. "*HardRingCorrectness*" makes use of the "History" variable to check that all mandatory elementary message are sent in a *DoRiS* cycle. "*ReservationSafety*" guaranty that if a task holds the token in a reservation slot, then all other tasks are aware of its reservation.
"*SoftRingFairness*" guaranty that each process will eventually receive the token.
"Omission" states that some task omission failure takes place.
"Failure" states that some process failure takes place.
"*NoReservationSafety*", "*NoCollisionAvoidance*" and "*NoOmission*" are contradiction of the respective properties, used to generate handfull conter-examples.

$$HardMsg \triangleq Seq([id : Task,\ type : \{\text{“hard”}\},\ res : \text{SUBSET}\ (\{-1\} \cup Task)])$$

$$\begin{aligned} MediumMsg \triangleq\ & \{m : m \in [id : Proc,\ type : \{\text{“soft”}\}] \cup \\ & \qquad [id : Task,\ type : \{\text{“hard”}\},\ res : \text{SUBSET}\ (\{-1\} \cup Task)]\} \end{aligned}$$

$$\begin{aligned} TypeInvariance \triangleq\ & \\ & \land Shared.chipCount \in Task \\ & \land Shared.chipTimer \in 0\,..\,deltaChip \\ & \land Shared.macTimer \in 0\,..\,maxTxTime \cup \{Infinity\} \\ & \land \forall\, m \in Shared.medium : m \in MediumMsg \\ & \land ProcState \in [Proc \to [token : Proc,\ count : 0\,..\,50, \\ & \qquad list : \{\langle\rangle\} \cup Seq([txTime : \{maxTxTime\}])]] \\ & \land TaskState \quad \in [Task \to [msg : \{\langle\rangle\} \cup HardMsg,\ res : [Task \to \{-1\} \cup Task\ ], \\ & \qquad\qquad execTimer : 0\,..\,pi \cup \{Infinity\},\ cons : Task]] \end{aligned}$$

(*Continue*)

$$
\begin{array}{l}
Send(Q) \triangleq \ \lor \ \land \ Q \in TaskSet \\
\qquad\qquad\qquad \land \ (\text{ENABLED } SendElem(Q) \lor \text{ENABLED } SendRese(Q)) \\
\qquad\qquad \lor \ \land \ Q \in ProcSet \\
\qquad\qquad\qquad \land \ \text{ENABLED } SendSoft(Q)
\end{array}
$$

$$
\begin{array}{l}
CollisionAvoidance \triangleq \ \forall \, P, \ Q \in TaskSet \cup ProcSet : \\
\qquad\qquad\qquad\qquad\qquad \Box(\text{ENABLED } (Send(P) \land Send(Q)) \implies (P = Q))
\end{array}
$$

$$
\begin{array}{l}
NoCollisionAvoidance \triangleq \ \exists \, P, \ Q \in TaskSet \cup ProcSet : \\
\qquad\qquad\qquad\qquad\qquad \Diamond((P \neq Q) \land \text{ENABLED } (Send(P) \land Send(Q)))
\end{array}
$$

$$
\begin{array}{l}
HardRingCorrectness \triangleq \\
\qquad \land \forall \, T \in TaskSet : \ \land \Box(Len(TaskState[taskId(T)].msg) \leq 3) \\
\qquad\qquad\qquad\qquad\qquad \land \Box\Diamond\text{ENABLED } SendElem(T) \\
\qquad \land \Box(\text{ENABLED } NextChip \implies History.elem = Shared.chipCount)
\end{array}
$$

$$
\begin{array}{l}
ReservationSafety \triangleq \\
\qquad \Box \ \forall \, chip, \, j \in Task : \ \land \text{ENABLED } SendRese(Ti[j]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \land \ Shared.chipCount = chip \\
\qquad\qquad \implies \ \land \ TaskState[j].res[chip] = j \\
\qquad\qquad\qquad \land \forall \, i \in Task \setminus \{j\} : TaskState[i].res[chip] \in \{j, \ -1\}
\end{array}
$$

$$
\begin{array}{l}
SoftRingFairness \triangleq \\
\qquad \land \forall \, i \in Proc : \Box\Diamond(i = ProcState[i].token) \\
\qquad \land \Box\Diamond(\forall \, i \in Proc \setminus Failed : Len(ProcState[i].list) = 0)
\end{array}
$$

$$
\begin{array}{l}
NoReservationSafety \triangleq \\
\qquad \Box\Diamond \ \exists \, chip \in Task : \exists \, j \in Task : \\
\qquad\qquad\qquad \land \ TaskState[j].res[chip] \neq \ -1 \\
\qquad\qquad\qquad \land \text{ENABLED } SendRese(Ti[j]) \\
\qquad\qquad\qquad \land \ Shared.chipCount = chip \\
\qquad\qquad\qquad \land \exists \, i \in Task \setminus \{j\} : \neg TaskState[i].res[chip] \in \{j, \ -1\}
\end{array}
$$

$$
\begin{array}{l}
Omission \triangleq \ \exists \, T \in TaskSet : \\
\qquad\qquad \Diamond(\text{ENABLED } SendElem(T) \ \land TaskState[taskId(T)].cons < nTask)
\end{array}
$$

$$
\begin{array}{l}
NoOmission \triangleq \ \forall \, T \in TaskSet : \\
\qquad\qquad \Box(\text{ENABLED } SendElem(T) \implies TaskState[taskId(T)].cons = nTask)
\end{array}
$$

$$
Failure \triangleq \ \Box\Diamond(\exists \, P \in ProcSet : procId(P) \in Failed)
$$

APÊNDICE B

EXEMPLOS DE TRACES PRODUZIDOS POR TLC

Os dois traces apresentados aqui correspondem as execuções de TLC para verificar as propriedades $CollisionAvoidance$ e $NoCollisionAvoidance$ (ver figura 3.16). Na primeira execução (coluna de esquerda), TLC não detectou violação da propriedade $CollisionAvoidance$ e na segunda, TLC detectou a sua violação e produziu o contra-exemplo mostrado abaixo, na coluna direita e demais páginas. Observar as informações de cobertura de cada ação da especificação que permitem detectar erros eventuais. Aqui, as ações nunca verdadeiras (linha 130, 131, 229 e 232 da especificação) foram causadas pelo uso dos valores $nTask = 1$ e $nProc = 1$.

```
TLC Version 2.0 of November 26, 2005
Model-checking
Parsing file DoRiS.tla
Parsing file /opt/TLA/tlasany/StandardModules/Naturals.tla
Parsing file /opt/TLA/tlasany/StandardModules/Reals.tla
Parsing file /opt/TLA/tlasany/StandardModules/TLC.tla
Parsing file /opt/TLA/tlasany/StandardModules/Sequences.tla
Parsing file /opt/TLA/tlasany/StandardModules/Integers.tla
Semantic processing of module Naturals
Semantic processing of module Integers
Semantic processing of module Reals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module DoRiS
Implied-temporal checking--satisfiability problem has 4 branches.
Finished computing initial states: 1 distinct state generated.
--Checking temporal properties for the complete state space...
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
    calculated (optimistic):  1.5124620306172787E-17
    based on the actual fingerprints:  3.24949862660705E-15
The coverage stats:
  line 101, col 16 to line 102, col 66 of module DoRiS: 1
  line 103, col 16 to line 103, col 59 of module DoRiS: 1
  line 104, col 26 to line 104, col 34 of module DoRiS: 1
  line 130, col 24 to line 130, col 72 of module DoRiS: 0
  line 131, col 34 to line 131, col 42 of module DoRiS: 0
  line 135, col 30 to line 135, col 78 of module DoRiS: 3
  line 136, col 40 to line 136, col 48 of module DoRiS: 3
  line 138, col 30 to line 140, col 79 of module DoRiS: 28
  line 141, col 30 to line 144, col 78 of module DoRiS: 28
  line 145, col 29 to line 145, col 37 of module DoRiS: 31
  line 145, col 40 to line 145, col 46 of module DoRiS: 31
  line 162, col 10 to line 162, col 50 of module DoRiS: 2
  line 163, col 10 to line 171, col 93 of module DoRiS: 2
  line 172, col 23 to line 172, col 31 of module DoRiS: 2
  line 172, col 34 to line 172, col 40 of module DoRiS: 2
  line 177, col 10 to line 177, col 50 of module DoRiS: 28
  line 178, col 10 to line 181, col 81 of module DoRiS: 28
  line 182, col 23 to line 182, col 31 of module DoRiS: 28
  line 182, col 34 to line 182, col 40 of module DoRiS: 28
  line 210, col 13 to line 214, col 77 of module DoRiS: 31
  line 215, col 13 to line 219, col 87 of module DoRiS: 31
  line 220, col 26 to line 220, col 34 of module DoRiS: 31
  line 220, col 37 to line 220, col 43 of module DoRiS: 31
  line 225, col 10 to line 227, col 67 of module DoRiS: 1
  line 229, col 16 to line 231, col 86 of module DoRiS: 0
  line 232, col 26 to line 232, col 32 of module DoRiS: 0
  line 234, col 16 to line 237, col 86 of module DoRiS: 1
  line 238, col 16 to line 238, col 52 of module DoRiS: 1
  line 239, col 20 to line 239, col 28 of module DoRiS: 1
  line 81, col 20 to line 83, col 84 of module DoRiS: 1
  line 84, col 20 to line 86, col 86 of module DoRiS: 1
  line 87, col 14 to line 87, col 57 of module DoRiS: 1
  line 88, col 24 to line 88, col 32 of module DoRiS: 1
  line 98, col 16 to line 100, col 79 of module DoRiS: 1
96 states generated, 93 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 93.
```

```
TLC Version 2.0 of November 26, 2005
Model-checking
Parsing file DoRiS.tla
Parsing file /opt/TLA/tlasany/StandardModules/Naturals.tla
Parsing file /opt/TLA/tlasany/StandardModules/Reals.tla
Parsing file /opt/TLA/tlasany/StandardModules/TLC.tla
Parsing file /opt/TLA/tlasany/StandardModules/Sequences.tla
Parsing file /opt/TLA/tlasany/StandardModules/Integers.tla
Semantic processing of module Naturals
Semantic processing of module Integers
Semantic processing of module Reals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module DoRiS
Implied-temporal checking--satisfiability problem has 1 branches.
Finished computing initial states: 1 distinct state generated.
--Checking temporal properties for the complete state space...
Error: Temporal properties were violated.
The following behaviour constitutes a counter-example:

STATE 1: <Initial predicate>
/\ Shared = [chipTimer |-> 0, chipCount |-> 1, macTimer |-> 0, medium |-> {}]
/\ TaskState = <<[msg |-> <<  >>, execTimer |-> 999999999, res |-> <<-1>>, cons |-> 1]>>
/\ History = [elem |-> 0, rese |-> 0]
/\ ProcState = <<[list |-> <<[txTime |-> 122]>>, token |-> 1, count |-> 0]>>

STATE 2: <Action line 76, col 7 to line 88, col 86 of module DoRiS>
/\ Shared = [ chipTimer |-> 0,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[res |-> {1}, id |-> 1, type |-> "hard"]} ]
/\ TaskState = <<[msg |-> <<  >>, execTimer |-> 999999999, res |-> <<1>>, cons |-> 1]>>
/\ History = [elem |-> 1, rese |-> 0]

STATE 3: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 6,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[res |-> {1}, id |-> 1, type |-> "hard"]} ]

STATE 4: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 6, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 5: <Action line 94, col 7 to line 104, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 6,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[res |-> {-1}, id |-> 1, type |-> "hard"]} ]
/\ TaskState = <<[msg |-> <<  >>, execTimer |-> 999999999, res |-> <<-1>>, cons |-> 1]>>
/\ History = [elem |-> 1, rese |-> 1]

STATE 6: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 12,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[res |-> {-1}, id |-> 1, type |-> "hard"]} ]

STATE 7: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 12, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 8: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 12,
  chipCount |-> 1,
  macTimer |-> 122,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
```

```
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 1]>>

STATE 9: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 134,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 10: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 134, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 11: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 134,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 2]>>

STATE 12: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 140,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 13: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 140, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 14: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 140,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 3]>>

STATE 15: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 146,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 16: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 146, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 17: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 146,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 4]>>

STATE 18: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 152,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 19: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 152, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 20: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 152,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 5]>>

STATE 21: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 158,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 22: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 158, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 23: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 158,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 6]>>

STATE 24: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 164,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 25: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 164, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 26: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 164,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 7]>>

STATE 27: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 170,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 28: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 170, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 29: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 170,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 8]>>

STATE 30: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 176,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 31: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 176, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 32: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 176,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 9]>>

STATE 33: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 182,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 34: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 182, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 35: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 182,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 10]>>

STATE 36: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 188,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 37: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 188, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 38: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 188,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 11]>>

STATE 39: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 194,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 40: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 194, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 41: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 194,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 12]>>

STATE 42: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 200,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 43: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 200, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 44: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 200,
  chipCount |-> 1,
```

```
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 13]>>

STATE 45: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 206,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 46: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 206, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 47: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 206,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 14]>>

STATE 48: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 212,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 49: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 212, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 50: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 212,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 15]>>

STATE 51: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 218,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 52: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 218, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 53: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 218,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 16]>>

STATE 54: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 224,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 55: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 224, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 56: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 224,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 17]>>

STATE 57: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 230,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 58: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 230, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 59: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 230,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 18]>>

STATE 60: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 236,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 61: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 236, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 62: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 236,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 19]>>

STATE 63: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 242,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 64: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 242, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 65: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 242,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 20]>>

STATE 66: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 248,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 67: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 248, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 68: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 248,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 21]>>

STATE 69: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 254,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 70: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 254, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 71: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 254,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 22]>>

STATE 72: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 260,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 73: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 260, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 74: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 260,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 23]>>

STATE 75: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 266,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 76: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 266, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 77: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 266,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 24]>>

STATE 78: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 272,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 79: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 272, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 80: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
```

```
/\ Shared = [ chipTimer |-> 272,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 25]>>

STATE 81: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 278,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 82: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 278, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 83: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 278,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 26]>>

STATE 84: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 284,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 85: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 284, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 86: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 284,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 27]>>

STATE 87: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 290,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 88: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 290, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 89: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [ chipTimer |-> 290,
  chipCount |-> 1,
  macTimer |-> 6,
  medium |-> {[id |-> 1, type |-> "soft"]} ]
/\ ProcState = <<[list |-> <<  >>, token |-> 1, count |-> 28]>>

STATE 90: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [ chipTimer |-> 296,
  chipCount |-> 1,
  macTimer |-> 0,
  medium |-> {[id |-> 1, type |-> "soft"]} ]

STATE 91: <Action line 251, col 14 to line 251, col 70 of module DoRiS>
/\ Shared = [chipTimer |-> 296, chipCount |-> 1, macTimer |-> 0, medium |-> {}]

STATE 92: <Action line 124, col 7 to line 145, col 79 of module DoRiS>
/\ Shared = [chipTimer |-> 296, chipCount |-> 1, macTimer |-> 999999999, medium |-> {}]

STATE 93: <Action line 209, col 10 to line 220, col 87 of module DoRiS>
/\ Shared = [chipTimer |-> 300, chipCount |-> 1, macTimer |-> 999999999, medium |-> {}]

STATE 94: Back to state 1.

The coverage stats:
  line 101, col 16 to line 102, col 66 of module DoRiS: 1
  line 103, col 16 to line 103, col 59 of module DoRiS: 1
  line 104, col 26 to line 104, col 34 of module DoRiS: 1
  line 130, col 24 to line 130, col 72 of module DoRiS: 0
  line 131, col 34 to line 131, col 42 of module DoRiS: 0
  line 135, col 30 to line 135, col 78 of module DoRiS: 3
  line 136, col 40 to line 136, col 48 of module DoRiS: 3
  line 138, col 30 to line 140, col 79 of module DoRiS: 28
  line 141, col 30 to line 144, col 78 of module DoRiS: 28
  line 145, col 29 to line 145, col 37 of module DoRiS: 31
  line 145, col 40 to line 145, col 46 of module DoRiS: 31
  line 162, col 10 to line 162, col 50 of module DoRiS: 2
  line 163, col 10 to line 171, col 93 of module DoRiS: 2
  line 172, col 23 to line 172, col 31 of module DoRiS: 2
  line 172, col 34 to line 172, col 40 of module DoRiS: 2
  line 177, col 10 to line 177, col 50 of module DoRiS: 28
  line 178, col 10 to line 181, col 81 of module DoRiS: 28
  line 182, col 23 to line 182, col 31 of module DoRiS: 28
  line 182, col 34 to line 182, col 40 of module DoRiS: 28
  line 210, col 13 to line 214, col 77 of module DoRiS: 31
  line 215, col 13 to line 219, col 87 of module DoRiS: 31
  line 220, col 26 to line 220, col 34 of module DoRiS: 31
  line 220, col 37 to line 220, col 43 of module DoRiS: 31
  line 225, col 10 to line 227, col 67 of module DoRiS: 1
  line 229, col 16 to line 231, col 86 of module DoRiS: 0
  line 232, col 26 to line 232, col 32 of module DoRiS: 0
  line 234, col 16 to line 237, col 86 of module DoRiS: 1
  line 238, col 16 to line 238, col 52 of module DoRiS: 1
  line 239, col 20 to line 239, col 28 of module DoRiS: 1
  line 81, col 20 to line 83, col 84 of module DoRiS: 1
  line 84, col 20 to line 86, col 86 of module DoRiS: 1
  line 87, col 14 to line 87, col 57 of module DoRiS: 1
  line 88, col 24 to line 88, col 32 of module DoRiS: 1
  line 98, col 16 to line 100, col 79 of module DoRiS: 1
96 states generated, 93 distinct states found, 0 states left on queue.
```

APÊNDICE C

ESPECIFICAÇÃO DE DORIS COM CONFIGURAÇÃO DINÂMICA

Por ter sido desenvolvido durante a fase final deste trabalho, a especificação formal do mecanismo de configuração dinâmica do anel não-crítico não foi apresentada no capítulo 3. No entanto, a sua especificação em TLA+ e sua verificação com TLC foram realizados com sucesso. Este apêndice apresenta esta nova versão da especificação.

This is a TLC configuration file for testing *DoRiS*

This statement tells TLC that it is to take formula *Spec* as the specification it is checking.

$SPECIFICATION\ \ Spec$

This statement defines the type invariants of formula *Spec*

$INVARIANTS\ \ TypeInvariance$

This statement tells TLC to check that the specification implies the listed properties. In TLA+, a specification is also a property.

$PROPERTIES$

$CollisionAvoidance$

$HardRingCorrectness$

$ReservationSafety$

$SoftRingFairness$

$Omission$

$Failure$

This 3 properties are false and are used to generate counter-examples

$NoCollisionAvoidance$

$NoReservationSafety$

$NoOmission$

This statement defines the constants of the *Spec*

CONSTANTS

$nServ = 7$

$deltaChip = 300$

$delta = 6$

$pi = 111$

$maxTxTime = 122$

$horiz = 5$

Figura C.1: Arquivo de configuração da especificação de *DoRiS*

MODULE $DoRiS$

EXTENDS $Naturals$, $Reals$, TLC, $Sequences$

VARIABLES $Shared$, Distributed vision shared by tasks and processes
$TaskState$, Tasks local state
$ProcState$, Processes local state
$History$ An observer variable

CONSTANTS $nServ$, Number of hard tasks
$deltaChip$, Duration length of $DoRiS$ Period,
$delta$, Duration length of hard message slot
pi, Slot message processing time
$maxTxTime$, Maximum soft message sizes
$horiz$

Miscellaneous definitions

$min(Set) \triangleq$ CHOOSE $m \in Set : \forall\, y \in Set : m \leq y$

$next(i, Set) \triangleq$ IF $\forall\, j \in Set : j \leq i$
THEN $min(Set)$
ELSE $min(\{j \in Set : j > i\})$

$Task \triangleq 1 \,..\, nServ$ Tasks indices
$T \triangleq [i \in Task \mapsto \langle \text{“T”}, i \rangle]$ Tasks identifiers array
$TaskSet \triangleq \{T[i] : i \in Task\}$ Tasks identifiers set
$taskId(t) \triangleq$ CHOOSE $i \in Task : t = T[i]$

$Proc \triangleq 1 \,..\, nServ$
$P \triangleq [j \in Proc \mapsto \langle \text{“P”}, j \rangle]$ Processes identifiers array

$vars \triangleq \langle Shared, ProcState, TaskState, History \rangle$

Two alternative soft message list fabric.

$C\_list(j, cycle) \triangleq$
CASE $j \in \{cycle\} \rightarrow [i \in 1 \,..\, 4 \mapsto [txTime \mapsto maxTxTime]]$
$\square$ OTHER $\rightarrow \langle\rangle$

$list(j, cycle) \triangleq$
CASE $j \in \{cycle\} \rightarrow [i \in 1 \,..\, 4 \mapsto [txTime \mapsto maxTxTime]]$
$\square\ j \in \{2, 3\} \setminus \{cycle\} \rightarrow \langle\rangle$
$\square\ j \in \{4, 7\} \setminus \{cycle\} \rightarrow [i \in 1 \,..\, 2 \mapsto [txTime \mapsto maxTxTime]]$
$\square$ OTHER $\rightarrow 1 :> [txTime \mapsto maxTxTime]$

$A\_list(j) \triangleq 1 :> [txTime \mapsto maxTxTime]$

$B\_list(j) \triangleq$ IF $j = 1$
THEN $1 :> [txTime \mapsto maxTxTime] \,@@\, 2 :> [txTime \mapsto maxTxTime] \,@@$
$3 :> [txTime \mapsto maxTxTime] \,@@\, 4 :> [\ txTime \mapsto 50\ ]$
ELSE IF $j = 2$ THEN $\langle\rangle$ ELSE $1 :> [txTime \mapsto maxTxTime]$
$@@\, 2 :> [\ txTime \mapsto maxTxTime\ ]$

(*Continue*)

Initializations of the variables.

$$
\begin{array}{l}
Init \triangleq \\
\quad \land Shared = [proc \mapsto 1 \,..\, nServ,\ chipTimer \mapsto 0,\ chipCount \mapsto 1,\ cycleCount \mapsto 1, \\
\qquad\qquad macTimer \mapsto 0,\ medium \mapsto \{\}] \\
\quad \land TaskState = [i \in Task \mapsto [msg \mapsto \langle\rangle,\ execTimer \mapsto Infinity, \\
\qquad\qquad\qquad\qquad res \mapsto [j \in Task \mapsto -1],\ cons \mapsto nServ - i + 1]] \\
\quad \land ProcState = \\
\qquad [j \in Shared.proc \mapsto [token \mapsto 1,\ list \mapsto list(j, 1),\ count \mapsto 0]]\ @@ \\
\qquad [j \in Proc \setminus Shared.proc \mapsto [token \mapsto -1,\ list \mapsto list(j, 1),\ count \mapsto 0]] \\
\quad \land History = [elem \mapsto 0,\ rese \mapsto 0]
\end{array}
$$

$ProcSet(S) \triangleq \{P[j] : j \in S\}$ Processes identifiers set

$procId(p, S) \triangleq$ CHOOSE $j \in S : p = P[j]$

The elementary and reservation slot action.
$reservation(i)$ is an arbitrary reservation function that states that tasks wants to reserve as much slots as possible.
"pi" is the time the message will need to be processed by others tasks.

$$
\begin{array}{l}
reservation(i) \triangleq \\
\quad \text{IF } TaskState[i].cons = nServ \\
\quad \text{THEN } \{j \in Task : TaskState[i].res[j] = -1\} \\
\quad \text{ELSE } \{(((i - 1) + (nServ - 1))\ \%\ nServ) + 1\}
\end{array}
$$

$$
\begin{array}{l}
SendElem(t) \triangleq \\
\quad \land Shared.medium = \{\} \\
\quad \land Shared.chipTimer = 0 \\
\quad \land \text{LET } i \triangleq taskId(t) \\
\qquad\quad flag \triangleq \text{IF } ProcState[i].list \neq \langle\rangle \text{ THEN } 1 \text{ ELSE } 0 \\
\qquad \text{IN } \land Shared.chipCount = i \\
\qquad\quad \land \text{LET } resSet \triangleq reservation(i) \\
\qquad\qquad \text{IN } \land Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.macTimer = delta, \\
\qquad\qquad\qquad\qquad !.medium = \{[id \mapsto i,\ type \mapsto \text{“hard”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto resSet,\ procFlag \mapsto flag]\}] \\
\qquad\qquad\qquad\quad \land TaskState' = [TaskState \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad ![i].res = [j \in Task \mapsto \text{IF } j \in resSet \text{ THEN } i \text{ ELSE } @[j]], \\
\qquad\qquad\qquad\qquad\qquad ![i].cons = 1] \\
\qquad\qquad\qquad\quad \land ProcState' = [ProcState \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad ![i].token = \text{IF } flag = 0 \text{ THEN } -1 \text{ ELSE } @] \\
\qquad\qquad\qquad\quad \land History' = [History \text{ EXCEPT } !.elem = @ + 1]
\end{array}
$$

(*Continue*)

```
SendRese(t) ≜
    ∧ Shared.medium = {}
    ∧ Shared.chipTimer = delta
    ∧ LET i ≜ taskId(t)
      IN ∧ TaskState[i].res[Shared.chipCount] = i
         ∧ Shared' = [Shared EXCEPT
                !.macTimer = delta,
                !.medium = {[id ↦ i, type ↦ "hard", res ↦ { − 1}]}]
         ∧ TaskState' = [j ∈ Task ↦ [TaskState[j] EXCEPT
                !.res[Shared.chipCount] = − 1]]
         ∧ History' = [History EXCEPT !.rese = @ + 1]
         ∧ UNCHANGED ProcState
```

The soft window action. *lenTX* is the message transmission time. The *Failed* set has to be define according to which processes failure scenario is chosen.

```
Failed ≜ CASE  Shared.chipCount = 2 → {3}
             □ Shared.chipCount ∈ {3, 4} → {3, 5}
             □ Shared.chipCount = 5 → {3, 5}
             □ OTHER → {}

A_Failed ≜ {}

lenMsg(i) ≜
    IF ProcState[i].list ≠ ⟨⟩ THEN Head(ProcState[i].list).txTime ELSE delta

SendSoft(p) ≜
    ∧ 2 ∗ delta ≤ Shared.chipTimer
    ∧ Shared.chipTimer ≤ deltaChip
    ∧ Shared.medium = {}
    ∧ LET i ≜ procId(p, Proc)
          lenTX ≜ lenMsg(i)
          d ≜ Shared.chipTimer + lenTX
          NoMsg ≜ i ∈ Failed ∨ d > deltaChip
          Cond ≜ ∧ i = Shared.chipCount
                 ∧ ProcState[i].list = ⟨⟩
      IN ∧ i = ProcState[i].token
         ∧ Shared' = [Shared EXCEPT
                !.macTimer = IF NoMsg THEN Infinity ELSE lenTX,
                !.medium = IF NoMsg THEN @ ELSE {[id ↦ i, type ↦ "soft"]}]
         ∧ ProcState' = [ProcState EXCEPT
                ![i].token = IF NoMsg THEN @ ELSE IF Cond
                                                  THEN − 1
                                                  ELSE next(i, Shared.proc),
                ![i].list = IF d > deltaChip ∨ @ = ⟨⟩ THEN @ ELSE Tail(@),
                ![i].count = IF NoMsg THEN @ ELSE @ + 1]
         ∧ UNCHANGED ⟨TaskState, History⟩
```

(*Continue*)

The hard and soft messages receiving actions. We may introduce omission failures in the hard message reception action. *noMsgReceptionSet* defines the set of nodes that shall not receive a message. When an omission failure happens, the "Reserv" and "Consistency" sequence are not updated.

$NoRecvSet(m) \triangleq$ IF $Shared.chipCount \in \{2, 4\}$ THEN $\{m.id, 3\}$ ELSE $\{m.id\}$

$A\_NoRecvSet(m) \triangleq \{m.id\}$

$RecvHard(m) \triangleq$
$\wedge\ m.type =$ "hard"
$\wedge\ Shared.macTimer = 0$
$\wedge$ CASE $m.res \neq \{-1\} \wedge m.procFlag = 1$
$\rightarrow \wedge\ Shared' = [Shared$ EXCEPT $!.medium = \{\},$
$!.proc = @ \cup \{m.id\}]$
$\wedge$ UNCHANGED $ProcState$
$\square\ m.res \neq \{-1\} \wedge m.procFlag = 0$
$\rightarrow \wedge\ Shared' = [Shared$ EXCEPT $!.medium = \{\},$
$!.proc = @ \setminus \{m.id\}]$
$\wedge\ ProcState' =$
$[i \in (Proc \setminus Shared.proc) \cup NoRecvSet(m) \mapsto ProcState[i]]\ @@$
$[i \in Shared.proc \setminus NoRecvSet(m) \mapsto [ProcState[i]$ EXCEPT
$!.token =$ IF $m.id = ProcState[i].token$
THEN $next(m.id, Shared.proc)$
ELSE $@]]$
$\square\ m.res = \{-1\}$
$\rightarrow \wedge\ Shared' = [Shared$ EXCEPT $!.medium = \{\}]$
$\wedge$ UNCHANGED $ProcState$
$\wedge\ TaskState' =$
$[i \in NoRecvSet(m) \mapsto TaskState[i]]\ @@$
$[i \in Task \setminus NoRecvSet(m) \mapsto [TaskState[i]$ EXCEPT
$!.msg = Append(@, m),$
$!.execTimer =$ IF $Len(TaskState[i].msg) = 0$ THEN pi ELSE $@,$
$!.cons =$ IF $m.res \neq \{-1\}$ THEN $@ + 1$ ELSE $@,$
$!.res =$ IF $m.res = \{-1\}$
THEN $[j \in Task \mapsto$ IF $j = m.id$ THEN -1 ELSE $@[j]]$
ELSE $[j \in Task \mapsto$ IF $j \in m.res$ THEN $m.id$ ELSE $@[j]]]]$
$\wedge$ UNCHANGED $History$

(*Continue*)

```
RecvSoft(m) ≜
    ∧ m.type = "soft"
    ∧ Shared.macTimer = 0
    ∧ LET lastSoft ≜  ∧ 2 * delta ≤ Shared.chipTimer
                      ∧ Shared.chipTimer ≤ deltaChip
                      ∧ ∀ j ∈ Proc : ProcState[j].token = -1
      IN Shared' = [Shared EXCEPT !.medium = {},
                                  !.macTimer = IF lastSoft THEN Infinity ELSE @,
                                  !.proc = IF lastSoft THEN {} ELSE @]
    ∧ ProcState' = [j ∈ (Proc \ Shared.proc) ∪ {m.id} ↦ ProcState[j]] @@
                   [j ∈ Shared.proc \ {m.id} ↦ [ProcState[j] EXCEPT
                       !.token = next(@, Shared.proc),
                       !.count = @ + 1]]
    ∧ UNCHANGED ⟨TaskState, History⟩
```

NextTick and *NextChip* are the only actions that update timers according to passage of time. *NextTick* search the best next time increment and updates all timers *NextChip* reset the *ChipTimer* and increment *ChipCount* modulo the number of task. It sets the arbitrary soft sequence of message of each process during every cycle. Alltogether, this 2 actions allow the time circularity of the model. The count counter allows to check if a soft message has been sent in the previous chip. If it is not the case, all processes increment the token value by 1 at the end of the chip.

```
NextTick ≜
    LET noRese ≜  ∧ Shared.medium = {}
                  ∧ Shared.chipTimer = delta
                  ∧ ∀ i ∈ Task : TaskState[i].res[Shared.chipCount] ≠ i
        noSoft ≜  ∧ 2 * delta ≤ Shared.chipTimer
                  ∧ Shared.chipTimer ≤ deltaChip
                  ∧ Shared.medium = {}
                  ∧ ∀ j ∈ Proc : ProcState[j].token = -1
        tmp ≜ min({TaskState[i].execTimer : i ∈ Task} ∪
                  {deltaChip - Shared.chipTimer})
        d ≜ CASE noRese → min({delta, tmp})
              □ noSoft  → min({tmp})
              □ OTHER   → min({Shared.macTimer, tmp})
    IN ∧ d > 0
       ∧ Shared' = [Shared EXCEPT
             !.chipTimer = @ + d,
             !.macTimer = CASE noRese → @
                             □ noSoft  → Infinity
                             □ @ = Infinity → Infinity
                             □ OTHER → @ - d]
       ∧ TaskState' = [i ∈ Task ↦ [TaskState[i] EXCEPT
             !.msg = IF TaskState[i].execTimer - d = 0 THEN Tail(@) ELSE @,
             !.execTimer = IF @ - d = 0
                           THEN IF Len(TaskState[i].msg) > 1 THEN pi ELSE Infinity
                           ELSE IF @ = Infinity THEN @ ELSE @ - d]]
       ∧ UNCHANGED ⟨ProcState, History⟩
```

(*Continue*)

$$
\begin{array}{l}
NextChip \triangleq \\
\quad \land\ Shared.medium = \{\} \\
\quad \land\ Shared.chipTimer = deltaChip \\
\quad \land\ \text{LET } Overflow \triangleq \exists j \in Shared.proc : Len(ProcState[j].list) > 14 \\
\qquad\qquad TimeCircle \triangleq Shared.cycleCount = horiz \\
\qquad\qquad NextCycle \triangleq Shared.chipCount = nServ \\
\qquad \text{IN } \land\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.macTimer = 0, \\
\qquad\qquad\qquad\qquad !.chipCount = (@\ \%\ nServ) + 1, \\
\qquad\qquad\qquad\qquad !.chipTimer = \text{IF } Overflow \text{ THEN } -1 \text{ ELSE } 0, \\
\qquad\qquad\qquad\qquad !.cycleCount = \text{IF } TimeCircle \text{ THEN } 1 \text{ ELSE IF } NextCycle \text{ THEN } @ + 1 \text{ ELSE } @] \\
\qquad\qquad\qquad \land\ ProcState' = [j \in (Proc \setminus Shared.proc) \mapsto ProcState[j]]\ @@ \\
\qquad\qquad\qquad\qquad\qquad\qquad [j \in Shared.proc \mapsto [ProcState[j] \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad !.token = \text{IF } ProcState[j].count = 0 \text{ THEN } (@\ \%\ nServ) + 1 \text{ ELSE } @, \\
\qquad\qquad\qquad\qquad !.count = 0, \\
\qquad\qquad\qquad\qquad !.list = \text{IF } NextCycle \text{ THEN } @ \circ list(j, Shared.cycleCount) \text{ ELSE } @]] \\
\qquad\qquad\qquad \land\ \text{IF } NextCycle \\
\qquad\qquad\qquad\qquad \text{THEN } History' = [elem \mapsto 0, rese \mapsto 0] \\
\qquad\qquad\qquad\qquad \text{ELSE UNCHANGED } History \\
\quad \land\ \text{UNCHANGED } TaskState
\end{array}
$$

Next defines the set of possible actions that doesn't alterate the timers. *Init* is the initial state, *Next* is the disjunct of steps choices, and *Tick* describes the passage of time.

$$Tick \triangleq NextTick \lor NextChip$$

$$
\begin{array}{l}
Next \triangleq \lor \exists t \in TaskSet : SendElem(t) \lor SendRese(t) \\
\qquad\quad \lor \exists p \in ProcSet(Shared.proc) : SendSoft(p) \\
\qquad\quad \lor \exists msg \in Shared.medium : RecvHard(msg) \lor RecvSoft(msg)
\end{array}
$$

$$Liveness \triangleq \Box\Diamond Tick$$

$$Spec \triangleq Init \land \Box[Next \lor Tick]_{vars} \land Liveness$$

(*Continue*)

The following temporal properties are checked when specified in the $DoRiS.cfg$ configuration file.
"$TypeInvariance$" checks the variables type invariance.
"$CollisionAvoidance$" checks that the token can be hold by only one task or process at once. "$HardRingCorrectness$" makes use of the "History" variable to check that all mandatory elementary message are sent in a $DoRiS$ cycle. "$ReservationSafety$" guaranty that if a task holds the token in a reservation slot, then all other tasks are aware of its reservation.
"$SoftRingFairness$" guaranty that each process will eventually receive the token.
"Omission" states that some task omission failure takes place.
"Failure" states that some process failure takes place.
"$NoReservationSafety$", "$NoCollisionAvoidance$" and "$NoOmission$" are contradiction of the respective properties, used to generate handfull conter-examples.

$$HardMsg \triangleq Seq([id : Task,\ type : \{\text{“hard”}\},\ res : \text{SUBSET } (\{-1\} \cup Task),\ procFlag : 0\,..\,1] \cup [id : Task,\ type : \{\text{“hard”}\},\ res : \{\{-1\}\}])$$

$$MediumMsg \triangleq \{m : m \in [id : Proc,\ type : \{\text{“soft”}\}] \cup [id : Task,\ type : \{\text{“hard”}\},\ res : \text{SUBSET } (\{-1\})] \cup [id : Task,\ type : \{\text{“hard”}\},\ res : \text{SUBSET } (\{-1\} \cup Task),\ procFlag : 0\,..\,1\ ]\}$$

$$\begin{aligned} TypeInvariance \triangleq\ & \\ & \wedge Shared.chipCount \in 1\,..\,nServ \\ & \wedge Shared.cycleCount \in 1\,..\,horiz \\ & \wedge Shared.chipTimer \in 0\,..\,deltaChip \\ & \wedge Shared.macTimer \in 0\,..\,maxTxTime \cup \{Infinity\} \\ & \wedge \forall\, m \in Shared.medium : m \in MediumMsg \\ & \wedge ProcState \in [Proc \to [token : Shared.proc \cup \{-1\},\ count : 0\,..\,50, \\ & \qquad list : \{\langle\rangle\} \cup Seq([txTime : \{maxTxTime\}])]] \\ & \wedge TaskState \in [\ Task \to [\ msg : \{\langle\rangle\} \cup HardMsg,\ res : [\ Task \to \{-1\} \cup Task\ ], \\ & \qquad execTimer : 0\,..\,pi \cup \{Infinity\},\ cons : Task\ ]\ ] \end{aligned}$$

$$\begin{aligned} Send(q) \triangleq\ & \vee \wedge q \in TaskSet \\ & \quad \wedge (\text{ENABLED } SendElem(q) \vee \text{ENABLED } SendRese(q)) \\ & \vee \wedge q \in ProcSet(Shared.proc) \\ & \quad \wedge \text{ENABLED } SendSoft(q) \end{aligned}$$

$$CollisionAvoidance \triangleq \forall\, p,\ q \in TaskSet \cup ProcSet(Proc) : \Box(\text{ENABLED } (Send(p) \wedge Send(q)) \implies (p = q))$$

$$NoCollisionAvoidance \triangleq \exists\, p,\ q \in TaskSet \cup ProcSet(Proc) : \Diamond((p \neq q) \wedge \text{ENABLED } (Send(p) \wedge Send(q)))$$

$$\begin{aligned} HardRingCorrectness \triangleq\ & \\ & \wedge \forall\, t \in TaskSet : \wedge \Box(Len(TaskState[taskId(t)].msg) \leq 3) \\ & \qquad \wedge \Box\Diamond\text{ENABLED } SendElem(t) \\ & \wedge \Box(\text{ENABLED } NextChip \implies History.elem = Shared.chipCount) \end{aligned}$$

(*Continue*)

$$
\begin{array}{l}
\mathit{ReservationSafety} \triangleq \\
\quad \Box\ \forall\, \mathit{chip}, j \in \mathit{Task} : \ \wedge\ \text{ENABLED}\ \mathit{SendRese}(T[j]) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{Shared.chipCount} = \mathit{chip} \\
\qquad \Longrightarrow\ \ \wedge\ \mathit{TaskState}[j].\mathit{res}[\mathit{chip}] = j \\
\qquad\quad \wedge \forall\, i \in \mathit{Task} \setminus \{j\} : \mathit{TaskState}[i].\mathit{res}[\mathit{chip}] \in \{j,\ -1\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{SoftRingFairness} \triangleq \\
\quad \wedge \forall\, i \in \mathit{Proc} : \Box(\text{IF}\ i \in \mathit{Shared.proc} \\
\qquad\qquad\qquad\qquad \text{THEN}\ \mathit{ProcState}[i].\mathit{list} \neq \langle\rangle \implies \Diamond(i = \mathit{ProcState}[i].\mathit{token}) \\
\qquad\qquad\qquad\qquad \text{ELSE TRUE}) \\
\quad \wedge \Box\Diamond(\forall\, i \in \mathit{Proc} \setminus \mathit{Failed} : \text{IF}\ i \in \mathit{Shared.proc} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN}\ \mathit{Len}(\mathit{ProcState}[i].\mathit{list}) = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE TRUE})
\end{array}
$$

$$
\begin{array}{l}
\mathit{NoReservationSafety} \triangleq \\
\quad \Box\Diamond\ \exists\, \mathit{chip} \in \mathit{Task} : \exists j \in \mathit{Task} : \\
\qquad \wedge\ \mathit{TaskState}[j].\mathit{res}[\mathit{chip}] \neq -1 \\
\qquad \wedge\ \text{ENABLED}\ \mathit{SendRese}(T[j]) \\
\qquad \wedge\ \mathit{Shared.chipCount} = \mathit{chip} \\
\qquad \wedge \exists\, i \in \mathit{Task} \setminus \{j\} : \neg\, \mathit{TaskState}[i].\mathit{res}[\mathit{chip}] \in \{j,\ -1\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Omission} \triangleq \exists\, t \in \mathit{TaskSet} : \\
\qquad \Diamond(\text{ENABLED}\ \mathit{SendElem}(t)\ \ \wedge\ \mathit{TaskState}[\mathit{taskId}(t)].\mathit{cons} < \mathit{nServ})
\end{array}
$$

$$
\begin{array}{l}
\mathit{NoOmission} \triangleq \forall\, t \in \mathit{TaskSet} : \\
\qquad \Box(\text{ENABLED}\ \mathit{SendElem}(t)\ \ \Longrightarrow\ \mathit{TaskState}[\mathit{taskId}(t)].\mathit{cons} = \mathit{nServ})
\end{array}
$$

$$
\mathit{Failure} \triangleq \Box\Diamond(\exists\, p \in \mathit{ProcSet}(\mathit{Proc}) : \mathit{procId}(p,\ \mathit{Proc}) \in \mathit{Failed})
$$

REFERÊNCIAS

ABADI, M.; LAMPORT, L. An old-fashioned recipe for real time. *ACM Trans. on Programming Languages and Systems*, v. 16, n. 5, p. 1543–1571, 1994.

ALUR, R.; HENZINGER, T.; HO, P.-H. Automatic symbolic verification of embedded systems. *IEEE Trans. on Software Engineering*, v. 22, p. 181–201, 1996.

AUDSLEY, N. C. et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, v. 8, n. 5, p. 284–292, 1993.

AUSLANDER, D. M. What is mechatronics. *IEEE/ASME Trans. on Mechatronics*, v. 1, n. 1, p. 5–9, 1996.

BACH, M. *The Design of the Unix Operating System*. [S.l.]: Prentice-Hall, 1986.

BARABANOV, M. *A Linux based real-time operating system*. Dissertação (Mestrado) — New Mexico Institution of Mining and Technology, 1997.

BARBOZA, F. et al. Specification and verification of the IEEE 802.11 medium access control and an analysis of its applicability to real-time systems. In: *BSFM*. [S.l.: s.n.], 2006. v. 1, p. 9–26.

BENOIT, K.; YAGHMOUR, K. *Preempt-RT and I-pipe: the numbers*. 2005. Http://marc.info/?l=linux-kernel&m=112086443319815&w=2. Last access 03/08.

BERRY, G.; GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, v. 19, n. 2, p. 87–152, 1992.

BOVET, M. C. D. P. *Understanding the Linux Kernel*. 3rd. ed. [S.l.]: O'Reilly, 2005.

CALANDRINO, J. et al. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 2006. p. 111–126.

CARPENZANO, A. et al. Fuzzy traffic smoothing: An approach for real-time communication over Ethernet networks. In: *Proc. of the 4th IEEE Int. Workshop Factory Communication Systems*. [S.l.: s.n.], 2002. p. 241–248.

CARREIRO, F. B.; FONSECA, J. A.; PEDREIRAS, P. Virtual Token-Passing Ethernet - VTPE. In: *Proc. FeT2003 5th IFAC Int. Conf. on Fieldbus Systems and their Applications*. Aveiro, Portugal: [s.n.], 2003.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, v. 43, n. 2, p. 225–267, 1996. A preliminary version appeared in.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. *Model Checking*. [S.l.]: MIT Press, 1999.

ControlNet. *ControlNet specifications - edition 1.03*. 1997. ControlNet International.

CRISTIAN, F. Agreeing on who is present and who is absent in a synchronous distributed system. In: *Proc. of 18th International Conference on Fault-Tolerant Computing*. Tokyio, Japan: IEEE Computer Society Press, 1988.

CROW, B. et al. Ieee 802.11 wireless local area networks. *Communications Magazine, IEEE*, v. 35, n. 9, p. 116–126, 1997.

DAWS, C. et al. The tool Kronos. In: *Proc. of Hybrid Systems III, Verification and Control, LNCS 1066*. [S.l.]: Springer-Verlag, 1996. p. 208–219.

DECOTIGNIE, J.-D. Ethernet-based real-time and industrial communications. *Proc. IEEE (Special issue on industrial communication systems)*, v. 93, n. 6, p. 1102–1117, Jun. 2005.

DIJKSTRA, E. Solution of a problem in concurrent programming and control. *Communications of the ACM*, v. 8, n. 9, p. 569, September 1965.

DOLEJS, O.; SMOLIK, P.; HANZALEK, Z. On the Ethernet use for real-time publish-subscribe based applications. In: *Proc. IEEE Int. Workshop Factory Communication Systems*. [S.l.: s.n.], 2004. p. 39–44.

DOZIO, L.; MANTEGAZZA, P. Linux real time application interface (RTAI) in low cost high performance motion control. In: *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*. [S.l.: s.n.], 2003.

FELSER, M. Real-time ethernet-industry prospective. *Invited Paper - Proceedings of the IEEE*, v. 93, n. 6, p. 1118–1129, 2005.

FISCHER, S. Implementation of multimedia systems based on a real-time extension of Estelle. In: *Formal Description Techniques IX: Theory, application and tools*. [S.l.]: Chapman & Hall, 1996. p. 310–326.

FREESCALE. *MC9S12NE64 Datasheet*. 2008. Http://www.freescale.com - Last access: Jan 08.

FRY, G.; WEST, R. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In: *Proceedings of the International Conference on Embedded Systems and Applications (ESA'07)*. [S.l.: s.n.], 2007.

GERUM, P. *Life with ADEOS*. 2005. www.xenomai.org/documentation/branches/v2.0.x/pdf/Life-with-Adeos.pdf. Last access 01 / 2008.

GORZ, A. *Écologie et Liberté*. [S.l.]: Galilée, 1977.

GUO, H.-B.; KUO, G.-S. CSMA with priority reservation by interruptions for efficiency improvement and QoS support. In: *IEEE Consumer Communications and Networking Conference*. [S.l.: s.n.], 2005.

HANSSEN, F.; MADER, A.; JANSEN, P. G. Verifying the distributed real-time network protocol RTnet using UPPAAL. In: *14th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecom. Systems*. [S.l.]: IEEE Computer Society Press, 2006.

HANSSEN, F. T. Y.; JANSEN, P. G. *Real-time communication protocols: an overview*. The Netherlands, 2003.

HANSSEN, F. T. Y. et al. RTnet: a distributed real-time protocol for broadcast-capable networks. In: EDITION pp electronic (Ed.). *IEEE Joint Int. Conf. on Autonomic and Autonomous Systems and Int. Conf. on Networking and Services (ICAS/ICNS)*. Los Alamitos, California, held in Papeete, French Polynesia: IEEE Computer Society Press, 2005. p. 168–177.

HENZINGER, T.; MANNA, Z.; PNUELI, A. Temporal proof methodologies for real-time systems. In: *Proc. of the 18th Annual Symposium on Principles of Programming Languages*. [S.l.]: ACM Press, 1992. p. 353–366.

I. MOLNAR et al. *PreemptRT*. 2008. http://rt.wiki.kernel.org - Last access jan. 08.

IEEE. *IEEE Standards For Local And Metropolitan Area Networks: Enhancements for Physical Layer Diversity (Redundant Media Control Unit) - Withdrawn Standard*. 1992. IEEE Std 802.4b.

IEEE. *IEEE Standard for Information Technology - Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. 1999. ANSI/IEEE Std 802.11.

IEEE. *Information Technology - Telecommunications and Information exchange between systems - Local and Metropolitan Area Networks specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access and method Physical Layer Specifications*. 2001. ISO/IEC 8802-3.

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

ISO. *Road vehicles-exchange of digital information-Controller Area Network (CAN) for high-speed communication*. 1993. ISO Std. 11 898.

J. KISZKA et al. *RTnet*. 2008. http://www.rtnet.org - Last access jan. 08.

JOHNSON, J. E. et al. Formal specification of a web services protocol. In: *Proc. of Web Services and Formal Methods*. Pisa, Italy: [s.n.], 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.

KOPETZ, H. et al. The Time-Triggered Ethernet (TTE) design. In: *Eighth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. [S.l.: s.n.], 2005.

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

L. TORVALDS et al. *Kernel*. 2008. http://www.kernel.org - Last access jan. 08.

LAMPORT, L. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, v. 17, n. 8, p. 453–455, AUGUST 1974.

LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, v. 6, n. 2, p. 254–280, 1984.

LAMPORT, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, ACM Press, New York, NY, USA, v. 16, n. 3, p. 872–923, 1994. ISSN 0164-0925.

LAMPORT, L. *Specifying Systems: The TLA+ language and tools for hardware and software engineers*. 1st. ed. [S.l.]: Addison Wesley, 2002.

LAMPORT, L.; MELLIAR-SMITH, M. The Part-Time Parliament. *Journal of ACM*, v. 16, n. 2, p. 133–169, May 1998.

LAMPORT, L.; MELLIAR-SMITH, M. Real-time model checking is really simple. In: BORRIONE, I. D.; PAUL, W. J. (Ed.). *Correct Hardware Design and Verification Methods*. [S.l.]: Springer-Verlag, 2005. (LNCS, v. 3725), p. 162–175.

LAQUA, H.; NIEDERMEYER, H.; WILLMANN, I. Ethernet-based real-time control data bus. *IEEE Transactions on Nuclear Science*, v. 49, n. 2, p. 478–482, April 2002.

LARSEN, K. G.; PETERSON, P.; YI, W. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, v. 1, n. 1-2, p. 134–152, 1997.

LELANN, G.; RIVIERRE, N. *Real-time communication over broadcast networks: The CSMA-DCR and the DOD-CSMA-CD protocols*. [S.l.], 1993.

LIAN, F.-L.; MOYNE, J. R.; TILBURY, D. M. *Performance evaluation of control networks: Ethernet, ControlNet and DeviceNet*. [S.l.], Feb. 1999.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogram in a hard real-time environment". *Journal of ACM*, v. 20, n. 1, p. 40–61, 1973.

LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice-Hall, 2000.

LO BELLO, L.; MIRABELLA, O. Design issues for Ethernet in automation. In: *Proc. 8th IEEE Int. Conference on Emerging Technologies and Factory Automation*. Antibes Juan-les-pins, France: EFTA 2001, 2001. p. 213–221.

MAXIM/DALLAS Semicondutor. *DS80C410 and DS80C411 Datasheet*. 2008. Http://www.maxim-ic.com/quick_view2.cfm/qv_pk/4535 - Last access: Jan 08.

MCKENNEY, P. *A realtime preemption overview*. 2005. Http://lwn.net/Articles/146861/ - Last access jun. 08.

MELLOR-CRUMMEY, J. M.; SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 9, n. 1, p. 21–65, 1991.

MOLLE, M. *A new binary logarithmic arbitration method for Ethernet*. University of Toronto, Canada, July 1994.

MOLNAR, I. *kernel/timer.c design*. 2005. Internet, http://lkml.org/lkml/2005/10/19/46 - Last access 01/08.

OLIVEIRA, R. S. de; CARISSIMI, A. da S.; TOSCANI, S. S. *Sistemas Operacionais*. [S.l.]: SagraLuzzatto, 2001. ISBN 85-241-0643-3.

P. GERUM et al. *Xenomai*. 2008. http://www.xenomai.org - Last access jan. 08.

P. MANTEGAZZA et al. *RTAI*. 2008. http://www.rtai.org - Last access jan. 08.

PEDREIRAS, P.; ALMEIDA, L.; GAI, P. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency. In: *EUROMICRO Conf. Real-Time Systems*. [S.l.: s.n.], 2002. v. 2, p. 1631–1637.

PICCIONI, C. A.; TATIBANA, C. Y.; OLIVEIRA, R. S. de. *Trabalhando com o Tempo Real em Aplicações Sobre o Linux*. Florianópolis -SC, Dezembro 2001.

PNUELI, A. The temporal semantics of concurrent programs. In: *Proc. of the Int. Symp. on Semantics of Concurrent Computation*. London, UK: Springer-Verlag, 1979. p. 1–20. ISBN 3-540-09511-X.

PRITTY, D. et al. A realtime upgrade for Ethernet based factory networking. In: *Int. Conf. Industrial Electronics (IECON)*. [S.l.: s.n.], 1995.

PRITTY, D. W. et al. Performance assessment of a deterministic access protocol for high performance bus topology LANs. In: *IEEE Symposium on Computers and Computation*. Alexandria, Egypt: [s.n.], 1995. p. 206–211.

PROFIBUS. *General purpose field communication system, vol. 2/3 (Profibus)*. 1996. CENELEC, Std. EN 50170.

QI, X.; PARMER, G.; WEST, R. An efficient end-host architecture for cluster communication services. In: *Proceedings of the IEEE International Conference on Cluster Computing (Cluster '04)*. [S.l.: s.n.], 2004.

RAYNAL, M. *Algorithms for Mutual Exclusion*. Massachusetts, Cambridge: MIT Press, 1986.

REGNIER, P. *Especificação formal, Verificação e Implementação de um protocolo de comunicação determinista, baseado em Ethernet*. Dissertação (Mestrado) — Universidade Federal da Bahia, 2008.

REGNIER, P.; LIMA, G. Deterministic integration of hard and soft real-time communication over shared-ethernet. In: *Proc. of Workshop of Tempo Real*. Curitíba, BR: [s.n.], 2006.

ROSTEDT, S.; HART, D. V. Internals of the rt patch. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2007. p. 161–172.

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.

SANTOS, M. *Por uma outra globalização - do pensamento único à consciência universal*. São Paulo: Record, 2000.

SCHLICHTING, R. D.; SCHNEIDER, F. B. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, v. 1, n. 3, p. 222–238, 1983.

SCHNEIDER, S. *Making Ethernet work in real time*. 2000.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. Priority Inheritance Protocols: An approach to real-time synchronisation. *IEEE Transaction on Computers*, v. 39, n. 9, p. 1175–1185, 1990.

SINNOTT, R. O. The formal, tool supported development of real time systems. In: *Proc. of the 2nd Int. Conf. on Software Engineering and Formal Methods, 2004*. [S.l.]: IEEE Computer Society Press, 2004. p. 388–395.

SIRO, A.; EMDE, C.; MCGUIRE, N. Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system. In: *Proceedings of the 9th Real-Time Linux Workshop*. [S.l.: s.n.], 2007.

SOBRINHO, J.; KRISHNAKUMAR, A. S. EQuB - Ethernet Quality of service using black Bursts. In: *Proceedings of 23rd Annual IEEE International Conference on Local Computer Networks (LCN'98)*. [S.l.]: IEEE Computer Society Press, 1998. p. 286.

SRINIVASAN, B. et al. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In: *Proc. of the Real-Time Technology and Applications Symposium*. [S.l.: s.n.], 1998.

STODOLSKY, D.; CHEN, J.; BERSHAD, B. Fast interrupt priority management in operating systems. In: *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*. [S.l.: s.n.], 1993. p. 105–110.

TANENBAUM, A. *Modern Operating Systems*. [S.l.]: Prentice-Hall, 2001. ISBN 85-241-0643-3.

TINDELL, K.; BURNS, A.; WELLINGS, A. J. An extendible approach for analysing fixed priority hard real-time tasks". *Real-Time Systems*, v. 4, n. 2, p. 145–165, 1994.

TLA+. 2008. Http://www.lamport.org/ - Last access jan. 08.

TRIPAKIS, S.; COURCOUBETIS, C. Extending promela and spin for real time. In: *Proc. of Tools and Algorithms for Construction and Analysis of Systems*. [S.l.]: Springer Verlag, 1996. (LNCS, v. 1055), p. 329–348.

V. YODAIKEN et al. *RT-Linux*. 2008. http://www.rtlinuxfree.com - Last access jan. 08.

VENKATRAMI, C.; CHIUEH, T. Supporting real-time traffic on Ethernet. In: *15th IEEE International Real-Time Systems Symposium (RTSS'94)*. [S.l.: s.n.], 1994. p. 282–286.

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

WANG, J.; KESHAV, S. Efficient and accurate Ethernet simulation. In: *Proc. Conf. Local Computer Networks*. [S.l.: s.n.], 1999. p. 182–191.

WANG, Z. et al. Real-time characteristics of Ethernet and its improvement. In: *Proceeding of the 4th World Congress on Intelligent Control and Automation*. Shanghai (P.R. China): IEEE Computer Society Press, 2002.

YAGHMOUR, K. The real-time application interface. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2001.

YU, Y.; MANOLIOS, P.; LAMPORT, L. Model checking tla+ specifications. In: PIERRE, I. L.; KROPF, T. (Ed.). *Correct Hardware Design and Verification Methods*. Berlin, Heidelberg, New York: Springer-Verlag, 1999. (Lecture Notes in Computer Science, v. 1703), p. 54–66.