UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

ARQUITECTURAS DE SUPORTE AO DESENVOLVIMENTO DE PROTOCOLOS DE COMUNICAÇÃO DE TEMPO-REAL

João Carlos Teixeira Rodrigues

DOUTORAMENTO EM INFORMÁTICA

Outubro de 2005

ARQUITECTURAS DE SUPORTE AO DESENVOLVIMENTO DE PROTOCOLOS DE COMUNICAÇÃO DE TEMPO-REAL

João Carlos Teixeira Rodrigues

Tese submetida para obtenção do grau de

DOUTOR EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Professor Doutor Luís Eduardo Teixeira Rodrigues

Outubro de 2005

Resumo

A crescente complexidade das aplicações distribuídas de tempo-real leva a que estas necessitem de serviços de comunicação cada vez mais sofisticados e diversificados. Enquanto as concretizações monolitícas de protocolos de comunicação são difíceis de expandir, aperfeiçoar e ajustar às necessidads de cada aplicação, as concretizações baseadas na composição de camadas modulares são mais facilmente configuráveis e, por isso, mais atraentes para sistemas de tempo-real, onde não interessa gastar recursos com funcionalidades desnecessárias.

A dissertação estuda a construção de molduras de suporte à concepção, composição e execução de sistemas de comunicação modulares para aplicações de tempo-real. Um aspecto ao qual se dá particular ênfase é a capacidade de validar a correcção da composição de protocolos no domínio do tempo. A moldura proposta inclui os seguintes componentes: *i)* Um modelo de suporte à composição e desenvolvimento de protocolos de comunicação que facilita a posterior análise temporal da composição; *ii)* Uma ferramenta de análise temporal de composição de protocolos; *iii)* Uma ferramenta que automatiza o processo de alocação e atribuição de prioridades; *iv)* Um protótipo de um ambiente de execução de composições num sistema concreto.

Uma das principais contribuições deste trabalho consiste em demonstrar que é possível, através do uso de uma moldura de composição adequada, extrair informação relevante para a análise temporal a partir do código sem recorrer a linguagens de programação especializadas, facilitando desta maneira a análise e desenvolvimento do sistema. Um aspecto relevante da aproximção proposta consiste no facto da moldura usar um conjunto integrado de mecanismos que permite, simultaneamente, simplificar a tarefa de cálcular o pior tempo de resposta das composições de protocolos e a optimização e depuramento da concretização resultante.

PALAVRAS-CHAVE: Composição de protocolos, Tempo-real, Análise de escalonabilidade

Abstract

The growing complexity of distributed real-time applications creates the need for more sophisticated and diverse communication protocols. While monolithic implementation of communication protocols are hard to expand and to tune according to the specific needs of each application, modular implementations, based on the composition of micro-protocols can be easily adapted. Therefore, modular implementations are more appealing for real-time applications, where it is important not to consume critical resources when functionalities that are not strictly required.

The dissertation addresses the construction of protocol design, composition and execution frameworks for real-time applications. Emphasis is given to the validation of the correctness of the protocol in the time domain. The proposed framework includes the following components: *i)* A model for the design and composition of protocols that simplifies the timing analysis; *ii)* A timing analysis tool tailored for protocol compositions; *iii)* A tool that automates priority assignment and task allocation; *iv)* A prototype of a concrete protocol execution environment.

One of the main contributions of this work consists in showing that it is possible, by using the appropriate framework, to extract from the protocol code information that is relevant for the timing analysis without enforcing the use of specialized programming language. Furthermore, the framework uses a set of integrated mechanisms that are useful for both the protocol analysis and during the protocol execution.

KEY-WORDS: Protocol composition, Real-Time, Schedulability Analysis

Agradecimentos

Os meus agradecimentos vão em primeiro lugar para o Professor Doutor Luís Rodrigues, cuja orientação e ajuda foram fundamentais na realização deste trabalho.

Agradeço também ao Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa, pelos meios disponibilizados, e em particular ao Doutor António Casimiro. Um agradecimento ao Hugo Miranda, ao Alexandre Pinto, ao João Ventura e também a todos os que reviram o artigo publicado na conferência WFCS2004.

Gostaria também de deixar um agradecimento ao Instituto Nacional de Engenharia, Tecnologia e Inovação pelo tempo que me foi concedido e pelos meios disponibilizados e a todos os colegas do Departamento de Electrónica e em particular ao seu Director o Investigador António Miguel de Campos. Um agradecimento especial à falecida Eng. Margarida Machado, pelo apoio que me deu para iniciar este trabalho, ao Doutor Carvalho Rodrigues, aos meus pais, Álvaro e Maria Arminda Rodrigues e à minha esposa Maria José Leal.

Lisboa, Abril de 2005

João Carlos Teixeira Rodrigues

Para a Zita

Índice

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

A crescente complexidade das aplicações distribuídas para tempo-real leva a que estas necessitem de serviços de comunicação cada vez mais sofisticados e diversificados. Por exemplo, muitas aplicações de tempo-real com requisitos de tolerância a faltas necessitam de serviços de comunicação que asseguram diferentes propriedades de coerência tais como a ordem total ou o acordo distribuído (Rodrigues & Veríssimo, 1991) (Cristian & Dehn, 1990) (Kopetz & Grunsteidl, 1994) (Abdelzaher *et al.*, 1996).

Por sua vez, a necessidade de fornecer às aplicações um conjunto diversificado de propriedades exige uma maior flexibilidade por parte dos sistemas de comunicação. Neste contexto, as implementações monolitícas de protocolos de comunicação têm algumas desvantagens: são difíceis de expandir ou aperfeiçoar e podem introduzir uma sobrecarga adicional em tempo de execução devido à necessidade de suportar funcionalidades que não são necessariamente requeridas pela aplicação. Para além disso, a análise do atraso da comunicação para o pior caso das implementações monolitícas pode ser difícil de avaliar.

Em contrapartida, a construção de serviços de comunicação através da composição de camadas modulares de micro-protocolos traz múltiplas vantagens, nomeadamente:

- *Configuração*. Um serviço de comunicação é construído através de um conjunto de micro-protocolos, onde cada micro-protocolo concretiza uma propriedade semântica ou funcionalidade específica.

- *Reutilização*. Alguns micro-protocolos podem ser reutilizados em vários serviços.
- *Eficiência*. A personalização dos serviços de comunicação permitem que um serviço possua somente as funcionalidades requeridas por uma aplicação concreta e evitam a sobrecarga que possa resultar da inclusão de funcionalidades desnecessárias.
- *Análise Temporal*. A utilização de serviços "à medida" pode simplificar a análise temporal de pilhas de protocolos, em particular o cálculo do tempo de resposta para o pior caso na troca de mensagens, visto que os protocolos a analisar são mais simples.

Sendo assim, é relevante estudar e construir molduras de suporte à concepção, composição e execução de sistemas de comunicação modulares para aplicações de tempo-real. A dissertação aborda precisamente este tema.

1.1 Contexto

1.1.1 Molduras de suporte à composição e execução de protocolos

Uma moldura de suporte à composição e execução de protocolos de comunicação é um pacote de *software* que fornece um conjunto de mecanismos e serviços que facilitam o desenvolvimento de sistemas de comunicação modulares, fiáveis e eficientes. Fornece um modelo de composição flexível baseado num conjunto de abstracções que permitem ao programador construir pilhas de protocolos de forma modular. Fornece também um modelo de execução com um conjunto de mecanismos que minimizam a dependência em relação ao sistema operativo subjacente.

Um dos objectivos destas molduras é simplificar a tarefa do programador, através da inclusão de vários mecanismos e serviços complementares que respondem a difer-

entes necessidades que emergem no desenvolvimento de protocolos, incluindo o projecto, a análise, a concretização e a execução de pilhas de protocolos, nomeadamente:

- Na fase de projecto, a moldura deve favorecer a reutilização de código, permitindo a criação de pilhas de protocolos complexas através da composição de micro-protocolos. Este aspecto é particularmente relevante no contexto dos sistemas de tempo-real embebidos, favorecendo a utilização dos recursos computacionais estritamente necessários à execução do serviço pretendido.

- No contexto dos sistemas de tempo-real, a moldura deve fornecer um suporte adequado para avaliar o comportamento temporal da composição de protocolos.

- Ao nível da concretização, a moldura deve oferecer uma biblioteca exportando um número de funções necessárias ao programador, tais como gestão de tampões para mensagens de dados (incluindo funções para adicionar e remover cabeçalhos das mensagens), gestão de temporizadores, gestão de tarefas, etc.

- Ao nível do desenvolvimento, de código, a moldura deve concretizar os serviços básicos que suportam a execução de módulos de protocolos, a comunicação e sincronismo entre esses módulos, etc. Uma moldura de composição e execução de pilhas de protocolos de comunicação para tempo-real deve também suportar a reserva de recursos requeridos por um canal de comunicação de modo a assegurar a disponibilidade dos recursos em tempo de execução.

Diversas molduras para a composição e execução de protocolos têm sido desenvolvidas para sistemas sem constrangimentos de tempo-real, tais como o x-Kernel (Hutchinson & Peterson, 1991), o Coyote (Bhatti *et al.*, 1998), o Ensemble (Hayden, 1998), e o *Appia* (Miranda *et al.*, 2001) assim como para ambientes de tempo-real, como o CORDS (Travostino *et al.*, 1996) ou o Cactus (Hiltunen *et al.*, 1996).

1.1.2 Problemas específicos de tempo-real

Os sistemas computacionais que interagem e evoluem em resposta a estímulos do ambiente físico em que se inserem são designados sistemas de tempo-real. Um dos aspectos fundamentais de um sistema de *tempo-real* é a noção de tempo. Estes sistemas devem evoluir e reagir a eventos externos produzidos pelo exterior com constrangimentos temporais precisos. Por consequência, o funcionamento correcto destes sistemas depende não só do resultado lógico da computação mas também do *tempo* limite (ou *meta*) em que o resultado é produzido. Uma reacção fora de tempo pode ser inútil ou até perigosa.

Em sistemas de tempo-real *estrito*, as metas devem ser escrupulosamente cumpridas. Caso contrário, considera-se que ocorreu uma falta grave no sistema (que pode resultar numa catástrofe, devido a razões económicas, ou mesmo à perda de vidas humanas). Os sistemas de tempo-real onde a violação de uma meta pode representar uma degradação do seu funcionamento mas não resulta numa falha do sistema, designam-se por sistemas de tempo-real *lato*. O domínio das aplicações de tempo-real é vasto abrangendo sistemas aeroespaciais, sistemas de defesa, a automatização industrial, a instrumentação, o controlo do tráfego aéreo, a indústria automóvel e as telecomunicações.

Uma tarefa de tempo-real é uma entidade executável que é executada pelo processador de modo sequencial, sendo caracterizada por um tempo de execução para o pior caso e por uma meta. Uma tarefa pode ser periódica (sendo activada regularmente com um período fixo), aperiódica (sendo activada irregularmente sem um período conhecido) ou esporádica (sendo activada irregularmente mas com um período mínimo conhecido entre duas activações sucessivas).

O conjunto de regras que, a qualquer momento, determina a ordem pela qual um conjunto de tarefas concorrentes são executadas pelo processador é designado por algoritmo de escalonamento. Existem diversas classes de algoritmos de escalonamento.

No entanto, neste trabalho serão utilizados algoritmos preemptivos de prioridades fixas que ordenam um conjunto de tarefas por ordem decrescente das suas prioridades. Neste contexto, preemptivo significa que a execução de uma tarefa pode ser interrompida a qualquer momento para atribuir ao processador, outra tarefa de maior prioridade. As prioridades das tarefas são atribuídas *à priori* e mantêm-se fixas durante o funcionamento do sistema.

Na concepção de um sistema de tempo-real estrito, um dos objectivos fundamentais, é assegurar que este se comporta de um modo previsível no que respeita aos seus requisitos temporais. Isto significa que deve ser possível mostrar, demonstrar ou provar que os requisitos temporais são cumpridos durante o tempo de vida do sistema (Stankovic & Ramamritham, 1990). A análise de escalonabilidade permite predizer se os constrangimentos temporais das tarefas de uma aplicação serão ou não cumpridos durante o tempo de execução, devendo ser realizada à priori, independentemente da classe do algoritmo de escalonamento utilizado.

O ambiente de execução deve garantir que a execução das tarefas corresponde ao comportamento deterministico previsto na análise de escalonabilidade. Alguns dos factores que afectam a previsibilidade de um sistema de tempo-real estão relacionados com as características arquitecturais do hardware e com os mecanismos adoptados pelos núcleos de muitos sistemas operativos comerciais. Por exemplo, a utilização de memória *cache*, cuja motivação é baseada em observações estatísticas, dificulta uma análise precisa para o pior-caso nas arquitecturas correntes. As técnicas de gestão de memória utilizadas pelo núcleo dos sistemas operativos podem também provocar atrasos imprevisíveis causados por faltas e substituições de páginas. O número de interrupções geradas pelos periféricos de entrada/saída pode ser difícil de analisar à priori, podendo causar atrasos imprevisíveis na execução de qualquer tarefa. Outro problema relacionado com as interrupções é o facto de qualquer rotina de atendimento de uma interrupção poder interromper a execução de qualquer tarefa, independentemente da sua urgência. Na tentativa de minimizar estes efeitos e de permitir um maior

controlo sobre a sua previsibilidade, alguns sistemas operativos optam por reduzir ao mínimo o processamento dentro das rotinas de atendimento de interrupções, ficando este a cargo de tarefas dedicadas à gestão de dispositivos e escalonadas pelo sistema operativo.

A previsibilidade do funcionamento do sistema também depende do modo como as primitivas do sistema operativo estão implementadas. Todas as funções do sistema operativo deveriam ser caracterizadas por um tempo de execução para o pior caso conhecido e limitado.

Outra causa de não determinismo está relacionado com o fenómeno de inversão de prioridade, que ocorre quando a execução de uma tarefa de alta prioridade é bloqueada, por um intervalo de tempo ilimitado, pela execução de uma tarefa de prioridade mais baixa. São exemplo disso as primitivas de sincronização típicas utilizadas em sistemas operativos tradicionais, tais como os semáforos. Hoje em dia o problema da exclusão mútua já é tratado em alguns sistemas operativos utilizando um protocolo de herança de prioridade (Sha *et al.*, 1990) como o Windows CE ou o núcleo ETS[1].

A ausência de sistemas operativos comerciais capazes de suportar de modo eficiente conjuntos de tarefas com constrangimentos temporais estritos, deu origem a um esforço considerável de investigação em novos paradigmas computacionais e novas estratégias, que resultaram numa nova geração de sistemas operativos para tempo-real estrito. Alguns exemplos de sistemas operativos que foram desenvolvidos de acordo com estes princípios são o CHAOS (Gheith & Schwan, 1993), MARS (Kopetz & al., 1989), Spring (Stankovic & Ramamritham, 1991), ARTS (Tokuda & Mercer, 1989a), MARUTI (O. Gudmundsson & Tripathy, 1990) e o HARTS (Shin, 1991).

[1]O núcleo ETS é propriedade da Phar Lap Software, Inc.

1.2 Objectivos

Um dos requisitos fundamentais na concepção de um sistema de tempo-real distribuído é a avaliação dos atrasos de comunicação associados à troca de mensagens entre os processos que participam na computação. Mais precisamente, é necessário avaliar o atraso na comunicação ponto-a-ponto para o pior caso. Com a crescente complexidade dos serviços de comunicação, a verificação dos constrangimentos temporais entre a tarefa emissora e a(s) tarefa(s) receptora(s) torna-se extremamente difícil (se não impossível) sem o suporte de ferramentas e metodologias que suportem a análise de escalonabilidade.

O objectivo deste trabalho é desenvolver uma moldura de suporte à composição e execução de protocolos de comunicação para tempo-real que facilite a tarefa de validar a correcção da composição de protocolos no domínio do tempo.

1.3 Resultados e Contribuições

Esta dissertação apresenta uma moldura de suporte à concepção, composição e execução de sistemas de comunicação modulares para aplicações de tempo-real que designámos por *RT-Appia*. Esta moldura enquadra um conjunto de componentes que pretende facilitar o desenvolvimento de sistemas de comunicação modulares para ambiente de tempo-real, nomeadamente:

- Um modelo de suporte à composição e desenvolvimento de protocolos de comunicação que facilita a posterior análise temporal da composição;
- Uma ferramenta de análise temporal de composição de protocolos, obtida através da adaptação e extensão das ferramentas desenvolvidas por Tindell (1994);
- Uma ferramenta que automatiza o processo de atribuição e optimização de prioridades;

- Um protótipo de um ambiente de execução de composições de protocolos, integrado com as ferramentas anteriormente enumeradas, e completamente funcional num sistema concreto.

A versão do *RT-Appia* descrita nesta dissertação foi codificada em Visual C++. O protótipo liga-se com o sistema operativo WindowsNT/Windows 2000 ou com o núcleo de tempo-real ETS, podendo ser executado em sistemas embebidos usando processadores compatíveis com o x86 de 32 dígitos.

Uma das principais contribuições deste trabalho consiste em demonstrar, através do uso de uma moldura adequada, que é possível extrair informação relevante para a análise temporal a partir do código sem recorrer a linguagens de programação especializadas, facilitando deste modo a análise e desenvolvimento do sistema. Um aspecto relevante e inovador da aproximação proposta consiste no facto da moldura usar um conjunto integrado de mecanismos que permite, simultaneamente, simplificar a tarefa de calcular o pior tempo de resposta das composições de protocolos e simplificar a optimização e depuramento da concretização resultante.

Parte destes resultados foram divulgados nas seguintes publicações:

- J. Rodrigues, H. Miranda, J. Ventura e L. Rodrigues. "The design of RTAppia", *in Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-Time Dependable Systems*, pp. 261-268, Rome, 8-10 January 2001.

- J. Ventura, J. Rodrigues e L. Rodrigues. "Response Time Analysis of Composable Micro-Protocols", *in Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May, 2 - 4, 2001, pp 335-342.

- J. Rodrigues, J. Ventura e L. Rodrigues. "Schedulability Analysis of an Event-based Real-Time Protocol Framework", *in Proceedings of the Seventh IEEE Inter-*

national Workshop on Object-oriented Real-time Dependable Systems, (WORDS 2002), pp. 319-325, San Diego, CA, USA, January 2002.

- J. Rodrigues e L. Rodrigues. "From running code to event-graphs: a pragmatic approach to derive WCRT of protocol compositions", *in Proceedings of the 5th IEEE International Workshop on Factory Communication Systems*, pp 265-274, Vienna, Austria, September 2004.

1.4 Estrutura da Dissertação

Esta dissertação está organizada do seguinte modo: no Capítulo 2 é apresentada uma panorâmica sobre molduras de composição e execução de protocolos de comunicação e no Capítulo 3 uma panorâmica sobre técnicas de análise de escalonabilidade. No Capítulo 4 irá ser descrita a moldura de composição *RT-Appia*. No Capítulo 5 é descrita uma técnica de análise de escalonabilidade para a composição de protocolos. No Capítulo 6 é descrito um algoritmo heurístico para a alocação e optimização de prioridades para um conjunto de canais de tempo-real. O modelo de execução e a avaliação da moldura protótipo são descritos no Capítulo 7. Finalmente são apresentadas as conclusões no Capítulo 8.

2

Moldulas de suporte à composição e execução de protocolos

A concretização de serviços de comunicação através da composição de camadas modulares de micro-protocolos tem sido uma aproximação seguida com sucesso por alguns sistemas de comunicação de grupo como por exemplo o Consul (Mishra *et al.*, 1993), o Horus (van Renesse *et al.*, 1996), o ARMADA (Abdelzaher *et al.*, 1997), o Coyote (Bhatti *et al.*, 1998), o Ensemble (Hayden, 1998), entre outros. Esta aproximação favorece a reutilização de micro-protocolos e permite às aplicações configurar pilhas de protocolos que satisfaçam precisamente as suas necessidades.

O x-Kernel (Hutchinson & Peterson, 1991) é um trabalho pioneiro e influente na área das molduras de composição de protocolos. Este sistema introduziu um modelo de composição simples definindo um conjunto uniforme de abstracções para encapsular protocolos e estruturar pilhas de protocolos de forma hierárquica. No x-Kernel a interacção entre camadas de protocolos adjacentes é concretizada através da chamada de funções, executadas no contexto da tarefa da mensagem. Posteriormente ao x-Kernel outras molduras de protocolos foram desenvolvidas, com destaque para o Ensemble (Hayden, 1998), Coyote (Bhatti *et al.*, 1998) e o Appia (Miranda *et al.*, 2001) que oferecem uma estrutura mais flexível, propondo uma interacção entre camadas baseada na troca de eventos.

No entanto, nenhum dos sistemas anteriores foi projectado para suportar

aplicações de tempo-real, pelo que alguns deles foram posteriormente estendidos e adaptados para ambientes de tempo-real. São exemplos disso o CORDS (Travostino *et al.*, 1996), que estende o x-Kernel para incluir mecanismos de reserva de recursos e o Cactus (Hiltunen *et al.*, 1996) que estende a moldura Coyote para suportar a construção de serviços de tempo-real configuráveis.

Nas secções seguintes é feita uma breve panorâmica sobre as molduras de suporte mais relevantes para o desenvolvimento e execução de composições de protocolos e que influenciaram este trabalho.

2.1 x-Kernel

O x-Kernel (Hutchinson & Peterson, 1991) introduziu um modelo no qual as pilhas de protocolos são estruturadas de forma hierárquica, formando grafos de protocolos. Este modelo, onde os protocolos comunicam através da troca de mensagens, baseia-se em três classes distintas de objectos: *protocolos*, *sessões* e *mensagens*.

Os objectos da classe *protocolo* representam protocolos convencionais. Estes objectos executam duas funções fundamentais: criam objectos *sessão* e realizam a desmultiplexagem de mensagens no sentido ascendente. Os objectos da classe *sessão*, que podem ser vistos como instâncias de protocolos, são criados dinamicamente e fornecem as operações para enviar e receber mensagens. Os objectos da classe *mensagem* visitam as sessões através das operações push e pop fornecidas pelos objectos sessão. A operação push é invocada para passar mensagens no sentido descendente. No sentido ascendente, uma mensagem visita alternadamente um objecto protocolo através da operação demux seguido de um objecto sessão através da operação pop. A operação demux de um protocolo decide para que sessão a mensagem deve ser encaminhada. Na Figura 2.1 estão representadas estas relações. A Figura 2.1a) ilustra um grafo de protocolos e a Figura 2.1b) ilustra a utilização de push para guiar uma mensagem no

sentido descendente da pilha e demux e pop para guiar uma mensagem no sentido ascendente da pilha.

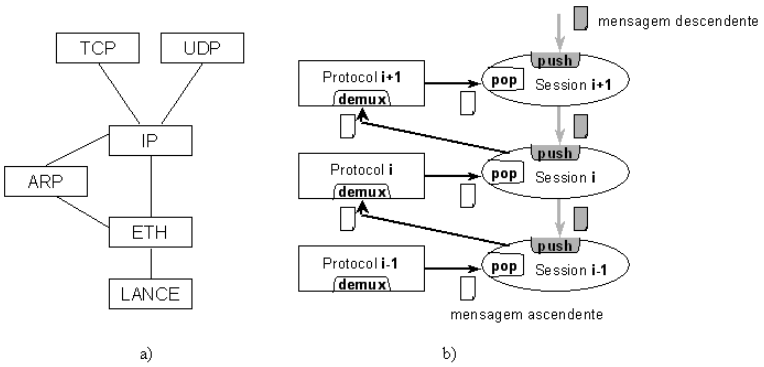

Figura 2.1: x-kernel

Grosso modo, os objectos protocolo fornecem operações para abrir canais, de que resulta a criação de um objecto sessão, e os objectos sessão fornecem operações para enviar e receber mensagens. A reutilização de protocolos é promovida recorrendo a uma interface entre protocolos uniforme, nomeadamente através da utilização das funções push e pop.

Em tempo de execução, o x-Kernel suporta um modelo designado por "tarefa por mensagem", segundo o qual, como o próprio nome indica, é atribuída uma tarefa para o processamento de cada mensagem. Deste modo, a interacção entre camadas de protocolos adjacentes é concretizada através da chamada de funções, executadas no contexto da tarefa atribuída à mensagem. O x-Kernel fornece ainda um conjunto de rotinas de suporte à implementação de protocolos, como a gestão de tampões (utilizadas para manipular o conteúdo e cabeçalho das mensagens) e a gestão de temporizadores.

O x-Kernel ilustra a flexibilidade das arquitecturas modulares baseadas na composição de protocolos e na construção de serviços de comunicação complexos em sistemas distribuídos, face às implementações monolíticas. Em (O'Malley & Peterson, 1992) os autores exploram a possibilidade de decompor os protocolos do x-Kernel, em diversos micro-protocolos, onde cada micro-protocolo implementa uma das fun-

cionalidades existentes no protocolo original. As ramificações condicionais introduzidas deste modo no diagrama de protocolos hierárquico do x-Kernel são suportadas por *protocolos virtuais*.

Sobre o x-Kernel foram criados sistemas para ambientes específicos, como o subsistema de comunicação Consul (Mishra *et al.*, 1993), orientado para a construção de aplicações distribuídas tolerantes a faltas. O trabalho com o Consul revelou algumas limitações do modelo adoptado pelo x-Kernel, nomeadamente a falta da flexibilidade necessária para concretizar certos tipos de protocolos como, por exemplo, protocolos de difusão atómica. Os autores do Consul mostraram também como o modelo de composição do x-kernel leva a dependências implícitas entre diferentes protocolos, o que de certo modo contraria algumas das vantagens das molduras de composição, como a reutilização e a configuração.

2.2 CORDS

O CORDS (Travostino *et al.*, 1996) é um subsistema de comunicação orientado para objectos, de suporte a aplicações distribuídas de tempo-real. Este sistema foi desenvolvido como uma extensão do sistema x-Kernel, tendo sido adicionados mecanismos orientados para a execução em tempo-real. Um destes mecanismos consiste na reserva de recursos, como a alocação de memória, a capacidade de seleccionar o número de tarefas e os seus atributos de escalonamento e a largura de banda da rede de comunicações. A "unidade"de reserva de recursos é designada um Path (Figura 2.2).

O CORDS segue o modelo de uma tarefa por mensagem do x-Kernel mas permite ao utilizador seleccionar o número máximo de tarefas disponíveis para processamento de mensagens, assim como as suas prioridades e a política de escalonamento usada num Path. Em tempo de execução, o CORDS é suportado por uma versão de tempo-real do núcleo Mach (o OSF MK) (Tokuda *et al.*, 1990). O escalonador in-

tegrado ITDS (Tokuda & Mercer, 1989b) deste núcleo permite ao utilizador definir várias políticas de escalonamento para um conjunto de processadores, utiliza um protocolo de herança de prioridade nas primitivas de sincronização de tarefas e fornece um mecanismo de gestão de memória, para evitar atrasos descontrolados na falta de páginas.

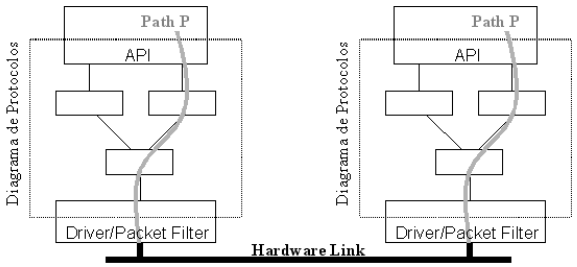

Figura 2.2: Um Path.

2.3 Coyote e Cactus

O Coyote (Bhatti *et al.*, 1998) é uma moldura que aumenta a flexibilidade do x-Kernel, permitindo que um protocolo (também designada por *protocolo composto*), possa ser decomposto num conjunto de micro-protocolos. Um micro-protocolo é uma secção de código que implementa uma propriedade bem definida ou fornece uma funcionalidade específica e é estruturado como um conjunto de *gestores de eventos*[1] que são executados quando o evento para o qual estão registadas ocorre. Os micro-protocolos comunicam entre si principalmente através da troca de eventos mas podem também partilhar memória. A Figura 2.3 representa um protocolo composto.

Por sua vez, o Cactus tenta preservar a flexibilidade adicional do sistema Coyote e ao mesmo tempo fornecer um suporte para aplicações de tempo-real (Hiltunen *et al.*, 1996). O Cactus é construído sobre o sistema CORDS e cada *protocolo composto* é implementado como um protocolo no grafo de protocolos do CORDS. O Cactus suporta

[1] Do Inglês, "*event handlers*"

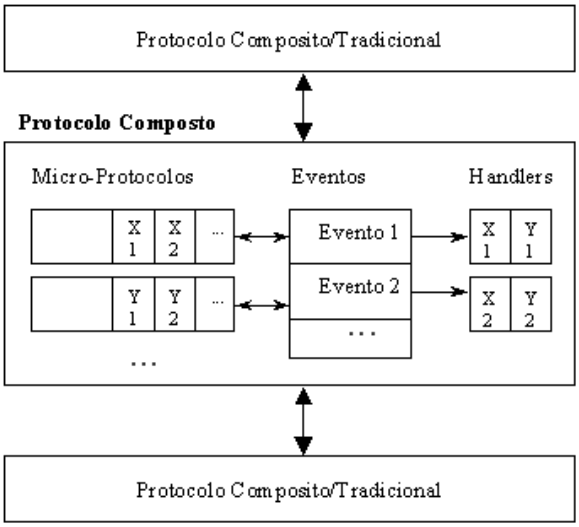

Figura 2.3: Protocolo Composto.

a implementação de canais de tempo-real utilizando um protocolo composto e usa os recursos de sistema alocados com os Paths do CORDS.

Quando um determinado evento ocorre, todos os gestores associados são colocados numa lista de execução, ordenada por um atributo de ordem, e são executados sequencialmente por uma tarefa de despacho. A execução das funções é atómica, isto é, cada função é executada sem interrupção. A ordem de execução das funções de atendimento é determinada pela urgência relativa do respectivo evento.

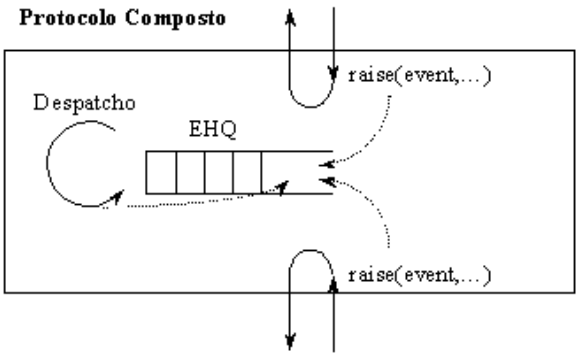

Figura 2.4: Modelo de execução para um Protocolo Composto.

A Figura 2.4 ilustra os mecanismos de execução num protocolo composto. As se-

tas a cheio representam tarefas de execução, incluindo a tarefa de despacho associada à lista de funções de atendimento de eventos. As outras tarefas representam as tarefas de execução que invocam as funções de interface push e pop para os protocolos acima e abaixo, respectivamente. As linhas a tracejado representam as funções a serem inseridas na lista de funções como resultado da sinalização de um evento através da chamada da função raise no protocolo composto.

A alocação de canais de tempo-real está dividida num módulo de controlo de canal (CCM) e num serviço de controlo de admissão, o qual é composto por um módulo de controlo de admissão (ACM) por máquina. O CCM traduz os argumentos do canal em requisitos de recursos genéricos do sistema, como um conjunto de micro-protocolos, um conjunto de Paths e ciclos CPU. O ACM mantém a informação sobre os recursos disponíveis e, baseado nesta informação, aceita ou rejeita o pedido. Para além de determinar a disponibilidade de memória e de largura de banda, este módulo calcula também o atraso das mensagens no canal, para o pior caso, utilizando um método de análise de escalonabilidade para redes de comunicação com ligações ponto-a-ponto. Especificamente, cada ACM envolvido na criação do canal, tenta determinar uma prioridade a atribuir ao canal que garanta a sua escalonabilidade. A Figura 2.5 ilustra os componentes do controlo de admissão e a interacção entre componentes.

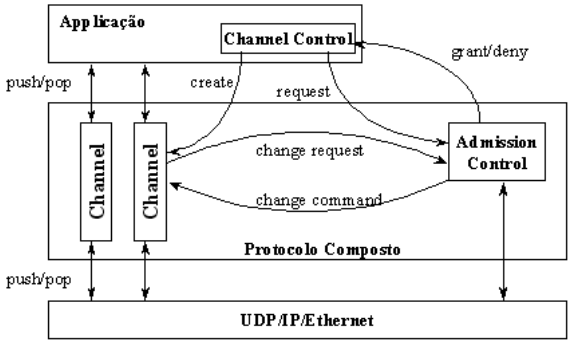

Figura 2.5: Componentes do controlo de admissão.

O caracter não preemptivo do sistema CORDS, pode causar inversões de prioridade num canal de tempo-real do Cactus, entre tarefas de diferentes prioridades. Em

(Das *et al.*, 1999) são descritos alguns factores que causam inversões de prioridade e comportamentos temporais não deterministicos em canais de tempo-real no sistema Cactus. Estes factores estão relacionados com o caracter não preemptivo do sistema CORDS. Um dos problemas, por exemplo, deve-se à transição de tarefas entre o espaço de núcleo, onde é executado o Cactus/CORDS, e o espaço do utilizador, onde reside a aplicação.

2.4 ARMADA

O ARMADA (Abdelzaher *et al.*, 1997) fornece um conjunto de serviços de comunicação *middleware* para o desenvolvimento de aplicações de tempo-real tolerantes a faltas, em sistemas embebidos. Estes serviços incluem um conjunto de serviços de comunicação de tempo-real (RTC) e um conjunto de serviços de comunicação de grupo (RTCAST). Os serviços RTC adoptam o modelo de canais de tempo-real unidireccionais com ligação, ponto-a-ponto, para garantir um atraso limitado na comunicação. Os serviços RTCAST incluem um serviço de difusão atómica com limite de tempo, um serviço de filiação de grupo e um serviço de controlo de admissão. Este último utiliza uma análise de escalonabilidade em tempo de execução para realizar o controlo de admissão de mensagens de difusão. Todos os serviços do ARMADA são concretizados sobre a plataforma CORDS suportada pelo sistema operativo Mach OSF MK7.2.

2.5 *Appia*

O *Appia* (Miranda *et al.*, 2001) é uma moldura de protocolos com um modelo de composição orientado para objectos e desenvolvida em linguagem Java, que fornece um conjunto de classes de suporte para o desenvolvimento e execução de pilhas de protocolos.

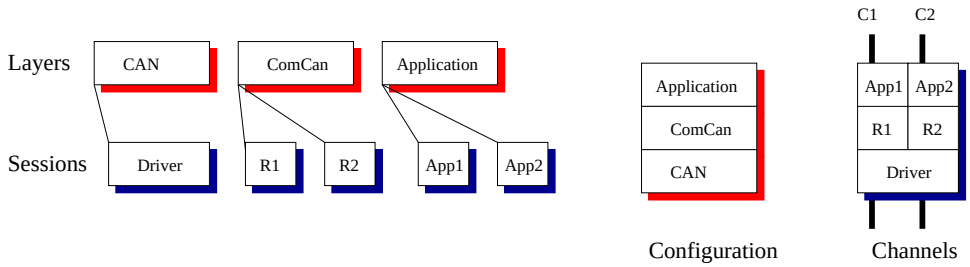

Figura 2.6: Relação entre *configuração*, *camadas* e *sessões* no *Appia*.

No *Appia* (Miranda *et al.*, 2001) cada pilha de protocolos é composta por um ou mais canais. Cada canal é uma sequência ordenada de *sessões* que são instâncias de uma *camada* específica de um protocolo. A sessão mantém o estado que é usado pela camada para processar eventos. Por exemplo, uma camada que concretiza um protocolo de ordem pode manter um número de sequência ou um vector de relógio como parte do estado da sessão. Em protocolos com ligação, a sessão também mantém a informação acerca dos pontos da ligação. A sequência de *camadas* associadas a um dado canal define a *configuração* concretizada pelo canal. Um aspecto importante de uma pilha de protocolos no *Appia* é o de diferentes canais poderem partilhar sessões em uma ou mais camadas. Este mecanismo permite suportar a coordenação entre canais independentes. A relação entre uma *configuração*, *camadas* e *sessões* no *Appia* são ilustradas na Figura 2.6.

A comunicação entre *camadas* é realizada através da troca de *eventos*. Os eventos são estruturas de dados orientadas para objectos, descendentes da classe base Event. Novos eventos podem ser criados derivando uma nova classe de outra previamente definida.

De modo a permitir a refinação futura de eventos, a verificação do tipo é sempre realizada sobre a classe base que satisfaz os requisitos desejados. O objectivo é o de suportar a especialização de eventos utilizando os mecanismos de herança. Deste modo, apesar do desconhecimento da definição de novos atributos, os protocolos legados continuarão a ser executados correctamente, favorecendo a sua reutilização, que,

como já foi referido, é uma propriedade importante nas molduras de composição.

Durante a definição da *configuração* de um canal, cada camada declara o conjunto de eventos que a camada produz e o conjunto de eventos que a camada está interessada em subscrever. Utilizando esta informação, o *Appia* constrói tabelas de encaminhamento de eventos com o conjunto exacto de sessões que necessitam ser visitadas por cada evento. Em tempo de execução o *Appia* utiliza esta informação, optimizando o tempo necessário para o evento atravessar o canal.

Associado a cada canal existe um escalonador de eventos (EventScheduler), que é um objecto passivo responsável por seleccionar o próximo evento a processar. É possível associar diferentes objectos EventScheduler a diferentes conjuntos de canais; no entanto, todos os objectos (canais e eventos) são executados por uma única tarefa, sem atributos de ordem.

2.6 Síntese e Discussão

O sistema CORDS herda as restrições do x-Kernel, limitando a comunicação de dados entre camadas adjacentes à chamada de funções. Os mecanismos de reserva de recursos fornecidos são um excelente suporte para o desenvolvimento de aplicações de tempo-real, no entanto o CORDS não fornece nenhum suporte adicional que permita uma análise de escalonabilidade mais precisa. Por exemplo, podem ocorrer inversões de prioridade em canais de tempo-real, devido ao caracter não preemptivo do CORDS, que resultam da necessidade das tarefas de aplicação executarem o código do núcleo. No entanto, não é fornecido nenhum suporte adicional que permita determinar com precisão estes atrasos.

O Cactus é o único sistema que contrabalança a flexibilidade e eficiência de micro-protocolos com os requisitos de previsibilidade das aplicações de tempo-real e ilustra algumas das vantagens em personalizar serviços de comunicação de tempo-real

através da composição de micro-protocolos de grão fino. A simplicidade das funções de atendimento de eventos pode facilitar a análise do tempo de execução para o pior caso dos serviços de comunicação. No entanto, o tempo de execução para o pior caso de um protocolo composto pode ser difícil de avaliar devido à necessidade de saber quais os eventos que serão sinalizados por cada gestor de eventos em em cada micro-protocolo. Esta informação pode ser fornecida explicitamente pelo utilizador; caso contrário, é necessária uma ferramenta adicional para a identificar. Em (Hiltunen *et al.*, 1999) é usada uma estimativa conservadora na qual são incluídos na computação todos os eventos sinalizados em cada função, mesmo se estes são sinalizados em ramos separados de instruções condicionais. Para além disso, no Cactus as garantias temporais do controlo de admissão são baseadas em modelos estatísticos normalmente utilizadas em redes de pacotes de dados com ligação ponto-a-ponto.

O sistema *Appia* fornece um modelo aberto estimulado por eventos, modular e flexível, herdando as vantagens já previamente demonstradas por outros sistemas. A moldura *Appia* foi implementada em linguagem Java. Uma das fortes vantagens desta linguagem é a de permitir a independência em relação à plataforma de execução, através da utilização de uma máquina virtual, uma característica publicitada como "write once run everywhere". No entanto, a versão actual do *Appia* não foi desenhada para explorar os mecanismos propostos pelas versões para tempo-real da linguagem Java, nomeadamente, da Real-Time Specification for Java (RTSJ) (Bollella & Gosling, 2000). Outra limitação da moldura *Appia* é a de não possuir um escalonamento preemptivo com prioridades fixas. Esta limitação não permite tirar partido da flexibilidade fornecida por este tipo de escalonamento, nomeadamente na utilização de técnicas de análise de escalonabilidade para o cálculo dos tempos de resposta para o pior caso de mensagens em pilhas de protocolos.

Apesar destas limitações, a moldura *Appia* fornece uma excelente plataforma para desenvolver pilhas de protocolos eficientes e especializadas, para serem usadas em aplicações de tempo-real:

- A especialização de objectos utilizando mecanismos de herança permite ao programador de protocolos definir o conjunto de eventos mais apropriado para uma determinada aplicação, promovendo a reutilização de protocolos já implementados.

- Uma separação clara entre o modelo de composição e o modelo de execução, facilita a integração de ferramentas de análise de escalonabilidade para calcular os tempos de resposta para o pior caso de pilhas de protocolos de comunicação, em tempo de compilação.

- Vai um pouco além dos outros sistemas, fornecendo mecanismos que simplificam a tarefa de expressar e implementar constrangimentos entre canais usados pela mesma aplicação, permitindo alargar o espectro de aplicações suportadas pelos outros sistemas.

Pensamos pois que se justifica uma extensão da moldura *Appia* para suportar protocolos de comunicação em aplicações de tempo-real. Para além das razões apontadas, nenhum dos sistemas descritos suporta a análise dos tempos de resposta para o pior caso de sistemas de comunicação de tempo-real estrito.

Mais recentemente (Mena & Nestmann, 2005) foi proposto um modelo *dirigido a cabeçalhos*[2] para molduras de composição de protocolos, em alternativa ao modelo estimulado por eventos. Neste modelo é utilizada uma etiqueta[3] nos cabeçalhos das mensagens que indicam qual o gestor de cada protocolo que deve processar cada mensagem, em alternativa à declaração de eventos utilizada nas molduras descritas neste capítulo. No entanto não se encontra desenvolvida nenhuma moldura utilizando o modelo proposto.

[2]Do inglês *header-driven*.
[3]Do inglês *tag name*.

3

Análise de escalonabilidade

Um dos trabalhos mais influentes para a teoria do escalonamento em sistemas preemptivos de tempo-real, periódicos e com prioridades fixas, foi publicado em 1973 por Liu e Layland (1973). Um dos conceitos fundamentais introduzidos por Liu e Layland é o conceito de *instante crítico* para uma tarefa i: o pior tempo de resposta da tarefa i ocorre quando todas as tarefas com maior prioridade do que i são activadas em simultâneo com a tarefa i. Foi demonstrado que se as tarefas cumprem as suas metas num instante crítico, as metas serão cumpridas durante o tempo de vida do sistema.

Com base no conceito de *instante crítico*, Liu e Layland estudam o comportamento do algoritmo de atribuição de prioridades, designado por *Rate Monotonic* (RM). De acordo com este algoritmo, as prioridades das tarefas são atribuídas numa proporção inversa ao comprimento do seu período (quanto menor o período da tarefa maior a sua prioridade). Liu e Layland demonstram que a atribuição de prioridades do algoritmo RM é a óptima de entre todas as atribuições de prioridades fixas, no sentido que nenhum outro algoritmo de prioridades fixas pode escalonar um conjunto de tarefas que não possa ser escalonável pelo RM. Liu e Layland (Liu & Layland, 1973) deduziram também um teste de escalonabilidade simples baseado na utilização total do CPU. Este teste garante um limite superior mínimo de $69,31\%$ na utilização total do CPU, para que um conjunto arbitrário de tarefas periódicas cumpra sempre as suas metas.

O teste de escalonabilidade do RM é caracterizado, quanto à sua cobertura, como *necessário mas não suficiente* (isto é, pessimista). Isto significa que um conjunto de tarefas

que falhe o teste de escalonabilidade pode contudo ser escalonável pelo RM. Um estudo realizado por Lehoczky et al. (Lehoczky *et al.*, 1989) demonstrou a capacidade de o algoritmo poder escalonar conjuntos aleatórios de tarefas com factores de utilização médios próximos de 88% e derivou um teste de escalonabilidade *necessário e suficiente* para o RM.

No entanto, os constrangimentos impostos pela análise RM são bastante restritivos: as tarefas têm uma meta relativa igual ao seu período, são independentes, ou seja não possuem relações de precedência, e não podem sofrer bloqueios; adicionalmente as prioridades têm que ser atribuídas de acordo com o RM.

Leung e Whitehead (Leung & Whitehead, 1982) propuseram uma extensão ao RM permitindo que as metas relativas das tarefas possam ser menores que o seu período. O algoritmo de atribuição de prioridades *Deadline Monotonic* (DM) atribui a cada tarefa uma prioridade inversamente proporcional à sua meta relativa (a tarefa com a menor meta relativa possui a maior prioridade). A atribuição de prioridades segundo o DM é considerada óptima. Em (Audsley *et al.*, 1991), Audsley et al. deduzem um teste de escalonabilidade *necessário e suficiente* para o DM, baseado no cálculo do *tempo de resposta para o pior caso* de uma tarefa.

3.1 Cálculo do Tempo de Resposta para o Pior Caso

O tempo de resposta para o pior caso, r_i, de uma tarefa i pode ser calculado usando a seguinte equação:

$$r_i = C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{r_i}{T_j} \rceil C_j \tag{3.1}$$

Onde C_i é o tempo de execução para o pior caso da tarefa i e $hp(i)$ o conjunto de tarefas de maior prioridade do que i. O termo somatório expressa a interferência

sofrida pela tarefa i devido ao conjunto de tarefas $hp(i)$, que são activadas com um período T_j e possuem um tempo de execução para o pior caso C_j. O tempo de resposta r_i é calculado no instante crítico. Assume-se que todas as tarefas têm uma prioridade fixa única, um desfasamento nulo e que as metas não excedem o seu período. O tempo de bloqueio B_i, que corresponde ao maior tempo que uma tarefa de menor prioridade pode impedir a execução da tarefa i, será tratado com mais detalhe na Subsecção 3.2.

Este método de análise, publicado independentemente por Harter (Harter, 1987) (Harter, 1984), Joseph e Pandya (Joseph & Pandya, 1986) e (Audsley *et al.*, 1991) permite determinar se um conjunto de tarefas é escalonável, bastando comparar o tempo de resposta calculado para cada tarefa com a sua meta. Esta técnica é mais flexível que as anteriores pois permite, por um lado, analisar qualquer algoritmo de prioridades fixas (RM, DM ou outro) e, por outro lado, reduzir o pessimismo do teste de escalonabilidade baseado na utilização do CPU.

A ocorrência de r_i em ambos os lados da equação 3.1 faz com que o seu cálculo não seja trivial, sendo necessário fazê-lo por iterações sucessivas através da seguinte equação modificada:

$$r_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \lceil \frac{r_i^n}{T_j} \rceil C_j \tag{3.2}$$

Se a utilização ($U_j = \frac{C_j}{T_j}$) dos i níveis de prioridade mais elevados for menor que 1, esta equação converge garantidamente (isto porque as sucessivas aproximações a r_i são monotonicamente crescentes). As iterações começam com $r_i^0 = C_i$ como valor inicial e terminam quando $r_i^{n+1} = r_i^n$ ou $r_i^{n+1} > D_i$.

Quando a meta é menor ou igual ao período, só é necessário considerar uma única activação de cada tarefa. No entanto, quando a meta é maior do que o período, é possível que uma tarefa possa ser reactivada sem que as invocações anteriores tenham terminado (a sua execução será atrasada até que todas as invocações anteriores tenham

sido executadas).

Para analisar estas situações Lehoczky (Lehoczky *et al.*, 1990) estendeu a análise dada pela equação 3.1, para suportar tarefas com metas arbitrárias, utilizando o conceito de *períodos ocupados*: um *período ocupado de nível* i é definido como sendo o máximo intervalo de tempo em que um processador está ocupado a executar tarefas de prioridade igual ou superior à prioridade da tarefa *i*. O seu início coincide com o instante crítico para a tarefa i e termina quando todas as activações pendentes de i forem processadas. Este conceito permite ter em conta as instâncias anteriores da tarefa *i*.

Um período ocupado de nível *i*, que comece no instante qT_i antes da invocação actual da tarefa *i*, tem um comprimento:

$$\omega_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{\omega_i(q)}{T_j} \rceil C_j \tag{3.3}$$

onde o termo qC_i considera a contribuição das *q* instâncias da tarefa *i* que começam a executar no instante crítico e que ainda possam estar activas.

O tempo de resposta para o pior caso da tarefa *i* é o máximo dos valores obtidos ao subtrair do comprimento do período ocupado o tempo qT_i, de forma a obter apenas o tempo de resposta efectivo da invocação da tarefa *i*:

$$r_i = max_{q=0,1,2,\cdots}(\omega_i(q) - qT_i) \tag{3.4}$$

São considerados os valores de *q* tal que $\omega_i(q) < (q+1)T_i$. É possível ver que, no caso em que *q=0*, a equação 3.4 é equivalente à equação 3.1. Para uma descrição mais detalhada remete-se o leitor para (Tindell *et al.*, 1992b; Lehoczky *et al.*, 1990).

Tindell (Tindell *et al.*, 1992b), fornece um teste exacto para a análise do tempo de resposta para tarefas com metas arbitrárias, removendo as restrições sobre os períodos relativos das tarefas impostas pelos testes de Lehoczky.

3.2 Controlo de Acesso a Recursos Partilhados

Em sistemas operativos com um escalonamento preemptivo baseado em prioridades e mecanismos de controlo de acesso a recursos partilhados, a inversão de prioridade no acesso a secções críticas (recursos partilhados) pode levar a um tempo de bloqueio imprevisível (ilimitado) de tarefas de alta prioridade. Apesar de não ser possível eliminar completamente este fenómeno, os seus efeitos devem ser minimizados e controlados. Para isso foram propostos dois protocolos de controlo de acesso a recursos partilhados (Sha *et al.*, 1990): o Protocolo de Herança de Prioridade (*Priority Inheritance Protocol*) e o Protocolo de Tecto de Prioridade (*Priority Ceiling Protocol*).

A ideia base por detrás do Protocolo de Herança de Prioridade é modificar a prioridade das tarefas que causam bloqueios. Em particular, quando uma tarefa bloqueia o acesso de uma ou mais tarefas de maior prioridade a uma secção crítica, herda temporariamente a maior prioridade de entre as tarefas bloqueadas.

Teorema 1 *[Sha-Rajcumar-Lehoczky] Com o Protocolo de Herança de Prioridade, uma tarefa J pode ser bloqueada durante um tempo máximo correspondente a $min(n, m)$ secções critícas, onde n é o número de tarefas de menor prioridade que podem bloquear J e m o número de semáforos distintos que podem ser usados para bloquear J.*

Apesar deste protocolo limitar o fenómeno de inversão de prioridade, apresenta alguns problemas. A duração de um bloqueio pode ser substancial devido a bloqueios encadeados e o protocolo não previne possíveis situações de *interboqueio*[1] causadas pela utilização errada de semáforos.

O Protocolo de Tecto de Prioridade é uma extensão ao Protocolo de Herança de Prioridade que evita a formação de interbloqueios encadeados. Esta extensão é realizada introduzindo uma regra que não permite que uma tarefa entre numa secção crítica se

[1] Do Inglês, "*deadlock*"

existirem semáforos que a possam bloquear. Deste modo, uma vez que uma tarefa entre na primeira secção crítica, nunca pode ser bloqueada por instâncias de tarefas de menor prioridade até libertar essa secção crítica.

De modo a concretizar esta regra, é atribuído a cada semáforo um *tecto de prioridade* igual à maior prioridade de entre as tarefas que podem adquirir o semáforo. Só é permitido a uma tarefa J entrar numa secção crítica se a sua prioridade for maior do que todos os tectos de prioridade dos semáforos correntemente adquiridos por outras tarefas que não J.

Teorema 2 *[Sha-Rajcumar-Lehoczky] Com o Protocolo de Tecto de Prioridade, uma tarefa J pode ser bloqueada durante um tempo máximo correspondente à duração de uma secção crítica.*

O máximo tempo de bloqueio B_i de uma tarefa J_i com prioridade P_i, é igual à secção crítica mais longa entre as que pertencem a tarefas com uma prioridade menor do que P_i e guardadas por um semáforo com um tecto de prioridade maior ou igual a P_i.

Os tempos de bloqueio obtidos com a utilização destes protocolos é contemplado nas equações 3.1 e 3.3 através do termo B_i.

3.3 Efeitos do Escalonador

Usualmente, um escalonador preemptivo baseado em prioridades é executado periodicamente (ou em resposta a estímulos específicos). Em cada activação, o escalonador verifica se existem tarefas pendentes (prontas para ser executadas) e coloca-as numa lista de tarefas executáveis, ordenadas por um atributo de prioridade. Por fim, a função de *despacho* coloca em execução a tarefa de maior prioridade. Se a tarefa de maior prioridade for diferente da que se encontrava em execução, diz-se que ocorreu uma *mudança de contexto* ou que a tarefa sofreu uma preempção.

Os modelos de análise apresentados anteriormente, pressupõem que as tarefas periódicas são activadas em múltiplos exactos do período do relógio do escalonador. Deste modo, as tarefas são colocadas na lista de tarefas executáveis logo que ficam potencialmente activas.

Na prática, pode existir um desfasamento (ou atraso) entre o instante em que uma tarefa fica pendente (ou activa) e o instante em que o escalonador detecta essa activação e a coloca na lista de tarefas executáveis. A esse efeito, dá-se o nome de *release jitter*. Tindell (Tindell *et al.*, 1992b) estendeu a análise dada pela equação 3.4 para incluir o efeito do *release jitter*:

$$r_i = max_{q=0,1,2,\cdots}(J_i + \omega_i(q) - qT_i) \tag{3.5}$$

$$\omega_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{J_j + \omega_i(q)}{T_j} \rceil C_j + \tau_i(\omega_i(q)) \tag{3.6}$$

Como se pode verificar pelas equações 3.5 e 3.6 o efeito do *release jitter* tem influência no cálculo do tempo de resposta de uma tarefa; directamente sobre a tarefa i (J_i) e indirectamente na interferência sobre as tarefas de menor prioridade do que i (J_j).

O termo $\tau_i(\omega_i(q))$ representa a sobrecarga de um escalonador periódico durante uma janela de tempo com largura ω, numa tarefa i, e é dado pela equação seguinte:

$$\tau_i(\omega_i(q)) = L(\omega_i)C_{clk} + min(L(\omega_i), K(\omega_i))C_{QL} + max(K(\omega_i) - L(\omega_i), 0)C_{QS} \tag{3.7}$$

onde $L(\omega_i)$ representa o número máximo de activações do escalonador num intervalo ω, $K(\omega_i)$ representa o número de tarefas que o escalonador coloca na lista de tarefas executáveis no intervalo ω, C_{QL} o custo de mover a primeira tarefa para essa lista e

C_{QS} o das tarefas seguintes. Segundo (Tindell *et al.*, 1992b) existe uma diferença de custo entre mover a primeira tarefa e mover as tarefas seguintes.

A resolução da equação 3.5 deve considerar os valores de q que verifiquem a inequação $\omega_i(q) < (q+1)T_i - J_i$, para ter em conta o efeito do *release jitter*.

Tindell (Tindell *et al.*, 1992b) estende a técnica de análise para tarefas com metas arbitrárias, dada pela equação 3.5, para ter em conta tarefas esporádicas.

3.4 Modelo dos Desfasamentos no Tempo

Os modelos de análise apresentados assumem um instante crítico, ou seja, que todas as tarefas de prioridade igual ou superior à prioridade da tarefa i são colocadas simultaneamente na lista de tarefas executáveis. No entanto existem aplicações modeladas por conjuntos de tarefas com desfasamentos (*offsets*) relativos entre si. Por exemplo, um conjunto de tarefas com constrangimentos de precedência pode ser modelado atribuindo desfasamentos temporais às tarefas, de modo que a que uma tarefa só será executada quando a anterior tiver terminado. Nestas situações, a suposição de um instante crítico pode levar a um resultado pessimista, uma vez que esse instante pode nunca ocorrer. Tindell (Tindell, 1994) desenvolveu um método de análise de escalonabilidade para conjuntos de tarefas com desfasamentos de tempo escalonadas por um escalonador preemptivo de prioridades fixas.

As tarefas estão organizadas em conjuntos de tarefas denominadas transacções. Cada transacção t_i tem um período T_{ti}. Cada tarefa da transacção pode ser activada a cada e_i invocações da transacção, tendo portanto um período $e_i T_{ti}$, e é executada com um desfasamento de tempo relativo ao início da transacção. Este desfasamento, ou *offset*, é dado por O_i. Considerando o efeito do *release jitter*, uma tarefa começa efectivamente a ser executada entre O_i e $O_i + J_i$.

O tempo de resposta da tarefa i é o intervalo de tempo que começa em O_i e termina no fim da execução de i. Para calcular o pior tempo de resposta de um período ocupado que incluia i, é necessário investigar todos os períodos ocupados anteriores tal como para a equação 3.3. Para isso, percorrem-se todas as anteriores qe_i invocações da transacção t_i, com $q = 0, 1, 2, ...$ Para cada valor de q é calculado um período ocupado em função de cada tarefa da transacção i. O período crítico começa após o início da transacção com a execução de uma tarefa j arbitrária, mas que faz parte do conjunto de tarefas da transacção t_i, começando a executar um intervalo de tempo $O_j + J_j$ após o início da transacção. O tempo de resposta para o pior caso corresponde ao maior valor calculado.

$$r_i = \max_{q=0,1,2,...} (\max_{\forall j \in tasks(ti)} (w_{i,q} + O_j + J_j - T_{ti}(qe_i + v_{i,ti}) - O_i)) \tag{3.8}$$

$$v_{i,ti} = \left\lceil \frac{O_j + J_j - O_i - J_i}{T_{ti}} \right\rceil \tag{3.9}$$

A equação 3.8 exprime o pior tempo de resposta da tarefa i pertencente à transacção ti, $tasks(ti)$ é o conjunto de tarefas de ti e $w_{i,q}$ o comprimento do pior período ocupado. A equação 3.9 serve como "função normalizadora"para garantir que o período ocupado começa antes (ou coincide com) a última activação de i.

O termo $w_{i,q}$ exprime o comprimento do pior período ocupado de nível i:

$$w_{i,q} = (q+1)C_i + \sum_{\forall t \in trans} I_{i,t} + B_i + \tau_i(w) \tag{3.10}$$

Esta equação é semelhante à equação 3.6 com excepção do termo somatório, que define a interferência das tarefas da transacção t_i, com maior ou igual prioridade à da tarefa i e que, à semelhança das equações 3.1 ou 3.3, corresponde ao número de execuções de cada uma das tarefas dentro do período ocupado, a multiplicar pelo seu

pior tempo de computação. A equação 3.11 exprime $I_{i,t}$.

$$I_{i,ti} = \sum_{\forall j \in hp(i) \cap tasks(ti)} \left\lceil \frac{W_{ti} + w_{i,q} - O_j - v_{j,ti}T_{ti}}{e_j T_{ti}} \right\rceil C_j \tag{3.11}$$

W_{ti}, define o desfasamento no tempo do início do período ocupado em relação ao início da transacção ti. Este desfasamento corresponde a $O_k + J_k$ de uma tarefa k arbitrária, do conjunto de tarefas da transacção t_i.

O maior problema desta técnica de análise é saber qual é a tarefa que deve ser usada para criar o período ocupado para o pior caso. A determinação exacta da equação 3.10 exige que para cada transacção t seja necessário encontrar qual é a tarefa k que origina a máxima interferência num período ocupado. Para isso é necessário examinar a equação 3.10 para todas as possíveis combinações de valores de W_{ti}, para todas as N transacções:

$$\max_{\substack{\forall j \in tasks(1) \\ W_1 = O_j + J_j}} \left(\max_{\substack{\forall j \in tasks(2) \\ W_2 = O_j + J_j}} \ldots \left(\max_{\substack{\forall j \in tasks(N) \\ W_N = O_j + J_j}} (w) \right) \right) \tag{3.12}$$

O que torna a avaliação da equação 3.10 computacionalmente intratável uma vez que o número de casos a considerar cresce exponencialmente com o número de tarefas.

É possível modificar a equação 3.8 de modo a que os valores W_{ti} sejam derivados da equação 3.12:

$$r_i = \max_{q=0,1,2,\ldots} (w_{i,q} + W_{ti} - T_{ti}(qe_i + v_{i,ti}) - O_i) \tag{3.13}$$

3.4.1 Análise tratável

Tindell (Tindell, 1994) desenvolveu um método aproximado que permite encontrar um valor de W_t que maximize a interferência para uma transacção t. Essa aproximação

modifica a dependência exponencial com o número de tarefas, para uma dependência polinomial, utilizando a função *maxv*[2].

Uma vez que $w_{i,q}$ ocorre em ambos os lados da equação 3.10 é necessário iterar sucessivamente, como foi feito para a equação 3.2. Assim, para uma dada iteração de $w^n_{i,q}$, o valor de W_t é dado por:

- $W_{ti} = O_j + J_j$, na equação 3.8 se $t = t_i = trans(i)$.
- $W_t = O_k + J_k$, sendo $k \in tasks(t)$ de forma a maximizar a interferência $I_{i,t}$:

$$W_t = maxv_{\substack{\forall k \in tasks(t) \\ W_t = O_k + J_k}} (I_{i,t}) \tag{3.14}$$

De notar que o número de possíveis períodos ocupados que é necessário examinar, com este algoritmo, é igual ao número de tarefas com prioridade maior ou igual à prioridade da tarefa sob análise.

Finalmente, e voltando à equação 3.8, resta definir um limite para a sequência de valores de q. O limite para o número de períodos ocupados a analisar é dado pela seguinte condição:

$$\forall k \in tasks(ti), (w_{i,q} + O_k + J_k \leq T_{ti}((q+1)e_i + v_{i,ti}) + O_i + J_i) \tag{3.15}$$

que indica que o período ocupado calculado termina antes da libertação da tarefa i.

Palencia e Harbour (Palencia & Harbour, 1998) estendem a técnica de análise dos desfasamentos no tempo, permitindo que o desfasamento e o *release jitter* de uma tarefa sejam maiores do que o período da sua transacção e introduzindo desfasamentos dinâmicos. Os autores introduzem uma notação diferente da utilizada por Tindell, pelo que será abordada de forma resumida.

[2]Se $v' = maxv_{\forall v \in r}(f(v))$, então $\forall v \in r, f(v) \leq f(v')$, que devolve o valor que maximiza o seu argumento.

A interferência, para o pior caso, de uma transacção i no tempo de resposta de uma tarefa b de uma transacção a (τ_{ab}) no instante crítico, potencialmente iniciado com a activação de uma tarefa k pertencente à mesma transacção i, é determinada pela equação seguinte:

$$W_{ik}(\tau_{ab}, t) = \sum_{\forall j \in hp_i(\tau_{ab})} \left(\left| \frac{J_{ij} + \varphi_{ijk}}{T_i} \right| + \left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil \right) C_{ij} \tag{3.16}$$

Onde J_{ij} e C_{ij}, são respectivamente o *release jitter* e o pior tempo de computação da tarefa j pertencente à transacção i, T_i o período da transacção i e $hp_i(\tau_{ab})$ o conjunto de tarefas da transacção i com prioridade maior ou igual à da tarefa τ_{ab} e a serem executadas no mesmo processador que τ_{ab}. O termo φ_{ijk} representa o desfasamento entre uma qualquer tarefa j da transacção i e o instante crítico iniciado por uma tarefa k da mesma transacção:

$$\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) mod T_i \tag{3.17}$$

onde ϕ_{ik} e ϕ_{ij} são os desfasamentos modificados das tarefas k e j, que são determinados pela função $\phi_{ix} = O_{ix} mod T_i$. Esta função é aplicada a todas as tarefas i de maior prioridade do que a tarefa sob análise quer essas tarefas pertençam ou não à mesma transacção.

De modo semelhante ao que é seguido no modelo proposto por Tindell, é necessário determinar, para cada transacção, qual a tarefa a utilizar para criar o instante crítico, para o pior caso. Deste modo, a determinação exacta da equação 3.16 requer que sejam verificadas todas as variações possíveis de uma tarefa entre todas as transacções.

Palencia e Harbour seguem o método aproximado desenvolvido por Tindell, para obter um limite superior na interferência das tarefas de uma transacção i, num período ocupado de largura w:

$$W_i^*(\tau_{ab}, w) = \max_{\forall k \in hp_i(\tau_{ab})} W_{ik}(\tau_{ab}, w) \tag{3.18}$$

O tempo de execução, $w_{abc}(p)$, de cada instância p de uma tarefa b no período ocupado criado com a tarefa τ_{ac} é dado por:

$$w_{abc}(p) = B_{ab} + (p - p_{0,abc} + 1)C_{ab} + W_{ac}(\tau_{ab}, w_{abc}(p)) + \sum_{\forall i \neq a} W_i^*(\tau_{ab}, w_{abc}(p)) \tag{3.19}$$

onde $p_{0,abc}$ corresponde à primeira instância da tarefa τ_{ab} que ocorre no instante crítico. O número de tarefas que é necessário verificar é igual ao número de tarefas de maior prioridade do que a tarefa i que existem em cada transacção do sistema, incluindo a própria tarefa i.

O pior tempo de resposta é obtido subtraindo do tempo de execução o instante que corresponde ao início da transacção:

$$R_{abc}(p) = w_{abc}(p) - \varphi_{abc} - (p-1)T_a + O_{ab} \tag{3.20}$$

onde O_{ab} é o valor do desfasamento não modificado de τ_{ab}. Finalmente, resta determinar o pior dos tempos de resposta, obtidos:

$$R_{ab} = \max_{\forall c \in hp_a(\tau_{ab}) \cup b} [\max_{p=p_{0,abc} \cdots p_{L,abc}} (R_{abc}(p))] \tag{3.21}$$

onde $p_{0,abc}$ corresponde à primeira instância que ocorre no instante crítico e $p_{L,abc}$ à última instância que é necessário verificar e que é proporcional ao comprimento do período ocupado, L, abc.

A técnica de análise descrita acima pode ser aplicada para analisar tarefas com desfasamentos dinâmicos em que os desfasamentos variam entre um valor mínimo e um valor máximo, $[O_{j,min}, O_{j,max}]$, em activações sucessivas de uma tarefa. A variação dos desfasamentos é modelada como um caso especial de um sistema com desfasamentos estáticos, definindo um desfasamento equivalente $O_j^{'}$ e um *release jitter*, $J_j^{'}$, equivalente para cada tarefa:

$$O_j^{'} = O_{j,min} \qquad J_j^{'} = J_j + O_{j,max} - O_{j,min} \tag{3.22}$$

No entanto, na maioria dos sistemas onde os desfasamentos das tarefas podem variar dinamicamente, os seus valores mínimos e máximos (ou ambos) dependem dos tempos de resposta das tarefas que as precedem na transacção. O desfasamento mínimo pode ser função do tempo de resposta para o melhor caso da tarefa precedente e o desfasamento máximo função do tempo de resposta para o pior caso dessa mesma tarefa. Esta aproximação será abordada em maior detalhe na subsecção 3.6.

3.5 Redes de Comunicações de Tempo-Real

Algumas redes de comunicação, como o barramento CAN (ISO, 1993), utilizam um mecanismo de acesso não preemptivo, baseado em prioridades fixas, permitindo que as técnicas de análise para prioridades fixas possam ser directamente aplicadas para calcular o pior tempo de resposta para um conjunto de mensagens.

3.5.1 Pior Tempo de Resposta para Mensagens numa Rede CAN

A análise do pior tempo de resposta para mensagens numa rede CAN foi derivada em (Tindell *et al.*, 1994), com base nos pressupostos seguintes:

- A meta de uma mensagem é menor do que o seu período.
- O *queuing jitter* associado à colocação de uma mensagem no tampão de mensagens da rede é menor do que o período da mensagem.
- O controlador da rede não transmite uma mensagem enquanto existirem mensagens pendentes de prioridade mais elevada.

O pior tempo de resposta para uma mensagem m é definido como o maior intervalo de tempo desde o instante em que a mensagem é colocada no tampão de mensagens do controlador da rede (*queueing delay*) e o instante em que chega ao controlador destino (*transmission delay*):

$$R_m = \omega_m + C_m \tag{3.23}$$

O *queueing delay* ω_m é o maior tempo necessário para a mensagem ganhar acesso à rede, devido à transmissão em curso de uma mensagem de menor prioridade e devido à transmissão de todas as mensagens de maior prioridade que entretanto possam estar pendentes.

$$\omega_m = E(\omega_m + C_m) + B_m + \sum_{\forall j \in hp(m)} \lceil \frac{\omega_i + J_j + \tau_{bit}}{T_j} \rceil C_j \tag{3.24}$$

Esta equação é análoga à equação 3.1, onde $hp(m)$ é o conjunto de mensagens com maior prioridade do que m, com período T_j, *queuing jitter* J_j e um tempo de transmissão C_j (*transmission delay*). O termo τ_{bit} é o tempo de transmissão de um *bit* na rede CAN.

O maior tempo de espera da mensagem m devido à transmissão em curso de uma mensagem de menor prioridade é dado pelo termo B_m. Este tempo é igual ao tempo de transmissão da maior mensagem entre as mensagens de menor prioridade porque,

uma vez iniciada a transmissão, uma mensagem não pode sofrer preempção para a transmissão de outra mensagem:

$$B_m = \max_{\forall k \in lp(m)} C_k \tag{3.25}$$

onde $lp(m)$ é o conjunto de tramas de menor prioridade do que m.

O termo C_m é o tempo de transmissão para o pior caso tendo em conta a máxima inserção de *bits* de enchimento (stuffed) e é dado por:

$$C_m = (\lfloor \frac{34 + 8s_m}{5} \rfloor + 47 + 8s_m)\tau_{bit} \tag{3.26}$$

O termo s_m corresponde à dimensão da trama m em octetos. O factor exprime o número máximo necessário de *bits* de enchimento. Esta equação leva em conta o período mínimo de três *bits* em que o barramento está no estado *idle* e que obrigatoriamente precede a transmissão de qualquer trama.

O termo $E(\omega_m + C_m)$, permite representar a sobrecarga causada pela retransmissão de mensagens num dado intervalo de tempo, inerente ao mecanismo de recuperação de erros do CAN.

Os autores utilizam um modelo de faltas muito simples, considerando um tempo mínimo entre duas faltas consecutivas. Segundo os autores este modelo pode ser determinado com maior precisão recorrendo a observações do comportamento do CAN sobre condições de interferência electromagnética extremas, ou através de modelos estatísticos.

3.6 Análise de Escalonabilidade em Sistemas Distribuídos

Uma das dificuldades em aplicar os métodos de análise de escalonabilidade de prioridades fixas a sistemas distribuídos está relacionada com a falta de uma análise integrada exacta, que possa ser aplicada à priori. Esta análise deve integrar o pior tempo de resposta da tarefa emissora que produz a mensagem (*generation delay*), os atrasos introduzidos pela rede de comunicações na transmissão da mensagem (*queueing+transmission delay*) e o tempo de entrega da mensagem à tarefa destino (*delivery delay*).

3.6.1 Modelo Holistico

Uma das técnicas de análise propostas para determinar os tempos de resposta globais de tarefas e mensagens em sistemas distribuídos é designada por modelo holistíco e foi desenvolvida por Tindell e Clark (**?**). O modelo holistíco utiliza o *release jitter* para modelar a activação diferida das diversas acções que compõem uma transacção. Em sistemas distribuídos, os instantes em que são activadas estas acções não têm um período exacto mas variam com o tempo de resposta da acção precedente.

De acordo com esta aproximação, uma mensagem herda um *release jitter* igual ao tempo de resposta da tarefa emissora e a tarefa destino herda um *release jitter* igual ao tempo de resposta da mensagem. A análise é decomposta, analisando os tempos de resposta para o pior caso de tarefas e mensagens em cada processador e rede de comunicação separadamente, calculando o *release jitter* que cada um destes tempos de resposta determinam e iterando sucessivamente até que todos os *release jitter* convergam. A iteração é necessária devido à dependência mútua entre os subsistemas.

Deste modo, uma tarefa destino $d(m)$ de uma mensagem m herda um *release jitter* $J_{d(m)}$ igual a:

$$J_{d(m)} = r_{s(m)} + a_m + r_{deliver} \tag{3.27}$$

onde $r_{s(m)}$ é o pior tempo de resposta da tarefa emissora da mensagem m, a_m é o pior tempo de envio de m (a chegar ao dispositivo de comunicações do processador destino), e $r_{deliver}$ o pior tempo de entrega de m à tarefa destino.

O pior tempo de resposta das tarefas emissora e receptora é calculado utilizando a análise de escalonabilidade para metas arbitrárias descrita pela equação 3.5. Para calcular a_m e $r_{deliver}$ é necessário estabelecer as equações para o modelo da rede de comunicações utilizada. Em (Tindell *et al.*, 1992b) são derivadas as equações para o protocolo de acesso ao meio, TDMA (*Time Division Multiple Access*), mas é possível utilizar outro protocolo.

Devido à dependência mútua entre as equações de escalonamento holístico é necessário recorrer a sucessivas iterações, à semelhança do que é utilizado para resolver a equação 3.2: na primeira iteração das equações de escalonamento colocamos os valores do *release jitter* a zero; na iteração n os valores herdados dos *release jitter* são os obtidos como resultado da resolução das equações na iteração $n - 1$. Caso exista uma solução, o algoritmo converge e a computação termina quando é obtido o mesmo tempo de resposta em duas iterações sucessivas.

3.6.2 Utilização dos Desfasamentos no Tempo

O conceito de transacção definido no modelo dos desfasamentos no tempo pode ser utilizado na análise de escalonabilidade em sistemas distribuídos (Palencia & Harbour, 1998) considerando que cada tarefa da transacção é activada no fim da execução da tarefa que a precede. Utilizando uma rede de comunicações de tempo-real baseada em mensagens com prioridades fixas, como a rede CAN descrita na subsecção 3.5, é possível modelar a rede como um processador e cada mensagem como uma tarefa.

Deste modo é possível aplicar directamente o modelo dos desfasamentos no tempo, utilizando o conceito de transacção para modelar cada conjunto de tarefas com relações de precedência. Palencia e Harbour (Palencia & Harbour, 1998) utilizam os desfasamentos dinâmicos, definidos na subsecção 3.4 para modelar um sistema distribuído, definindo um desfasamento e um *release jitter* equivalentes nulos para a tarefa que inicia uma transacção e utilizando os seguintes valores para as outras tarefas da transacção:

$$O'_{ij} = O_{ij,min} = R^b_{ij-1}, \quad J'_{ij} = J_{ij} + O_{ij,max} - O_{j,min} = R_{ij-1} - R^b_{ij-1} \tag{3.28}$$

onde R^b_{ij-1} é um limite inferior no tempo de resposta para o melhor caso da tarefa precedente (incluindo zero) e R_{ij-1} é um limite superior para o pior tempo de resposta, da mesma tarefa. Devido à dependência mútua entre os tempos de resposta e os desfasamentos é necessário utilizar um método iterativo semelhante ao descrito para o modelo holístico. Em cada iteração são recalculados os valores equivalentes do *release jitter* através da equação 3.28, com base nos tempos de resposta calculados com o modelo dos desfasamentos no tempo, que por sua vez são utilizados para calcular os novos tempos de resposta.

Em (Palencia & Harbour, 1999), Palencia e Harbour complementam o modelo dos desfasamentos dinâmicos explorando de um modo sistemático as relações de precedência entre tarefas na mesma transacção a serem executadas no mesmo processador. Este complemento tem como objectivo de identificar e resolver possíveis conflitos entre tarefas, no cálculo de um mesmo período ocupado. Utilizando esta técnica é possível reduzir o pessimismo do modelo dos desfasamentos dinâmicos.

3.7 Alocação e Atribuição de Prioridades

A atribuição de prioridades segundo os algoritmos RM ou DM pressupõe um instante crítico. No entanto, quando um conjunto de tarefas não partilha um instante crítico, a atribuição de prioridades segundo os algoritmos RM ou DM deixa de ser óptima. Audsley (Audsley *et al.*, 1991) propôs um algoritmo de atribuição de prioridades para conjuntos de tarefas com desfasamentos para um só processador e que é considerada óptima. Este algoritmo reduz o espaço de soluções do problema de $n!$ para $n^2 + n$, para um conjunto de n tarefas, onde a cada tarefa pode ser atribuída uma entre n prioridades.

Em sistemas multi-processador ou distribuídos com p processadores, o espaço de soluções pode ser ainda maior se considerarmos que uma atribuição de prioridades óptima pode depender também da alocação de tarefas a cada processador. Isto porque o número possível de alocações de n tarefas a p processadores é de p^n (Assumindo uma rede de comunicação como um processador, no caso de um sistema distribuído). Devido à sua dimensão, têm sido utilizadas heurísticas na obtenção de soluções aproximadas ou sub-óptimas para a resolução do problema da atribuição de prioridades e da alocação e escalonamento de tarefas, que vão desde a utilização de técnicas de busca não-guiada, de técnicas de busca guiada, até à utilização de programação linear, entre outras.

Provavelmente a técnica de optimização de busca não guiada mais popular é o *simulated annealing* (Kirkpatrick *et al.*, 1983). Uma das suas principais características é a capacidade de encontrar novas potenciais soluções, a partir de um mínimo local, através de saltos aleatórios no espaço de soluções. Esta capacidade é controlada e reduzida à medida que o algoritmo progride, fazendo com que os saltos aleatórios sejam menos prováveis. Isto corresponde ao "arrefecimento"do sistema, até que o sistema estabilize no ponto de baixa energia. Em (Tindell *et al.*, 1992a), o *simulated annealing* é utilizado para a resolução do problema da alocação em sistemas distribuídos de tempo-real,

em que o objectivo é minimizar a comunicação através da rede de comunicação. Di-Natale e Stankovic (Natale & Stankovic, 1995) assumem uma alocação estática e utilizam o *simulated annealing* para optimizar o escalonamento de tarefas e mensagens, minimizando o *jitter*. Diversas variantes deste algoritmo têm também sido propostas, como a combinação do *simulated annealing* com uma rede neuronal (Salleh & Zomaya, 1998) ou o *Adaptive Simulated Annealing* (Ingber, 1996), que permite amostrar espaços N-dimensionais com maior eficiência.

3.7.1 Simulated Annealing

O algoritmo do *simulated annealing* baseia-se na distribuição de probabilidades de Boltzmann. Começando com um *factor de temperatura* inicial, T_{init}, é avaliada a *energia* de um ponto aleatório inicial do espaço de soluções do problema. O valor de *energia* obtido quantifica o quanto boa ou má a solução é.

Após obter um novo ponto aleatório do espaço de soluções do problema, é avaliada a sua energia (E_{new}). A possibilidade do novo ponto ser aceite como o novo ponto de começo é dada pela distribuição de probabilidades de Boltzmann p:

$$p = \begin{cases} 1 & \text{se } E_{new} < E, \\ e^{\frac{-(E_{new}-E)}{kT}} & \text{caso contrário.} \end{cases} \tag{3.29}$$

onde E_{new} é o valor de energia avaliado para o novo ponto, E a energia do ponto anterior, k a constante de Boltzmann e T o *factor de temperatura*. Se o novo ponto não é aceite, é reposta a configuração anterior; senão, a nova iteração é realizada partindo do novo ponto aceite.

Este ciclo repete-se, usando o mesmo *factor de temperatura*, até ser atingido um determinado *equilíbrio térmico*, imposto pela aplicação, e que normalmente é proporcional

à dimensão do espaço de soluções. Após ser atingido o equilíbrio térmico, o *factor de temperatura* é reduzido proporcionalmente a um *factor de arrefecimento* (usualmente uma multiplicação por T_{cool}, onde $0.95 \leq T_{cool} \leq 0.99$) até que seja encontrada uma solução ou que o *factor de temperatura* atinja um valor mínimo (T_{final}).

É possível verificar (Natale & Stankovic, 1995) que, face a uma busca exaustiva, uma implementação baseada no *simulated annealing* não oferece nenhuma vantagem na obtenção de soluções exactas. No entanto permite obter soluções aproximadas com uma dependência polinomial do tempo, constituindo uma opção viável na resolução deste tipo de problemas. Devido à flexibilidade fornecida pelo *simulated annealing*, tanto a alocação como a atribuição de prioridades podem ser incorporadas no espaço de soluções, com a desvantagem de se obter um espaço de soluções maior.

3.8 Síntese e Discussão

Neste capítulo foi feita uma breve panorâmica sobre as técnicas de análise de escalonabilidade para sistemas periódicos de prioridades fixas, também designadas técnicas RM.

Desde o trabalho pioneiro de Liu e Layland (Liu & Layland, 1973), as técnicas RM evoluíram consideravelmente. Apesar de ser possível obter factores de utilização superiores ($\sum_{i=1}^{n} U_i \leq 1$) com algoritmos de escalonamento de prioridades dinâmicas, como por exemplo o EDF (Earliest Deadline First), de acordo com o qual as prioridades são atribuídas de modo inversamente proporcional às metas absolutas das tarefas, os sistemas de prioridades fixas são mais simples de analisar e são suportados pela maioria dos sistemas operativos e compiladores. No entanto, o algoritmo EDF é considerado óptimo entre todos os algoritmos de escalonamento preemptivos (Dertouzos, 1974) quer no escalonamento de tarefas periódicas quer no de tarefas aperiódicas.

A simplicidade e dependência polinomial do teste de escalonabilidade RM pro-

posto por Liu e Layland ($\sum_{i=1}^{n} U_i \leq n(2^{frac1n} - 1)$), torna-o bastante atractivo para realizar testes de escalonabilidade em tempo de execução, nomeadamente em subsistemas de controlo de admissão. Um exemplo entre muitos da sua aplicabilidade é no subsistema de controlo de admissão para mensagens síncronas, utilizado no protocolo FTT-CAN (Almeida *et al.*, 2002)[3]. Alguns investigadores têm procurado derivar testes mais eficientes e menos pessimistas. Por exemplo Bini et al. (Bini *et al.*, 2003) derivaram um teste de escalonabilidade RM (necessário mas não suficiente), menos pessimista que o de Liu e Layland:

$$\prod_{i=1}^{n}(U_i + 1) \leq 2 \tag{3.30}$$

A equação 3.30 fornece um limite superior na utilização do processador que se situa entre o limite superior mínimo do RM e o limite do EDF. Mais recentemente Bini e Buttazzo (Bini & Buttazzo, 2004) derivaram um teste de escalonabilidade, em que é possível variar a complexidade, em função da potência computacional disponível. Abdelzaher (Abdelzaher *et al.*, 2004) derivou um teste de escalonabilidade para tarefas não periódicas, que também pode ser utilizado no controlo de admissão em tempo de execução. Este trabalho pode representar um primeiro passo no desenvolvimento de uma teoria análoga à DM, para tarefas não periódicas.

No entanto, as técnicas RM não fornecem soluções exactas para o cálculo dos tempos de resposta em sistemas distribuídos. A análise para o pior caso destes sistemas continua a ser um assunto em aberto. As aproximações seguidas pelos modelos holístico e dos desfasamentos no tempo utilizam as técnicas RM, analisando cada processador e cada rede de comunicação como se fossem independentes, o que leva a resultados pessimistas. Apesar destas limitações, tanto o modelo holístico com o modelo dos desfasamentos são aproximações válidas para determinar os tempos de resposta para o pior caso de aplicações distribuídas. Chevochot e Puaut (Chevochot

[3]O protocolo FTT-CAN combina de modo eficiente a comunicação de mensagens síncronas (*time-triggered*) e assíncronas (*event-triggered*).

& Puaut, 2000) estendem o modelo holístico para calcular os tempos de resposta das tarefas de uma aplicação de tempo-real estrito, numa plataforma de execução complexa, tolerante a faltas. Richard, Cottet e Richard (Richard *et al.*, 2001) propõem um método para transformar relações de precedência assíncronas, complexas, entre tarefas que comunicam entre si num sistema distribuído de tempo-real estrito, de modo a que a análise de escalonabilidade do sistema possa ser realizada utilizando o modelo de análise holístico, proposto por Tindell. Pop, Eles e Peng (Pop *et al.*, 2003) desenvolveram um método de análise de escalonabilidade para sistemas de tempo-real distribuídos heterogéneos, compostos por tarefas estímuladas por eventos e tarefas estímuladas no tempo[4], Este método de análise é baseado no modelo dos desfasamentos dinâmicos proposto por Palencia e Harbour. Spuri (Spuri, 1996) adaptou a técnica de análise holística desenvolvida por Tindell a sistemas baseados no escalonamento EDF. Com base neste trabalho, Palencia e Harbour (Gutiérrez & Harbour, 2003) estendem as técnicas de análise dos desfasamentos no tempo (estáticos e dinâmicos), para analisar sistemas que utilizam um escalonamento EDF.

No âmbito das redes de comunicação de tempo-real, Broster, Burns e Navas (Broster *et al.*, 2002), estendem o trabalho desenvolvido por Tindell, utilizando um modelo de faltas mais genérico, baseado numa distribuição de probabilidades aleatória. Para além disso os autores reduzem ligeiramente o pessimismo da análise fornecida pelas equações descritas na Secção 3.5, separando os três bits delimitadores de tramas, de uma trama de dados, considerando que uma trama de dados pode ser processada logo após ter sido recebido o seu último bit. O modelo de análise proposto por Broster, Burns e Navas não considera o cenário de falhas, tal como o identificado em (Rufino *et al.*, 1998) relacionados com a difusão atómica de mensagens suportada pelo CAN, nem inclui os tempos de inacessibilidade do CAN devido a erros do próprio canal e dos *transceivers* dos dispositivos conectados (Pinho & Vasques, 2000). Em (**?**) é delineada uma análise dos tempos de resposta para o pior caso, para sistemas dis-

[4]Do inglês *event-triggered* e *time-triggered*, respectivamente.

tribuídos baseados numa rede Ethernet/IP, no âmbito das tecnologias COTS[5].

Tanto o problema da atribuição de prioridades a um conjunto de tarefas como o da alocação de um conjunto de tarefas a um conjunto de processadores em sistemas distribuídos/multi-processador, consideradas óptimas, são computacionalmente intratáveis ($NP - completo$), excepto para sistemas simples. Por essa razão têm sido utilizados algoritmos de optimização baseados em heurísticas na obtenção de soluções aproximadas ou sub-óptimas. Qualquer das técnicas apresentadas possuem vantagens e desvantagens e é difícil concluir sobre a supremacia de uma delas. Como é referido em (Santos *et al.*, 1997), não existe uma marca de nível[6] que permita testar e comparar as diversas heurísticas propostas na literatura. Para além do *simulated annealing*, outros exemplos de técnicas de busca não-guiada utilizadas em algoritmos de optimização, incluem o *tabu search* (Porto & Ribeiro, 1994) e os algoritmos genéticos[7] (Sandnes & Megson, 1996). Em (Lin *et al.*, 2000) são utilizados o *tabu search* e os algoritmos genéticos, na alocação e escalonamento de tarefas num ambiente multi-processador. É realizada uma avaliação das duas heurísticas e uma comparação dos resultados obtidos, onde se conclui que estes variam com o tipo de aplicação. Altenbernd (Altenbernd, 1996) (Altenbernd & Hansson, 1998) realiza uma avaliação de um conjunto de técnicas de busca não guiada e também de uma heurística construtiva, recorrendo a um largo conjunto de exemplos de aplicações, onde é possível verificar que o desempenho da cada um dos algoritmos varia com o tipo de aplicação. Talvez mais conclusiva seja a comparação de resultados de Garcia e Harbour (Garcia & Harbour, 1995) com o *simulated annealing*. Os autores propõem uma heurística para a optimização de atribuição de prioridades a tarefas e mensagens em sistemas distribuídos de tempo-real, baseada no DM. O algoritmo consiste em variar automaticamente as prioridades das diferentes tarefas de uma transacção em função das metas parciais atribuídas a essas tarefas. Os autores verificam que a heurística proposta

[5] *Comercial-Off-The-Shelf*
[6] Do inglês *benchmark*.
[7] Do inglês *genetic algorithms*.

tem um desempenho superior ao *simulated annealing*. No entanto a utilização desta heurística na optimização de prioridades a um conjunto de canais no *RT-Appia* não é trivial.

As técnicas de busca guiada (também designadas heurísticas construtivas) utilizam um diagrama em árvore para guardar soluções parciais do problema. São explorados os diferentes caminhos do diagrama até que seja encontrada uma solução válida[8]. Existem na literatura muitos exemplos da utilização desta técnica (Ramamritham, 1995) (Altenbernd & Hansson, 1998) (Peng *et al.*, 1997) (Richard *et al.*, 2003), entre outros. Richard, Richard e Cottet (Richard *et al.*, 2001) utilizam um algoritmo de busca guiada para atribuir prioridades numa arquitectura composta por múltiplos barramentos de campo. A análise de escalonabilidade é baseada no modelo holístico. Em (Richard *et al.*, 2003) os autores estendem este trabalho para tratar em simultâneo a alocação de tarefas a processadores e a atribuição de prioridades às tarefas. Dipippo (Dipippo *et al.*, 2001) apresenta uma heurística para mapear um número potencialmente alargado de prioridades globais (únicas) atribuídas a um conjunto de tarefas e recursos globais num sistema distribuído de tempo-real, num número de prioridades locais mais reduzido e disponíveis em cada nó do sistema. As prioridades são atribuídas segundo o algoritmo DM e a análise de escalonabilidade é realizada com uma variante do RM.

Outras heurísticas tratam o problema de alocação e do escalonamento de tarefas, recorrendo a agrupamentos (Abdelzaher & Shin, 2000; Ramamritham, 1995). Estas heurísticas repartem um conjunto de tarefas por vários grupos disjuntos, tentando minimizar um dado custo. Cada grupo disjunto de tarefas é designado por um *agregado* de tarefas[9]. O custo a minimizar na repartição pode ser, a comunicação entre aglomerados (Ramamritham, 1995) ou a preempção de tarefas para o pior caso (Abdelzaher & Shin, 2000). Ramamritham (Ramamritham, 1995) recorre a um algoritmo *branch-and-bound* para alocar aglomerados de tarefas a processadores. Peng, Shin e Abdelza-

[8]Método designado por *branch-and-bound*
[9]Do inglês *cluster*

her (Peng *et al.*, 1997) utilizam um algoritmo *branch-and-bound* para alocar tarefas num sistema de tempo-real distribuído constituído por processadores heterogéneos. Altenbernd e Hansson (Altenbernd & Hansson, 1998) propõe uma heurística construtiva para a alocação de tarefas periódicas de tempo-real estrito a sistemas multiprocessador ou distribuídos, baseada nas metas das tarefas. Em (Kodase *et al.*, 2003) é proposta uma aproximação que elimina as dependências entre tarefas utilizando tampões partilhados entre tarefas com relações de precedência, transformando um conjunto de tarefas *dependentes* em tarefas independentes.

Numa breve referência a ferramentas disponíveis para avaliar o comportamento temporal de sistemas de tempo-real, poderemos destacar a plataforma $PERTS$ (*Prototyping Environment for Real-Time Systems*),[Tri-Pacific], onde o modelo do sistema é especificado usando um editor gráfico. Este analisador suporta os algoritmos RM, DM e EDF. A plataforma $MAST$ (*Modeling and Analysis Suite for Real Time Applications*) (Harbour *et al.*, 2001) oferece um modelo aberto, estímulado a eventos, para descrever o comportamento temporal de sistemas de tempo-real. Para verificar o comportamento temporal das aplicações representadas pelo modelo, o $MAST$ utiliza um conjunto de ferramentas baseadas nas técnicas de análise holística e dos desfasamentos no tempo, descritas neste capítulo. Em complemento a este trabalho, Pasaje, Harbour e Drake (Pasaje *et al.*, 2001) definem uma metodologia baseada em UML, para modelar sistemas de tempo-real orientados a objectos. Esta metodologia foi designada por UML-MAST, é baseada no modelo desenvolvido no MAST. O principal objectivo da metodologia proposta, é o de simplificar a aplicação das técnicas de análise de escalonabilidade a sistemas de tempo-real orientados a objectos.

4

Modelo de Composição

O *RT-Appia* adiciona um conjunto de mecanismos ao modelo de composição do *Appia* que permitem extrair a informação necessária para calcular o pior tempo de resposta de uma composição de protocolos.

Partindo do *diagrama de eventos da composição* é possível calcular o pior tempo de resposta de uma composição de protocolos, utilizando um dos métodos de análise de escalonabilidade descritos no capítulo 3. Um diagrama de eventos é uma estrutura que descreve todos os eventos processados por cada camada de uma composição de protocolos e capta também as relações causais entre esses eventos. Estas relações são importantes pois podem reduzir o pessimismo da análise temporal. A análise de escalonabilidade de uma composição de protocolos baseada no diagrama de eventos será abordada em pormenor no capítulo 5.

Um problema prático não menos importante é o da construção automática do diagrama de eventos directamente a partir da implementação. Uma das vantagens do *RT-Appia* é a de permitir que o programador possa implementar cada camada uma única vez de tal modo que a informação necessária para a análise temporal e para o ambiente de execução possam ser derivadas a partir do mesmo código fonte. Um aspecto interessante da plataforma é a de que as estruturas de dados necessárias para derivar o diagrama de eventos são também usadas pelo ambiente de execução para optimizar o desempenho da implementação e tornar o código mais robusto realizando, a pedido, verificações em tempo de execução dos eventos gerados por cada camada. A

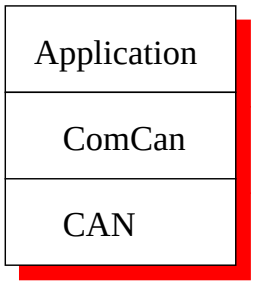

Figura 4.1: Uma pilha de protocolos simples com três camadas.

integração dos ambientes de desenvolvimento, análise e execução, simplifica a tarefa do programador no desenvolvimento de pilhas de protocolos para tempo-real complexas. Como foi referido no capítulo 1, muitos dos sistemas de tempo-real exigem que seja realizada uma análise de escalonabilidade à priori de modo a que seja possível predizer se os requisitos temporais das tarefas de uma aplicação são ou não cumpridos.

Este capítulo descreve o modelo de composição do *RT-Appia* dando ênfase ao modo como é extraída a informação necessária à construção do *diagrama de eventos* de uma composição de protocolos. Para isso, iremos recorrer a um exemplo prático. Esta versão do *RT-Appia* foi implementada em linguagem C++.

4.1 Um exemplo prático

4.1.0.1 Uma Composição de Protocolos Simples

Usaremos uma composição de protocolos muito simples consistindo numa pilha de três camadas como está ilustrado na figura 4.1.

A camada mais baixa corresponde ao acesso ao meio para uma rede CAN. A camada de topo corresponde à aplicação. A camada intermédia "Commited Can"(ComCan) é um protocolo de difusão fiável simples para o CAN, que é essencialmente uma versão simplificada dos protocolos descritos em (Rufino *et al.*, 1998). O

```
Application:
    upon event ⟨ Application.TxDown ⟩ do
        m := payload (); trigger ⟨ ComCan.TxDown, [ m ] ⟩;
    upon event ⟨ Application.TxUp, [ m ] ⟩ do
        // process message m
ComCan:
    upon event ⟨ ComCan.TxDown, [ m ] ⟩ do
        id = newid (); trigger ⟨ CAN.TxDown, [ data, ID, m ] ⟩;
    upon event ⟨ ComCan.TxCnf, [ data, ID, m ] ⟩ do
        trigger ⟨ CAN.TxDown, [ commit, ID ] ⟩;
    upon event ⟨ ComCan.TxUp, [ data, ID, m ] ⟩ do
        buffer: = buffer ∪ {[ data, ID, m ]}; trigger ⟨ ComCan.comCanTimer, [ data, ID, m ] ⟩ with offset timer;
    upon event ⟨ ComCan.TxUp, [ commit, ID ] ⟩ do
        if {[ data, ID, m ]} ∈ buffer then
            buffer := buffer \ {[ data, ID, m ]}; trigger ⟨ Application.TxUp, m ⟩; trigger ⟨ CAN.TxDown, [ commit, ID ] ⟩;
    upon event ⟨ ComCan.comCanTimer, [ data, ID, m ] ⟩ do
        if {[ data, ID, m ]} ∈ buffer then buffer := buffer \ {[ data, ID, m ]};
CAN:
    upon event ⟨ CAN.TxDown, packet ⟩ do
        //send packet to the CAN controller
    upon event controller confirms transmission of packet do
        trigger ⟨ ComCan.TxCnf, packet ⟩;
    upon event controller received message packet do
        trigger ⟨ ComCan.TxUp, packet ⟩;
```

Figura 4.2: Algoritmos para as três camadas.

algoritmo executado em cada uma destas camadas é descrito na figura 4.2 utilizando pseudo-código. No próximo parágrafo será descrita sumariamente a operacionalidade desta composição de protocolos.

A difusão fiável é reforçada pela combinação das propriedades básicas do CAN aumentadas pelo algoritmo ComCan. A análise racional do algoritmo é a seguinte. Foi demonstrado por Rufino *et al.* (Rufino *et al.*, 1998) que a fiabilidade das mensagens de difusão no CAN não é assegurada pela rede se o emissor da mensagem falhar durante a transmissão. Deste modo, o algoritmo "ComCan"só entrega uma mensagem quando tem a certeza de que o emissor não falhou durante a transmissão. Esta informação é fornecida pelo emissor, com a transmissão de uma mensagem designada COMMIT (e daí o nome do protocolo). O COMMIT é retransmitido por todos os nós, para garantir a entrega da mensagem de dados (DATA) original, na casualidade do emissor falhar durante a transmissão da mensagem COMMIT. Embora este protocolo não seja muito eficiente, permite ilustrar alguns dos problemas relacionados com a extracção do diagrama de eventos. Uma versão mais eficiente, mas também mais sofisticada, de um

conjunto de protocolos de difusão fiável para o CAN é descrita em (Rufino *et al.*, 1998).

Vale a pena também referir que a descrição do protocolo apresentada na figura 4.2 não é completamente modular, uma vez que cada camada activa explicitamente *Triggers* das camadas adjacentes. Como ficará claro mais adiante, o modelo de composição do *RT-Appia* elimina estas limitações.

4.1.0.2 O Diagrama de Eventos da Composição

Estudando o algoritmo de cada camada, é possível construir manualmente o diagrama de eventos da composição de protocolos. O diagrama de eventos captura uma cadeia causal de execução de *gestores de eventos*[1] em resposta a um estímulo externo (tipicamente pedidos de transmissão de dados impostos ao sistema). Cada nó do diagrama é identificado pelo nome do gestor e o estado associado ao evento a ser processado (nomeadamente a mensagem a ser processada).

O diagrama de eventos para o nosso exemplo está representado na figura 4.3. O diagrama pode ser construído capturando a sequência de eventos desde o instante em que é recebido um pedido de transmissão. Estes pedidos são modelados com um período T. Pode ser observado que, após a activação de um pedido (evento $E1$) a aplicação activa a transmissão da mensagem DATA ($E2$). Este pedido de transmissão faz activar uma cadeia de eventos nas restantes camadas, quer no nó emissor quer nos nós remotos. Só serão descritos em detalhe os primeiros passos desta cadeia, visto que o mesmo procedimento pode ser aplicado para derivar toda a cadeia. O pedido de transmissão activado pela camada de *Aplicação* é processado pela camada *ComCan*, a qual adiciona à mensagem um cabeçalho de controlo. De seguida a camada *ComCan* envia o pedido de transmissão à camada *CAN* ($E3$). Isto dá origem a uma confirmação local ($E4$) e a uma indicação em todos os nós ($E5$). Estes dois eventos por sua vez geram outros eventos como está representado no resto da figura. Note-se que no dia-

[1] Do Inglês, "*event handlers*"

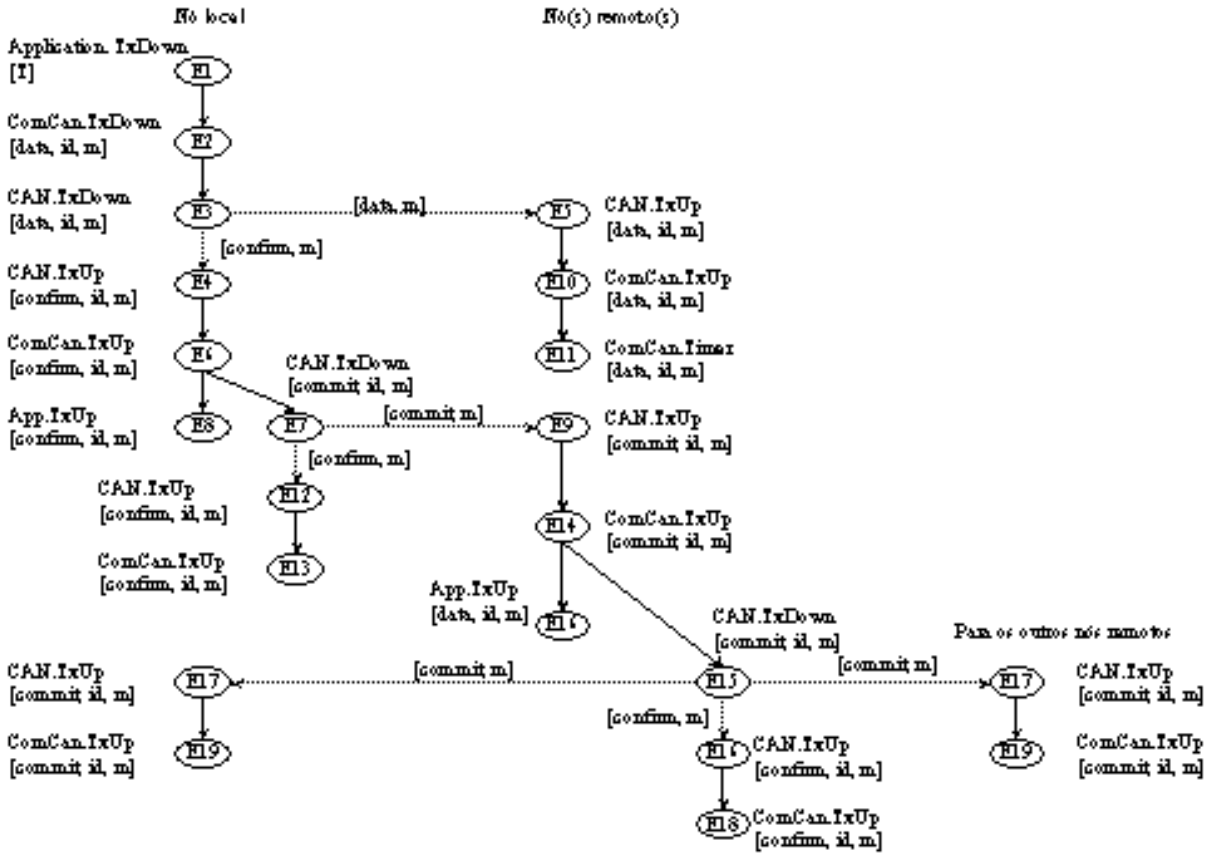

Figura 4.3: Diagrama de eventos da composição de protocolos.

grama de eventos, os eventos que transportam uma mensagem DATA estão etiquetados com o tipo de mensagem associada ao evento. Deste modo e para efeitos de análise temporal os eventos $E10$ e $E14$ são eventos diferentes.

4.2 Derivação do Diagrama de Eventos

Nesta secção iremos ver como a moldura *RT-Appia* permite derivar o diagrama de eventos directamente do código do programador, com base no exemplo descrito na

secção anterior. Antes serão introduzidos os mecanismos mais relevantes do modelo de composição do *RT-Appia*.

4.2.1 Canal de Tempo-Real

A Listagem 4.1 ilustra como pode ser criado um canal de tempo-real no *RT-Appia*. Primeiro é criada a configuração do canal, criando o objecto mystack. As camadas são adicionadas à configuração utilizando o método bottom. Note-se que cada camada é representada por um objecto *singleton* (um padrão utilizado nas linguagens orientadas para objectos para assegurar que uma classe só possuiu uma única instância). De seguida são criadas sessões a partir de cada camada. O paço seguinte consiste em criar um objecto canal e subsequentemente atribuir as sessões ao canal. Cada instância do canal é criada com um atributo de prioridade. O atributo de prioridade do canal deve reflectir a prioridade relativa do canal (em relação a outros canais criados) no processamento de eventos. Por exemplo, as prioridades dos canais podem ser atribuídas de acordo com o algoritmo *RM*, *DM* ou outro. Finalmente, visto que o canal é utilizado para estabelecer a comunicação entre dois ou mais nós, o utilizador deve especificar os identificadores dos nós envolvidos na troca de mensagens.

4.2.2 Eventos

Todos os eventos são descendentes da classe base, RTEvent. A declaração de eventos segue uma metodologia que oferece a informação reflexiva necessária para suportar quer a captura do diagrama de eventos, quer o escalonamento de eventos em tempo de execução. Deste modo é associado a cada tipo de evento um objecto *singleton* que representa essa classe (Gamma *et al.*, 1995). Por exemplo, o tipo RTEvent é representado pela classe RTEventClass. Os testes ao tipo de evento são sempre realizados sobre a classe base que satisfaça os requisitos desejados.

Listagem 4.1: Criar um canal de tempo-real.

```
void main (int argc, char* argv[])
{
  Configuration* mystack = new Configuration ();
  mystack.bottom (applicationLayer);
  mystack.bottom (comcanLayer);
  mystack.bottom (canLayer);

  Session* applSession = applicationLayer.createSession ();
  Session* comcanSession = comcanLayer.createSession ();
  Session* canSession = canLayer.createSession ();

  RTChannel* mychannel = mystack.createChannel (channel_priority);
  mychannel.bottom (applSession);
  mychannel.bottom (comcanLayer);
  mychannel.bottom (canLayer);

  mychannel.addnode (Node1);
  mychannel.addnode (Node2);
  mychannel.addnode (Node3);
  mychannel.addnode (Node4);
  // ...
}
```

4.2.3 Eventos aceites e gerados

Para derivar o diagrama de eventos de uma composição de protocolos, é necessário em primeiro lugar identificar quais os eventos processados em cada camada. Para isso o *RT-Appia* necessita que o programador identifique quais os eventos aceites e gerados em cada camada de protocolo. Esta informação permite por um lado derivar o diagrama de eventos e por outro lado construir as tabelas de encaminhamento de eventos (um conceito importado do *Appia*). Neste sentido, o programador deve invocar os métodos accepts e provides cuja assinatura é representada na listagem 4.2.

Estes métodos devem ser invocados no construtor da classe que representa a camada de um protocolo e designada RTLayer.

Os parâmetros para o método accepts são os seguintes: o primeiro parâmetro define a classe do evento aceite, um objecto *singleton*, que representa a classe; o segundo parâmetro indica a direcção com que o evento se propaga na pilha (UP/DOWN); o

Listagem 4.2: Métodos accepts e provides para uma camada (RTLayer).

```
class RTLayer {
public:
  RTLayer ();
  ~RTLayer ();

  void accepts (RTEventClass *event_class, Direction dir, RTHandler handler, Duration wcct);
  void provides (RTEventClass *event_class, Direction dir);
  // ....
};
```

terceiro parâmetro é o endereço do método da sessão que processa o evento (isto é, o gestor do evento); finalmente o último parâmetro é o tempo de computação para o pior caso ($TCPC$) do gestor, o qual na presente versão da moldura tem de ser declarado explicitamente pelo programador. O método provides aceita somente a classe do evento e a sua direcção.

A listagem 4.3 mostra a declaração dos eventos aceites e gerados para a nossa composição de protocolos simples. Existem alguns pontos importantes que vale a pena referir. O leitor notará que o mesmo evento, da classe dataTxEventClass, flui em ambas as direcções na pilha de protocolos. Este evento é produzido pela *Aplicação* na direcção descendente e é processado pela camada *ComCan*, que o encaminha para a camada *CAN*. Quando o evento é recebido da rede de comunicação, preserva a sua classe mas é agora propagado na direcção ascendente: é em primeiro lugar processado pela camada *CAN*, encaminhado para a camada *ComCan* e consumido pela *Aplicação*. Usando este padrão, o programador de cada camada não precisa de estar a par de quais são as camadas adjacentes que serão utilizadas na composição final de protocolos. Isto facilita a adição ou remoção de camadas para adaptar a pilha de protocolos a requisitos específicos.

Por outro lado, usando a informação fornecida na *configuração* de um canal, é possível verificar em tempo de execução exactamente quais as camadas que aceitam cada evento. Como já foi referido, a informação obtida através dos métodos accepts

e provides permite construir uma tabela de encaminhamento para cada evento. Deste modo esta informação é útil não só para a análise temporal, mas também para suportar uma execução eficiente da composição de protocolos.

Vale também a pena sublinhar como é que o modelo de composição do *RT-Appia* permite expressar o facto de o protocolo *ComCan* processar dois tipos de mensagens diferentes: DATA e COMMIT. Isto é conseguido derivando um nome especializado, CommitEvent da classe DataTxEvent, como está ilustrado na Listagem 4.4. Utilizando esta técnica, é possível distinguir ambos os eventos ao nível da camada *ComCan* (associando um *handler* diferente a cada evento) sem alterar uma única linha de código ao nível da camada *CAN* (que processa ambas as mensagens do mesmo modo, suportando unicamente a classe base). Esta característica suporta a reutilização de implementações de protocolos em diferentes contextos, uma propriedade importante das molduras de composição de protocolos.

4.2.4 Implementação de uma Sessão

A classe Sessão captura o estado mantido pelo protocolo e descreve o comportamento do protocolo em termos de gestores de eventos. Para ilustrar a implementação de uma sessão no *RT-Appia*, é apresentado na listagem 4.5, um exemplo de um gestor da sessão *ComCan*. O gestor em questão processa um pedido de transmissão de dados recebido da camada acima (no nosso caso, a camada de aplicação). Neste caso, o gestor adiciona simplesmente um cabeçalho à mensagem e faz seguir o evento para a camada seguinte. Esta operação é realizada invocando o método go do evento. Faz-se notar que o programador não tem conhecimento explícito sobre qual é a camada que está acima ou abaixo (pois esta informação é definida na fase de *configuração* do canal). No entanto, utilizando a informação sobre os eventos aceites, fornecida por cada camada, a moldura *RT-Appia* é capaz de encaminhar o evento no canal, garantindo que todas as sessões irão processar o evento pela ordem correcta.

Listagem 4.3: Métodos accepts e provides para as camadas ApplicationLayer, ComCanLayer e CANLayer.

```
ApplicationLayer:: ApplicationLayer ()
{
  accepts (dataTxEventClass, DOWN, &ApplicationSession::hDataTxDown, APP_DTXDOWN_WCCT);
  accepts (dataTxEventClass, UP, &ApplicationSession::hDataTxUp, APP_DTXUP_WCCT);
  accepts (dataCnfEventClass, UP, &ApplicationSession::hDataCnfUp, APP_DCNF_WCCT);

  provides (dataTxEventClass, DOWN);

  // ...
}

ComCanLayer::ComCanLayer ()
{
  accepts (dataTxEventClass, DOWN, & ComCanSession::hDataTxDown, CCAN_DTXDOWN_WCCT);
  accepts (dataTxEventClass, UP, & ComCanSession::hDataTxUp, CCAN_DTXUP_WCCT);
  accepts (dataCnfEventClass, UP, & ComCanSession::hDataCnfUp, CCAN_DCNF_WCCT);
  accepts (commitEventClass, DOWN, & ComCanSession::hComCanTimer, CCAN_TIME_WCCT);
  accepts (commitEventClass, UP, & ComCanSession::hConfirmUp, CCAN_DTXUP_WCCT);

  provides (dataTxEventClass, DOWN);
  provides (confirmEventClass, DOWN);
  provides (dataTxEventClass, UP);
  provides (commitEventClass, DOWN);

  // ...
}

CANLayer:: CANLayer ()
{
  accepts (dataTxEventClass, DOWN, & CanSession::hDataTxDown, CAN_DTXDOWN_WCCT);
  accepts (dataTxEventClass, UP, & CanSession::hDataTxUp, CAN_DTXUP_WCCT);
  accepts (dataCnfEventClass, UP, & CanSession::hDataCnfUp, CAN_DTXUP_WCCT);

  provides (dataTxEventClass, UP);
  provides (dataCnfEventClass, UP);

  // ...
}
```

Listagem 4.4: Tipos de eventos.

```
class CommitEventClass: public Class {
 public:
  CommitEventClass () {
    setsuper (dataTxEventClass);
  }

  virtual RtEventClass* make () {
   CommitEvent event = new CommitEvent (this);
   }
};

static CommitEventClass *commitEventClass = new CommitEventClass ();

class CommitEvent: public DataTxEvent {
 public:
  CommitEvent (Class *c) {
    setclass (c);
  }
  // ...
};
```

Listagem 4.5: Implementação de uma sessão.

```
ComCanSession::hDataTxDown (RtEvent* e)
{
  RtMessage *msg = e.getMessage ();

  msg.pushChar (DataHeader, HeaderSize);
  e.go ();
};
```

Listagem 4.6: Métodos para os *Triggers*.

```
const RTEventClass *FORWARD = NULL;
const NodeSet *ALLNODES = NULL;
const int PERIODIC = ~0;
enum TriggerSemantics { Default, OnFirst, OnLast }
enum TriggerLocation { Local, Remote }
enum IdOperation { NewId, SameId, AddSubId, RemoveSubId }
enum SizeOperation { NewSz, FromTrigerId, FromProvidedId }

class RTLayer {
public:
  // ....
  localTrigger        (NodeSet *source,
                       RTEventClass *accepted, Direction acc_dir,
                       RTEventClass *provided, Direction prov_dir,
                       IdOperation idop, int id_desc,
                       SizeOperation size_op, int size,
                       TriggerSemantics semantics,
                       Duration period, int max_activations);

  networkTrigger      (NodeSet *source,
                       RTEventClass *accepted, Direction acc_dir,
                       RTEventClass *provided, Direction prov_dir,
                       IdOperation idop, int id_desc,
                       SizeOperation size_op, int size,
                       TriggerSemantics semantics,
                       NetworkModel *network, TriggerLocation loc);
};
```

4.2.5 Triggers

O conhecimento do conjunto dos eventos aceites e gerados não é suficiente para construir o diagrama de eventos. De facto é necessário conhecer a relação causal entre eventos aceites e gerados. De modo a tornar esta relação explícita, o *RT-Appia* obriga o programador a declarar *Triggers*. Um *Trigger* indica que um dado evento *provided* é gerado em resposta a um associado evento *accepted*. Os métodos *Triggers* disponíveis estão representados na listagem 4.6.

Existem dois tipos de *Triggers*: locais (*local*) e de rede (*network*). Os *Triggers* locais captam a comunicação entre sessões num único nó. Os *Triggers* de rede captam a comunicação entre nós diferentes utilizando uma dada rede. Os parâmetros para

ambos os *Triggers* são os mesmos, com algumas excepções descritas adiante.

O primeiro parâmetro, source, indica o conjunto de nós onde o *Trigger* é activado. A maioria dos *Triggers* são activados em todos os nós (a palavra reservada, ALLNODES, é definida para este efeito). No entanto este parâmetro oferece a possibilidade de especificar que certos *Triggers* só são activados em certos nós. Como veremos mais adiante com o nosso exemplo, este aspecto é particularmente proveitoso para modelar a carga num sistema (uma vez que nem todos os nós podem ter permissão para transmitir mensagens).

Os quatro parâmetros seguintes são a classe e a direcção de um evento aceite e a classe e a direcção do evento gerado, respectivamente. Esta informação é o elemento chave de um *Trigger*, uma vez que permite associar um evento aceite com um evento gerado. A palavra reservada, FORWARD é usada para especificar que o evento gerado é idêntico ao evento aceite. Neste caso, a classe do evento é preservada.

Os restantes parâmetros são menos óbvios mas extremamente importantes para preservar a modularidade do sistema. O IdOperation é usado para resolver relações ambíguas (para distinguir *Triggers* distintos no mesmo *handler* no contexto da mesma cadeia de eventos). Existem casos em que é necessário distinguir ramos diferentes no diagrama de eventos para a mesma classe de evento aceite por uma camada. Quando um *Trigger* é activado é possível:

- Criar uma nova instância de um identificador raiz (NewId). Um identificador raiz é um quarteto que consiste: no identificador do nó, na camada onde o identificador foi criado, na classe do evento activado e finalmente num identificador numérico fornecido pelo programador (parâmetro id_desc).

- Preservar o identificador do evento aceite (SameId). Este é o comportamento mais comum de um *handler* que gera o mesmo evento após algum processamento.

- Adicionar um sub-identificador ao identificador do evento aceite (AddSubId). É

usado por uma camada que gera mais do que um evento em resposta a um único evento aceite (por exemplo uma camada que fragmenta uma mensagem). O programador especifica um valor numérico (o parâmetro id_desc) para distinguir cada sub-identificador. Os sub-identificadores são empilhados no topo do identificador raiz.

- Remover um sub-identificador (RemoveSubId). Esta operação é oposta à operação AddSubId. Uma camada que previamente adicionou um sub-identificador a um evento pode extrair o identificador original. Isto é útil para modelar a reconstrução de uma mensagem original a partir de múltiplos fragmentos.

O parâmetro SizeOperation permite ao programador especificar a dimensão das mensagens criadas por uma camada. Esta informação é importante para obter uma análise temporal precisa de uma composição de protocolos, uma vez que a dimensão das mensagens está directamente relacionada com o tempo de transmissão na rede de comunicações. O programador pode especificar três operações diferentes: declarar a dimensão absoluta de uma mensagem (NewSz), aumentar a dimensão de uma mensagem (esta operação, FromTriggerId, modela a adição de cabeçalhos às mensagens) e recuperar a dimensão da mensagem associada a um dado identificador (esta operação, ProvidedId, modela operações tais como a reconstrução de mensagens a partir dos seus fragmentos).

O parâmetro TriggerSemantics capta o facto de que, em muitos protocolos, os *Triggers* podem ser activados só uma vez para uma dada mensagem, mesmo nos casos em que o evento aceite é activado mais do que uma vez. Consideremos por exemplo uma camada que retransmite uma mensagem mas rejeita duplicados na recepção. Mesmo que a mensagem seja recebida mais do que uma vez, só uma cópia é encaminhada para a aplicação. De modo semelhante, uma camada que reconstrua uma mensagem pode receber diversos fragmentos e só entregar uma mensagem, quando todos os frag-

mentos forem recebidos. As três semânticas correntemente suportadas pelo *RT-Appia* modelam estes diferentes casos. A semântica Default activa o *Trigger* sempre que o evento aceite seja recebido. A semântica OnFirst para o mesmo identificador de mensagem, rejeita todas as activações duplicadas do *Trigger*. A semântica OnLast, para um dado identificador da mensagem, só considera a última activação (*Triggers* com a semântica OnLast só são avaliados quando não possam ser activados mais *Triggers* com as semânticas Default ou OnFirst.

Os últimos parâmetros distinguem os *Triggers* locais dos *Triggers* de rede. Num *Trigger* local o programador pode especificar um período e um número máximo de activações. Esta informação é usada para modelar temporizadores e cargas periódicas para a análise temporal. No *Trigger* de rede, o programador especifica um objecto *modelo de rede*. Um modelo de rede é um componente que determina as propriedades temporais da rede de comunicação de tempo-real utilizada. Num *Trigger* remoto é também possível especificar se o destinatário do evento num canal de tempo-real é local (por exemplo na confirmação de uma transmissão) ou remoto(s) (tipicamente nas indicações de dados).

As Listagens 4.7, 4.8, e 4.9 apresentam todos os *Triggers* definidos para o nosso exemplo prático. Estas Listagens ilustram como podem ser utilizados os diferentes parâmetros dos *Triggers*.

O *Trigger* da camada de aplicação, na Listagem 4.7, ilustra como é criada uma nova mensagem. Neste caso a mensagem é enviada em resposta ao temporizador associado ao evento dataTxEvent (um evento periódico com período T). Note-se que a mensagem só é enviada pelo nó 1. São também especificados, no *Trigger* do evento, a criação de um novo identificador e a dimensão da mensagem.

Os *Triggers* para a camada *ComCan*, na Listagem 4.8, mostram como as mensagens COMMIT podem ser combinadas com as messages DATA associadas, através da utilização de sub-identificadores. Por exemplo, quando é enviada uma mensagem

Listagem 4.7: Triggers para a camada de aplicação.

```
NodeSet* data_sources = new NodeSet (Node1);
Duration dataTxPeriod = 1000;

ApplicationLayer:: ApplicationLayer ()
{
  // ...
  // iniciar transmissao apos o temporizador de dataTxEvent disparar
  localTrigger (data_sources,          // no no 1
                dataTxEventClass, DOWN, // evento aceite
                dataTxEventClass, DOWN, // evento gerado
                NewId, 1,              // com um novo identificador
                NewSz, PAYLOAD_SZ,     // com uma nova dimensao
                Default,               // semantica por defeito
                dataTxPeriod, 0);      // com periodo dataTxPeriod
}
```

COMMIT (commitEventClass, DOWN), esta herda o identificador da mensagem DATA original e é adicionado um sub-identificador para a distinguir do pedido original. Mais tarde, quando é gerada uma indicação de dados (dataTxEventClass, UP), em resposta à mensagem de confirmação de dados, o sub-identificador é removido e a dimensão original da mensagem de DATA correspondente é recuperada utilizando o valor FromProvidedId para o parâmetro SizeOperation. O mesmo exemplo ilustra a utilização do parâmetro TriggerSemantics. Seleccionando o valor OnFirst, é eliminada a geração duplicada de eventos COMMIT e eventos DATA, redundantes.

Finalmente os *Triggers* da camada CAN ilustram a utilização dos networkTrigger. O objecto canModel é usado pela ferramenta de análise de escalonabilidade para derivar os desfasamentos temporais (*offsets*) das confirmações locais e das indicações remotas, em resposta a um pedido de transmissão de dados à rede CAN.

Os *Triggers* foram introduzidos no *RT-Appia* para derivar o diagrama de eventos de modo automático. No entanto são também usados para suportar a depuração e validação da execução de protocolos. De facto, é possível configurar a implementação de modo a que a geração de eventos em tempo de execução seja confrontada com a especificação dos *Triggers*. Se um *handler* fornece um evento que não conste da

Listagem 4.8: Triggers para a camada *ComCan*.

```
ComCanLayer::ComCanLayer ()
{
  // ...
  // faz seguir o mesmo evento
  localTrigger (ALLNODES,                  // trigger em todos os nos
                dataTxEventClass, DOWN,    // evento aceite
                FORWARD, DOWN,             // evento gerado
                SameId, 0,                 // com o mesmo identificador
                FromTriggerId, HeaderSize, // adicionar HeaderSize a dimensao
                Default,                   // semantica por defeito
                0, 0);                     // irrelevante
  // iniciar temporizador de disparo unico, apos receber dados
  localTrigger (ALLNODES,                  // trigger em todos os nos
                dataTxEventClass, UP,      // evento aceite
                commitTimerEventClass, DOWN, // evento gerado
                SameId, 0,                 // manter o mesmo identificador
                NewSz, 0,                  // irrelevante
                Default,                   // semantica por defeito
                comCanTimer_period, 1);    // temporizador de disparo unico
  // enviar commit quando receber confirm do controlador
  localTrigger (ALLNODES,                  // trigger em todos os nos
                dataCnfEventClass, UP,     // evento aceite
                commitEventClass, DOWN,    // evento gerado
                AddSubId, 1,               // adicionar subidentificador
                NewSz, CommitMessageSize, // dimensao da mensagem commit
                OnFirst,                   // activar trigger uma so vez
                0, 0);                     // irrelevante
  // entregar a mensagem de dados quando receber commit
  localTrigger (ALLNODES,                  // trigger em todos os nos
                commitEventClass, UP,      // evento aceite
                DataTxEventClass, UP,      // evento gerado
                RemoveSubId, 0,            // recuperar o identificador original
                FromProvidedId, 0,         // dimensao da mensagem original
                Default,                   // semantica por defeito
                0, 0);                     // irrelevante
  // re-transmitir commit
  localTrigger (ALLNODES,                  // trigger em todos os nos
                commitEventClass, UP,      // evento aceite
                commitEventClass, DOWN,    // evento gerado
                RemoveSubId, 0,            // recuperar o identificador original
                SameSz, 0,                 // mesma dimensao
                Default,                   // semantica por defeito
                0, 0);                     // irrelevante
}
```

Listagem 4.9: Triggers para a camada CAN.

```
CANLayer:: CANLayer ()
{
  // ...
  // entregar a mensagem aos nos remotos
  networkTrigger (ALLNODES,              // trigger em todos os nos
                  dataTxEventClass, DOWN, // evento aceite
                  FORWARD, UP,           // evento gerado
                  SameId, 0,             // manter o mesmo identificador
                  FromTriggerId, 0,      // mesma dimensao
                  Default,               // semantica por defeito
                  canModel,              // usar o modelo de rede CAN
                  Remote);               // trigger remoto
  // confirmar a transmissao da mensagem de dados
  networkTrigger (ALLNODES,              // trigger em todos os nos
                  dataTxEventClass, DOWN, // evento aceite
                  dataCnfClass, UP,      // evento gerado
                  SameId, 0,             // manter o mesmo identificador
                  NewSz, 0,              // nova dimensao
                  Default,               // semantica por defeito
                  canModel,              // usar o modelo de rede CAN
                  Local);                // trigger local
}
```

informação fornecida pelos *Triggers*, é gerada uma excepção em tempo de execução.

4.2.6 Diagrama de precedências de eventos

O diagrama de precedências de eventos é construído de um modo iterativo, começando pelo evento raiz e verificando as especificações dos *Triggers* e as declarações dos eventos aceites e gerados em cada camada. A verificação dos *Triggers* por si só não é suficiente, uma vez que um evento pode não ser entregue à camada adjacente na pilha de protocolos. Quando um trigger é activado, é verificada qual a próxima camada na configuração do canal, segundo a direcção do evento, que o declara como aceite e é adicionada um novo nó ao diagrama de eventos. Este procedimento é iterado até que não sejam criados mais eventos. Um algoritmo de exploração sistemática como o *Depth First Search* pode ser utilizado para implementar este procedimento iterativo.

Cada nó do diagrama é um descritor de evento. Cada descritor contém a seguinte

informação: a classe do evento e a sua direcção; o $TCPC$ do gestor associado ao evento; o período e a meta do evento; a camada onde o evento é activado; o tipo do *Trigger* (local/remoto); o identificador da mensagem associada ao evento (se existir) e a sua dimensão;

4.2.7 Camada de fragmentação e reconstrução de mensagens

Consideremos agora o caso de o programador necessitar de incluir na pilha de protocolos do exemplo prático uma nova camada, para fragmentação e reconstrução de mensagens. Esta nova camada (FragCan) seria introduzida entre a camada de aplicação e a camada *ComCan* e seria usada pela aplicação para transmitir mensagens com uma dimensão superior a oito octetos e menor que dezasseis octetos (ou seja num máximo de dois fragmentos).

A listagem 4.10 mostra a declaração dos *Triggers* e dos eventos aceites e gerados para a camada *FragCan*.

O novo evento DataTxFrag é um sub-tipo de DataTxEvent. A declaração dos *Triggers* para a direcção descendente (DOWN) modelam os dois fragmentos gerados pela camada, por cada mensagem recebida da aplicação. De notar que é adicionado um sub-identificador diferente a cada fragmento. O *Trigger* na direcção ascendente modela a reconstrução da mensagem e ilustra a utilização da semântica OnLast, que permite especificar que a mensagem será entregue à aplicação quando for recebido o último fragmento.

Na camada de aplicação o programador deverá adicionar o evento dataTxFrag, que é representado pela classe dataTxFragClass e é declarado de modo idêntico ao que foi descrito para o evento dataTxEvent. De notar também que durante a execução, o evento DataTxEvent não é entregue à camada *FragCan*.

Para terminar, é importante referir que a adição da camada *FragCan* não implica qualquer modificação nas camadas *ComCan* e *CAN*, quer na declaração de eventos ou

Listagem 4.10: Declaração dos eventos e Triggers para a camada FragCan.

```
FragCanLayer::FragCanLayer ()
{
  accepts (dataTxFragClass, DOWN, &FragCanSession::hFragTxDown, FCAN_FTXDOWN_WCCT);
  accepts (dataTxFragClass, UP, &FragCanSession::hFragTxUp, FCAN_FTXUP_WCCT);

  provides (dataTxFragClass, DOWN);
  provides (dataTxFragClass, UP);

  // gerar dois fragmentos para a camada ComCan
  localTrigger (ALLNODES,                 // trigger activado em todos os nos
                dataTxFragClass, DOWN,    // classe e direccao do evento aceite
                FORWARD, DOWN,            // gera a mesma classe e mesma direccao
                AddSubId, 1,              // adicionar um subidentificador
                FromTriggerId, HeaderSize,// dimensao = request + header
                Default,                  // semantica por defeito
                0, 0);                    // nao periodico
  localTrigger (ALLNODES,                 // trigger activado em todos os nos
                dataTxFragClass, DOWN,    // classe e direccao do evento aceite
                FORWARD, DOWN,            // gera a mesma classe e mesma direccao
                AddSubId, 2,              // adicionar um subidentificador
                FromTriggerId, HeaderSize,// dimensao = request + header
                Default,                  // semantica por defeito
                0, 0);                    // nao periodico

  // reconstrucao da mensagem para a aplicacao
  localTrigger (ALLNODES,                 // trigger activado em todos os nos
                DataTxFragClass, UP,      // classe e direccao do evento aceite
                FORWARD, UP,              // gera a mesma classe e mesma direccao
                RemoveSubId, 0,           // o mesmo identificador do pedido original
                FromProvidedId, 0,        // dimensao da mensagem original
                OnLast,                   // activar no ultimo fragmento
                0, 0);                    // nao periodico
  // ...
}
```

Triggers, quer no código dos *handlers* das sessões. A única modificação ocorrerá necessariamente na camada de aplicação, no sentido de garantir que cada evento só é entregue às sessões que o querem processar.

4.3 Avaliação e Discussão

Um dos maiores desafios no projecto do *RT-Appia* foi a definição do mecanismo dos *triggers*, de modo a automatizar a extracção do diagrama de eventos. Visámos obter os seguintes objectivos:

- Preservar a modularidade, no sentido de que, quando uma camada é especificada, o programador não se deveria preocupar com os parâmetros de configuração tais como: as camadas que serão executadas acima e abaixo; a dimensão das mensagens a serem trocadas; a sub-classe exacta dos eventos aceites (o/a programador(a) só deveriam ter conhecimento da sua classe base); etc.

- Suportar uma larga extensão de protocolos, incluindo: protocolos que fragmentam e reconstroem mensagens, protocolos que geram mensagens de controlo associados com mensagens de DATA (tais como mensagens de reconhecimento (*acknowledgement*) e mensagens COMMIT, como foi ilustrado no algoritmo *ComCan* apresentado); protocolos que retransmitem mensagens um número máximo de vezes (*bounded loops*); protocolos que eliminam duplicados; etc.

Estamos em crer que estes objectivos foram atingidos. No entanto, deve ser mencionado que a potência expressiva da declaração dos *triggers* é limitada e não pode capturar o comportamento de todos os protocolos. Por exemplo, protocolos multi-participante com um coordenador rotativo, onde a decisão de que nó reage a uma dada mensagem de difusão depende do estado da sessão, não pode ser expressa com

este mecanismo de *triggers*. No entanto extrair o modelo do código de um modo automático é uma tarefa árdua. Para além disso, o objectivo pretendido não é o de realizar uma validação completa de protocolos mas simplesmente capturar os padrões de execução para o pior caso. Isto alarga a aplicabilidade da ferramenta, apesar das suas limitações teóricas (em muitos casos a falta de detalhe só introduz um nível aceitável de pessimismo, na análise de escalonabilidade). Mesmo no exemplo de protocolos multi-participante, é possível analisar cada um dos estados possíveis separadamente e determinar qual o que corresponde ao pior caso. Como tal, pensamos que o *RT-Appia* fornece um compromisso pragmático entre complexidade e potência.

Uma preocupação relacionada com a nossa aproximação é a de que o programador deve realizar uma tarefa adicional (para especificar os *triggers*), de modo a permitir a extracção do diagrama de eventos. No entanto, é importante salientar que o "estado da arte"da maioria das ferramentas para validar automaticamente propriedades de protocolos, tais como o Kronos (Yovine, 1997) e UPPAAL (Amnell *et al.*, 2001), necessitam que um modelo do programa seja extraído automaticamente. De facto, ferramentas para derivar um modelo automaticamente a partir do código são um tópico activo de investigação (Ball & Rajamani, 2001; Dwyer *et al.*, 2001; Holzmann & Smith, 2002). Assim, o *RT-Appia* impõe um ónus ao programador que é comparável a outras aproximações existentes.

À primeira vista, pode parecer que a especificação dos *triggers* é quase tão difícil como a extracção do diagrama de eventos completo (já que a construção do diagrama dos *triggers* é de certo modo trivial). No entanto, é necessário reiterar que a especificação dos *triggers* é local a uma camada, ou seja, quando a programadora especifica os *triggers*, não precisa de ter conhecimento das outras camadas da pilha. Um dos desafios do trabalho apresentado foi o de projectar uma interface capaz de suportar esta independência. O facto desta interface ser realizável não era óbvio quando o trabalho foi iniciado. Este trabalho mostra que é possível atingir este objectivo. Por exemplo, no exemplo apresentado, os *triggers* para a camada CAN são especificados

somente para as mensagens DATA; mas isto não evita que as camadas acima tenham que definir novas sub-classes, tais como a mensagem COMMIT ou FRAG (e a plataforma é capaz de suportar a troca e ficar informado destas novas mensagens).

Finalmente, discutiremos como o *RT-Appia* ajuda a conservar a informação capturada pelos *triggers* em sincronismo com o código actual dos *handlers* dos eventos. Como já foi referido, uma das vantagens do *RT-Appia* é a de que a informação fornecida pelos *triggers* é também usada em tempo de execução. Em particular, o *RT-Appia* constrói automaticamente um conjunto de estruturas de dados de controlo que podem ser usadas em tempo de execução para verificar a conformidade da execução com a informação fornecida e capturada com os *triggers*. Por razões de desempenho, estas verificações em tempo de execução são opcionais, mas quando activadas, geram excepções quando é detectada uma inconformidade (por exemplo, se um gestor gera um evento que não consta em nenhum *trigger*). Deste modo, o *RT-Appia* oferece uma integração mais próxima entre o modelo e a implementação, especificamente se comparada com ferramentas onde o modelo é descrito numa linguagem diferente da linguagem de programação (tal como o Kronos e o UPPAAL).

5

Análise de Escalonabilidade

No capítulo anterior descrevemos o método utilizado para extrair o diagrama de eventos de uma composição de protocolos a partir do código fonte para a moldura *RT-Appia*. O diagrama de eventos descreve as relações causais entre os eventos gerados pelas camadas que caracterizam uma configuração concreta de um canal de tempo-real (numa dada composição de protocolos). Neste capítulo, partindo do diagrama de eventos, iremos calcular o pior tempo de resposta de uma composição de protocolos, utilizando as técnicas de análise introduzidas no Capítulo 3.

Antes de utilizar o diagrama de eventos para calcular o tempo de resposta de uma composição de protocolos, é necessário em primeiro lugar caracterizar o modelo computacional onde os protocolos irão ser executados. Este modelo, que se descreve de seguida, é definido pela arquitectura do sistema e também pelo modelo de execução do *RT-Appia*.

5.1 Modelo Computacional

Considera-se que o sistema a analisar possui uma arquitectura distribuída com vários processadores interligados por uma ou múltiplas redes de comunicação de tempo-real. Cada processador utiliza um escalonador de tarefas preemptivo com prioridades fixas, fornecido pelo sistema operativo ou por um núcleo de multi-programação. Deste modo é possível utilizar as técnicas de análise RM descritas

no Capítulo 3. Supõe-se também que a atribuição das tarefas aos processadores é estática. A mesma hipótese é feita em relação à associação das mensagens às redes de comunicação (no caso de existir mais que uma rede). Assume-se ainda que a troca de mensagens numa rede pode ser analisada utilizando técnicas semelhantes às utilizadas para calcular os piores tempos de resposta das tarefas em processadores.

Um canal de tempo-real é responsável pela troca de mensagens entre dois ou mais processadores. A cada canal é atribuída uma prioridade, que é fixa e igual em todos os processadores onde o canal é executado. Em cada processador onde o canal se projecta é criada uma tarefa de tempo-real que é responsável pela execução de toda a actividade num canal. Esta tarefa é escalonada localmente por um sistema operativo ou por um núcleo de multiprogramação. Cada tarefa processa uma sequência de gestores de eventos (capturada pelo diagrama de eventos) e a sua prioridade reflecte a prioridade do canal.

As tarefas são activadas por eventos externos, tipicamente pela chegada de uma mensagem ou pelo expirar de um temporizador[1]. Todos os eventos que fluem num canal são mantidos numa fila de *eventos escalonáveis* com uma disciplina *FIFO*. O escalonador de eventos selecciona sempre o primeiro evento da fila e entrega-o à sessão de destino, de acordo com a tabela de encaminhamento do próprio evento.

O diagrama de eventos captura uma ordem parcial de eventos, mesmo para eventos processados pela mesma tarefa no mesmo nó. No entanto, o modelo de execução acima descrito implica que, na realidade, todos os eventos são processados sequencialmente em cada nó, respeitando a ordenação FIFO na comunicação entre os canais (ou seja, uma ordem total). Posteriormente, discute-se como esta informação acerca do ambiente de execução pode ser utilizada para simplificar a análise de escalonabilidade.

[1]Os temporizadores são frequentemente utilizados para detectar e recuperar de omissões na rede.

5.2 Um Exemplo

Um diagrama de eventos descreve a resposta de um canal de tempo-real, sob a forma de uma sequência de gestores de eventos executados num sistema distribuído. Cada resposta é activada por um único evento externo (normalmente um pedido do utilizador ou um temporizador) que é representado pelo evento raiz do diagrama. Por sua vez, cada gestor é activado por um evento gerado por um gestor precedente. Um gestor pode gerar zero ou mais eventos , permitindo activar outros gestores, mas cada gestor só pode ser activado por um único evento. Os eventos podem activar gestores no mesmo processador ou em processadores diferentes; neste último caso, a activação do gestor remoto é despoletada pela recepção de uma mensagem transmitida através da rede de comunicações. Cada evento externo é caracterizado por um período e uma meta, que são conhecidos à priori. O período deste evento constitui a base temporal de uma resposta. Todos os eventos seus sucessores (gerados por uma sequência causal de execução de gestores) partilham o mesmo período, ou um múltiplo inteiro desse período.

Para ilustrar o processo e os resultados da análise de escalonabilidade, iremos recorrer a um exemplo que tem por base a pilha de protocolos utilizada no Capítulo 4 para ilustrar a captura do diagrama de eventos. Para tornar a pilha um pouco mais complexa, e em consequência, mais interessante de analisar, adicionou-se uma camada de fragmentação, a *FragCan*, entre a camada de *Aplicação* e a camada *ComCan*. Esta pilha encontra-se ilustrada na Figura 5.1.

O diagrama de eventos correspondente a esta pilha está representada na Figura 5.2. Naturalmente, a cadeia de eventos que descreve a transmissão de cada fragmento pela camada *ComCan* (descrita agora com o evento do tipo FragCan) é semelhante à que foi descrita no Capítulo 4. Por razões de legibilidade do diagrama, não se representam todos os eventos relacionados com o envio do segundo fragmento da mensagem, uma vez que são idênticos aos do primeiro fragmento. Por este motivo, o envio do

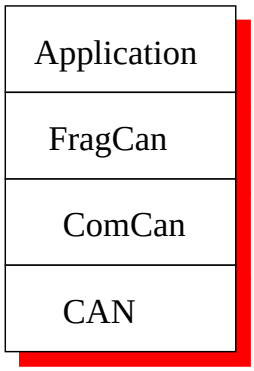

Figura 5.1: Uma pilha de protocolos com quatro camadas.

segundo fragmento está representado pelo evento $E19$ com uma ligação a tracejado largo ao evento $E3$. O evento $E29$, também com uma ligação a tracejado largo, indica a recepção da mensagem pela camada de aplicação após ter sido recebido o segundo fragmento na camada *FragCan*. Os pedidos da camada de aplicação são modelados por um temporizador com um período T.

5.3 Parâmetros

Definida a arquitectura do sistema a analisar, o passo seguinte consiste em determinar os parâmetros que é necessário a fornecer à ferramenta de análise de escalonabilidade para cálculo do pior tempo de resposta de cada um dos canais. Estes parâmetros são os seguintes:

- O conjunto de tarefas e o conjunto de processadores onde estas irão ser executadas.
- O período, o *Tempo de Computação para o Pior Caso* (*TCPC*) e a prioridade de cada tarefa.
- A meta de cada tarefa. Embora este parâmetro possa ser opcional, pois, uma vez calculado o tempo de resposta, basta compará-lo com uma meta para determinar

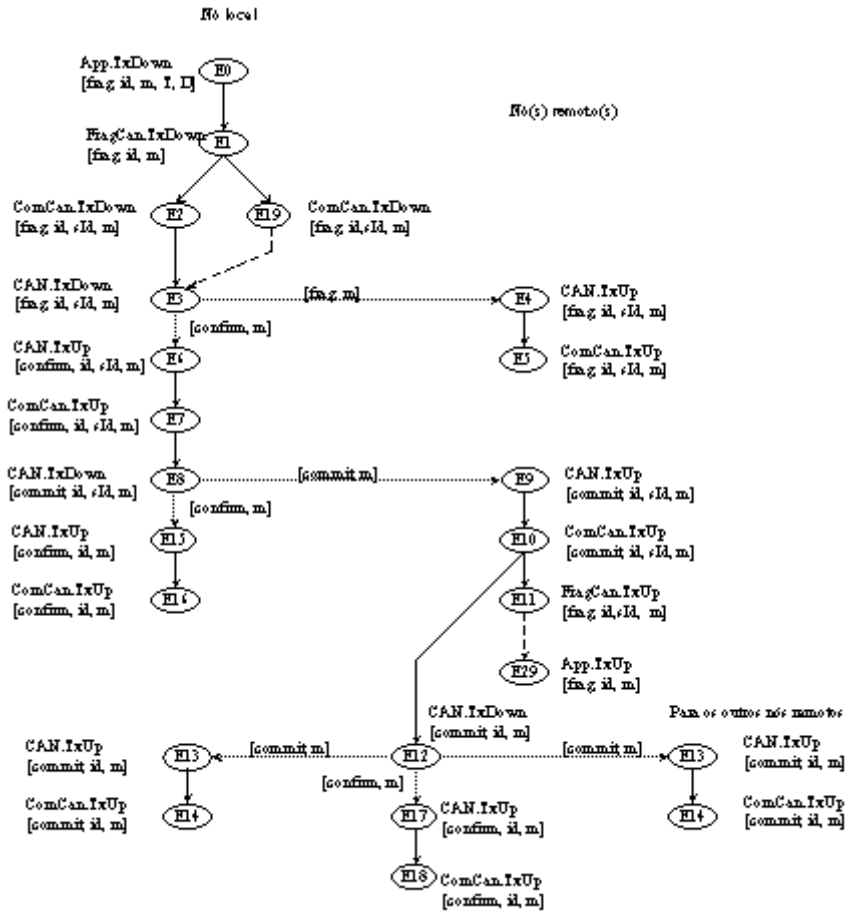

Figura 5.2: Diagrama de eventos para o FragCan

se o sistema é ou não escalonável.

- O tempo de bloqueio e o *release jitter* de cada tarefa (se existirem).
- O conjunto das mensagens trocadas entre os nós e a dimensão de cada mensagem.
- Os efeitos do escalonador.
- O modelo da rede de comunicações e seus parâmetros.

5.3.1 Conjunto de Tarefas e de Processadores

Com base no diagrama de eventos (extraído a partir da configuração do canal) e no número de processadores definidos para cada canal é possível identificar o conjunto de tarefas a fornecer à ferramenta de análise de escalonabilidade, assim como o número de mensagens trocadas entre os nós. Tanto o conjunto de tarefas como o número de mensagens dependem do número de processadores que forem definidos para cada canal.

Uma aproximação possível consiste em associar uma tarefa ao processamento de cada evento gerado por cada camada de protocolo. Deste modo, a execução de cada gestor, representado por cada nó do diagrama de eventos, é modelada por uma tarefa no modelo de análise. No entanto, desta aproximação pode resultar um número elevado de tarefas a analisar, sendo necessário um tempo de computação maior para calcular os piores tempos de resposta e um ligeiro aumento do pessimismo introduzido pelo modelo de análise. É possível reduzir o número de tarefas considerando a disciplina *FIFO* do escalonador de eventos de um canal e associando uma única tarefa à execução sequencial de um conjunto de gestores visitados pelo mesmo evento e no mesmo processador. Uma vez que cada gestor declara o *TCPC* de cada evento é fácil calcular o *TCPC* de uma tarefa que executa n gestores consecutivos da sua tabela de encaminhamento:

$$C_{chain} = \sum_{i=0}^{n} C_{H_i} + (n-1).C_{Sch} + C^i_{Sch} + C_{H_{channel}} \tag{5.1}$$

Onde C_{H_i} é o *TCPC* do gestor da sessão i e C_{Sch} o tempo gasto pelo escalonador de eventos. Este tempo é igual ao tempo gasto pelo escalonador de eventos a remover o primeiro evento da fila de eventos escalonáveis (C_{SchRem}) mais o tempo gasto a inserir o evento na fila de eventos (C_{SchIn}) e que corresponde à invocação do método `go`, (subsecção 4.2.4):

$$C_{Sch} = C_{SchRem} + C_{SchIn} \tag{5.2}$$

O termo $C_{H_{channel}}$ é o tempo gasto a devolver o evento ao canal após este ter executado o último gestor da sua tabela de encaminhamento. O termo C^i_{Sch} permite incluir a sobrecarga introduzida a processar novos eventos (temporizadores ou mensagens recebidas da rede de comunicações) activados na interface de um canal. Para os eventos gerados durante a execução de um gestor, C^i_{Sch} tem um valor igual a C_{Sch} , pois estes são inseridos directamente na lista de *eventos escalonáveis*.

Quando não é possível reduzir o número de tarefas a analisar, esta aproximação tende, no caso limite, a associar uma tarefa por gestor. No *RT-Appia* é possível analisar os tempos de resposta de um conjunto de canais utilizando qualquer das aproximações.

Outra questão que afecta a análise de escalonabilidade está relacionada com o modo de funcionamento suportado pela sessão CAN. Assume-se um modo de funcionamento assíncrono da sessão CAN, caso contrário seria necessário utilizar um servidor periódico para tratar a recepção de mensagens.

5.3.2 Tempo de Computação para o Pior Caso

Este parâmetro tem influência directa no cálculo dos piores tempos de resposta das tarefas. Cada tarefa é caracterizada por um *TCPC* dado pelo respectivo gestor, que está directamente relacionado com a arquitectura do ambiente de execução onde os protocolos são executados (mais precisamente com o número de instruções máquina, a frequência de relógio e a arquitectura do microprocessador).

A extracção do *TCPC* directamente a partir do código fonte da linguagem de programação, constitui um tópico de investigação activo (Puschner & Burns, 1999). Diversas aproximações têm sido propostas, entre as quais salientamos as seguintes: extensões a linguagens de programação como o RT-Euclid (Kligerman & Stoyenko, 1986) (Stoyenko, 1987) ou o RTC++ (Ishikawa *et al.*, 1992), entre outras; linguagens de programação dedicadas como o SPARK[2] (Chapman *et al.*, 1996) (Chapman *et al.*, 1994); compiladores interactivos de linguagens tradicionais (Park & Shaw, 1991), até ambientes de programação dedicados, incluindo compilador (Puschner & Schedl, 1993); analisador do *TCPC* (Puschner & Schedl, 1997) e editor (Pospischil *et al.*, 1992).

Seria interessante incluir no *RT-Appia* uma ferramenta capaz de extrair o *TCPC* a partir da análise do código fonte. No entanto, esta tarefa está fora do âmbito desta dissertação. Deste modo, na versão do *RT-Appia* aqui descrita, o programador deve fornecer explicitamente este valor para cada gestor (veja-se a interface do *RT-Appia* no capítulo anterior). Para obter este valor, o programador poderá usar uma ferramenta especializada desenvolvida por terceiros.

Da mesma forma, é possível estabelecer um *TCPC* para o tempo gasto pelo escalonador de eventos. O tempo gasto a inserir e a remover eventos da lista de eventos escalonáveis, C_{Sch}, pode-se considerar constante. O único parâmetro variável corresponde ao número máximo de novos eventos que possam ser activados na interface de um canal, num determinado intervalo de tempo (termo C^i_{Sch} na equação 5.1) e que cor-

[2]Baseado num subconjunto da sintaxe do Ada 83.

tarefa	*TCPC*	*tarefa*	*TCPC*	*tarefa*	*TCPC*	*tarefa*	*TCPC*
Frag_L3_D	20 + 29	Frag_L2_D	80	Frag_L1_D	33	Frag_L0_D	41 + 13
Frag_L0_U	43 + 29	Frag_L1_U	34	Frag_L2_U	37 + 13	Frag_L3_U	29 + 13
Comm_L0_D	41 + 13	Comm_L0_U	43 + 29	Comm_L1_U	34 + 13	Conf_L0_U	18 + 29
Conf_L1_U	34 + 13	-	-	-	-	-	-

Tabela 5.1: Conjunto de tarefas e respectivos *TCPC* (em μs).

respondem a temporizadores ou mensagens recebidas da rede de comunicações. Este parâmetro depende da aproximação utilizada para o escalonamento de eventos, num canal e será abordada com mais pormenor no capítulo 7.

Cada gestor pode gerar mais do que um evento mas é activado por um único evento. Recordamos que o *RT-Appia* permite associar a geração de um evento a uma sequência de invocações do mesmo gestor para modelar processos como a fragmentação e reconstrução de mensagens (ver a discussão na Subsecção 4.2.5). Na prática, cada invocação do gestor nesta sequência poderá possuir um tempo de execução ligeiramente diferente (por exemplo, na reconstrução de uma mensagem a partir de vários segmentos, o processamento do último segmento poderá exigir mais passos de computação). No entanto, o modelo de composição do *RT-Appia* obriga a que se considere sempre o *TCPC* para todas as instâncias, introduzindo algum pessimismo na análise.

A Tabela 5.1 apresenta, para a composição da Secção 5.2, os *TCPC*s associados a cada um dos gestores representados no diagrama de eventos. Cada gestor é identificado pelo nome do evento, a camada ($L0$ indica a camada CAN, $L1$ a camada imediatamente acima, etc...) e a direcção do evento processado ($D = Down$ ou $U = Up$). Para além do *TCPC* de cada gestor estão também indicados (com um sinal $+$) os tempos C^i_{Sch} e $C_{H_{channel}}$ referidos na secção anterior. Foram estabelecidos os seguintes tempos: $C_{H_{channel}} = 13\mu s$ e $C^i_{H_{Sch}} = 29\mu s$. O tempo C_{Sch} está incluído no *TCPC* do gestor. Por exemplo o *TCPC* do gestor $Frag\_L3\_D$ inclui o tempo gasto pelo escalonador a processar o temporizador activado na interface do canal ($C^i_{H_{Sch}}$) e o do gestor $Frag\_L0\_D$, o tempo gasto a entregar o evento ao canal ($C_{H_{channel}}$) por corresponder à invocação do

gestor da última sessão da sua tabela de encaminhamento.

Como já foi referido, a definição de um único *TCPC* para diferentes invocações do mesmo gestor, introduz algum pessimismo. Por exemplo o *TCPC* do gestor $Frag\_L2\_U$ inclui o tempo $C_{H_{channel}}$ gasto a devolver ao canal o evento recebido com o primeiro fragmento da mensagem. No entanto, o evento com o segundo fragmento só irá ser devolvido ao canal pela camada de aplicação após ser reconstruída a mensagem (o *TCPC* do gestor $Frag\_L3\_U$ também inclui $C_{H_{channel}}$), ficando o seu *TCPC* afectado com uma sobrecarga adicional de $C_{H_{channel}}$.

A Tabela 5.2 apresenta o conjunto de tarefas extraídas do diagrama de eventos aplicando a equação 5.1 aos *TCPCs*, apresentados na Tabela 5.1, para um canal estabelecido entre o nó $N1$ e o nó $N2$. Cada tarefa está identificada pelo nome do evento, seguido do nó onde é executada ($N1$ ou $N2$) e um identificador que é atribuído para evitar nomes ambíguos. A tarefa $Frag\_N1\_0$ corresponde ao envio do primeiro fragmento da mensagem e resulta da aplicação da equação 5.1 aos *TCPCs* dos gestores $Frag\_L3\_D$, $Frag\_L2\_D$, $Frag\_L1\_D$ e $Frag\_L0\_D$. A tarefa $Frag\_N2\_1$ corresponde à recepção do primeiro fragmento e resulta da execução dos gestores $Frag\_L0\_U$ e $Frag\_L1\_U$. O envio do segundo fragmento é modelado pela tarefa $Frag\_N1\_2$, que inclui a execução dos gestores $Frag\_L1\_D$ e $Frag\_L0\_D$ e a tarefa $Frag\_N2\_3$ modela a recepção do segundo fragmento, incluindo a entrega da mensagem original à camada de aplicação, e corresponde à execução dos gestores $Frag\_L2\_U$ e $Frag\_L3\_U$. As tarefas definidas para o evento $DataCnfEvent$ correspondem à execução dos gestores $Conf\_L0\_U$ e $Conf\_L1\_U$. Finalmente as tarefas $Comm\_N1\_5$, $Comm\_N1\_9$, $Comm\_N2\_13$ e $Comm\_N2\_16$ correspondem à execução do gestor $Comm\_L0\_D$ e as tarefas $Comm\_N2\_6$, $Comm\_N2\_10$, $Comm\_N1\_14$ e $Comm\_N1\_17$ à execução dos gestores $Comm\_L0\_U$ e $Comm\_L1\_U$. Como foi referido no Capítulo 4 a camada *CAN* não distingue os eventos CommitEvent dos eventos DataTxFrag, pelo que os *TCPCs* dos gestores $Comm\_L0\_D$ e $Comm\_L0\_U$ são os mesmos dos gestores $Frag\_L0\_D$ e $Frag\_L0\_U$, respectivamente.

tarefa	*TCPC*	*tarefa*	*TCPC*	*tarefa*	*TCPC*	*tarefa*	*TCPC*
Frag_N1_0	216	Frag_N2_1	106	Frag_N1_2	87	Frag_N2_3	106
Conf_N1_4	94	Comm_N1_5	54	Comm_N2_6	119	Frag_N2_7	50
Conf_N1_8	94	Comm_N2_13	54	Comm_N1_9	54	Conf_N2_18	94
Comm_N2_10	119	Conf_N1_12	94	Comm_N1_14	119	Frag_N2_11	92
Comm_N2_16	54	Conf_N1_15	94	Conf_N2_19	94	Comm_N1_17	119

Tabela 5.2: Conjunto de tarefas e respectivos *TCPC*.

5.3.3 Tempo de Bloqueio de Sessões Partilhadas

Considera-se cada sessão partilhada como uma secção crítica controlada pelo protocolo de Herança de Prioridade ou pelo protocolo de Tecto de Prioridade. Para calcular o factor de bloqueio B_i para as tarefas que executam sessões partilhadas, assume-se que cada sessão se comporta como um recurso binário e que não existem execuções encadeadas (as sessões são executados em sequência, uma de cada vez). Deste modo não podem ocorrer bloqueios encadeados nem interbloqueio *TCPC* (como foi descrito no Capítulo 3 os bloqueios encadeados e o interbloqueio são problemas não resolvidos no protocolo de Herança de Prioridade). Estas hipóteses simplificam o algoritmo para calcular o tempo máximo de bloqueio (B_i) para as tarefas que executam sessões multi-canal.

Para calcular B_i é necessária a seguinte informação: *i)* o conjunto de sessões a partilhadas; *ii)* o conjunto de tarefas que executam sessões partilhadas e as suas prioridades e; *iii)* o *TCPC*, para cada tarefa em cada gestor de eventos da sessão.

A informação acerca de uma sessão partilhada pode ser extraída automaticamente, baseada no conjunto de canais criados. Se uma sessão é partilhada por dois canais de prioridades diferentes, tem um tecto de prioridade $C(S_k)$ igual à prioridade da tarefa (canal) de maior prioridade que partilha a sessão. De acordo com a condição de bloqueio dos protocolos de acesso a recursos, uma tarefa t_i (com prioridade P_i) só pode ser bloqueada numa secção crítica executada por uma tarefa t_j (com prioridade P_j) se $P_j \leq P_i$ e $C(S_k) \geq P_i$. Em sessões partilhadas esta condição de bloqueio simplifica-se para $P_j \leq P_i$, porque o tecto de prioridade da sessão $C(S_k)$ é sempre maior ou igual à

prioridade de todas as tarefas e a condição $C(S_k) \geq P_i$ verifica-se sempre.

Deste modo o tempo de bloqueio de uma sessão partilhada é simplesmente o máximo dos *TCPC* de todos os eventos que executam essa sessão. Isto verifica-se porque qualquer evento de um canal de menor prioridade pode bloquear um evento de um canal de maior prioridade, independentemente da prioridade relativa dos eventos que fluem dentro do canal. Daqui resulta também que, independentemente do protocolo utilizado (ver Teoremas 1 e 2 na Subsecção 3.2), a execução de um canal pode ser bloqueada no máximo uma secção critica. Por exemplo, da Tabela 5.1 extrai-se que o tempo de bloqueio para a sessão CAN partilhada é igual ao *TCPC* do gestor $Frag\_L0\_U$, que é de $72\mu s$.

5.3.4 Parâmetros da Rede de Comunicações

O modelo de análise para o cálculo de pior tempo de resposta das mensagens transmitidas na rede de comunicações deve incluir o cálculo do tempo necessário para uma mensagem ganhar acesso ao meio físico e o tempo de transmissão da mensagem na rede. Os parâmetros a fornecer dependem do protocolo de acesso ao meio.

Neste trabalho foi utilizado um barramento de campo CAN 2.0A cujo modelo de análise foi descrito no Capítulo 3. Os parâmetros a fornecer consistem na dimensão das mensagens a transmitir, no tipo de rede (CAN 2.0A ou CAN 2.0B) e na largura de banda utilizada. Estes parâmetros são obtidos do diagrama de eventos e permitem determinar o tempo de transmissão de uma mensagem na rede. Para além destes parâmetros, é necessário fornecer também o período de cada mensagem que, como já foi referido, é dado pelo evento raiz do diagrama.

Recorrendo de novo ao exemplo da secção 5.2, e a partir da Equação 3.24 (ver Capítulo 3) obtém-se um tempo de transmissão para o pior caso de $130\mu s$, para uma mensagem FRAG com 8 octetos de dados e de $53\mu s$, para uma mensagem COMMIT com 0 octetos. Deste modo as mensagens de um canal de maior prioridade sofrem um

tempo de bloqueio de $130\mu s$.

5.3.5 O Pior Tempo de Entrega de Mensagens

O cálculo dos tempos de acesso e transmissão de uma mensagem permitem determinar o instante em que uma mensagem chega ao controlador da rede no nó destino. É também necessário calcular o tempo que o gestor do periférico gasta para entregar a mensagem ao respectivo canal. Este tempo depende da aproximação seguida pelo gestor do periférico. Numa aproximação "a pedido" (Tindell *et al.*, 1991), o periférico activa uma interrupção no processador central e a rotina de atendimento da interrupção sinaliza uma tarefa do sistema operativo que, por sua vez, copia os dados do tampão do controlador de rede e realiza as operações necessárias à entrega das mensagens ao respectivo canal. O pior tempo de entrega (D_H) pode ser calculado através da seguinte equação (Tindell *et al.*, 1991):

$$D_H = C_H + I_H + K_H \tag{5.3}$$

onde C_H é o *TCPC* da tarefa de entrega, incluindo os custos de mudanças de contexto e o tempo gasto a copiar os dados do tampão do controlador de rede, I_H a interferência que a tarefa pode sofrer devido a tarefas de maior prioridade que possam ficar activas (em princípio, esta tarefa terá uma prioridade elevada, pelo que não deverão existir muitas tarefa com prioridade superior) e também a interferência provocada por outras rotinas de atendimento de interrupções (incluindo as que são provocadas pela chegada de novas mensagens). Para um cálculo mais preciso é necessário também considerar a interferência devida à execução de invocações prévias da tarefa, que possam estar pendentes devido à chegada de mensagens anteriores. A interferência I_H é máxima quando o tempo entre a chegada de duas quaisquer mensagens é limitado a um valor mínimo. O termo K_H representa o

máximo tempo de bloqueio que a tarefa de entrega pode sofrer no acesso a secções critícas.

Para calcular o pior tempo de entrega é necessário um conhecimento pormenorizado da concretização do gestor do periférico e do sistema operativo utilizado. Por vezes, os fabricantes de dispositivos de rede e de sistemas operativos comerciais não fornecem pormenores suficientes sobre a sua concretização, tornando-se impossível extrair os parâmetros necessários para o cálculo do tempo de entrega. Na ausência desta informação, o tempo de entrega pode ser estimado através da medição directa da concretização na arquitectura final.

5.3.6 Verificação das Metas

Uma vez obtidos os tempos de resposta de todas as tarefas, é possível obter o pior tempo de resposta de um canal, que é dado pela tarefa do canal com o pior tempo de resposta calculado. Para verificar a escalonabilidade do sistema basta comparar este tempo com a meta fornecida pelo evento raiz do diagrama.

5.3.7 Prioridades das Tarefas

As prioridades das tarefas extraídas do diagrama de eventos são atribuídas com base na prioridade do canal a que pertencem e na ordem de execução estabelecida. A atribuição de prioridades garante que qualquer tarefa de um canal de menor prioridade pode sofrer preempção de uma tarefa de um canal com maior prioridade. Todas as mensagens trocadas entre as tarefas de um canal possuem uma prioridade dada pela prioridade do canal.

5.3.8 Os Efeitos do Escalonador

A sobrecarga introduzida pelo escalonador do sistema operativo na comutação de tarefas associadas aos diversos canais influencia também os tempos de resposta dos diversos canais. Cada evento pode ainda sofrer um *release jitter* introduzido pelo escalonador na comutação de tarefas associadas aos diversos canais. Este efeito afecta principalmente os eventos externos, temporizadores e eventos associados à recepção de mensagens.

Devido ao caracter assíncrono do escalonador de eventos de um canal, poderá acontecer que um evento associado a um temporizador ou a uma mensagem recebida da rede de comunicações, seja activado durante a execução de um gestor. Esse(s) evento(s) poderão ser detectados e inseridos na fila de eventos escalonáveis no fim da execução do gestor corrente ou quando o escalonador de eventos tiver processado todos os eventos que se encontrem na lista de eventos escalonáveis, introduzindo um *release jitter* variável no processamento desses eventos.

5.4 Cálculo dos Piores Tempos de Resposta

Uma vez determinado o conjunto de tarefas e mensagens, assim como os respectivos parâmetros, é possível calcular os piores tempos de resposta para um conjunto de canais de tempo-real utilizando os métodos de análise descritos no Capítulo 3.

Iremos de seguida ilustrar o resultado da análise obtidos com cada um dos modelos, usando os canais de comunicação representados na Figura 5.3. Nesta figura estão representados dois canais com a configuração da pilha de protocolos FragCan, numa arquitectura com apenas dois nós interligados com um barramento *CAN 2.0A* com $1Mbps$. A camada CAN é partilhada pelos dois canais. O canal $RC0$ transmite pedidos do nó 1 para o nó 2, com um período $T = 10ms$ e possui uma prioridade 20 e o canal $RC1$ transmite pedidos do nó 2 para o nó 1, com um período $T = 20ms$ e possui uma

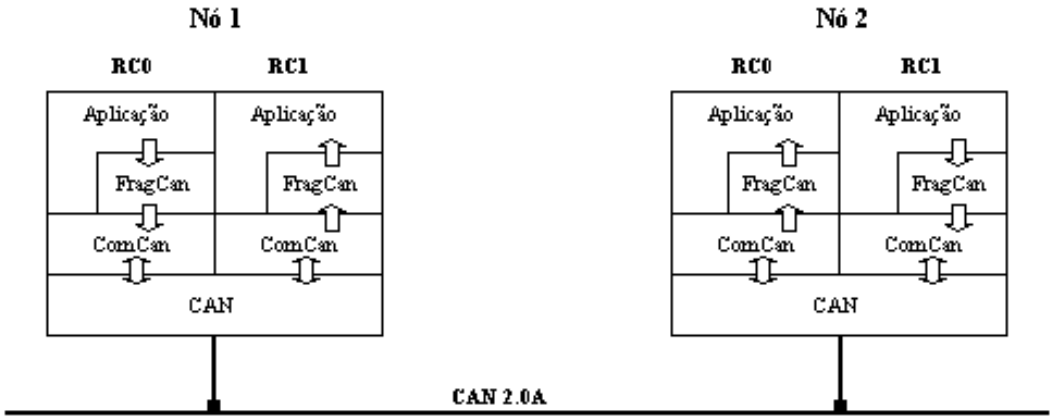

Figura 5.3: Pilha de protocolos do FragCan

prioridade de 60. Independentemente do modelo de análise utilizado, considera-se que o conjunto de tarefas e mensagens extraídas do diagrama de eventos compõem uma transacção.

O *RT-Appia* suporta a análise usando diferentes modelos de cálculo do pior tempo de resposta. O desenvolvimento de diferentes variantes da ferramenta de análise permitiu aferir qual o mais adequado para aplicar a canais de comunicação, como os suportados pelo *RT-Appia*. Nomeadamente, o *RT-Appia* suporta a análise de acordo com os seguintes modelos:

- O modelo holístico proposto por Tindell (Tindell *et al.*, 1992b);
- O modelo dos desfasamentos dinâmicos para sistemas distribuídos proposta por Palencia e Harbour (Palencia & Harbour, 1998);
- A adaptação do modelo dos desfasamentos estáticos para sistemas distribuídos proposto por Ventura (Ventura, 2001).

Como foi referido no capítulo 3, as aproximações seguidas pelos modelos de análise de escalonabilidade para sistemas distribuídos baseiam-se na *herança de atributos* entre tarefas executadas em processadores diferentes: *release jitter*, *desfasamento* ou

ambos. No entanto uma transacção pode incluir tarefas a serem executadas no processador onde se executa uma tarefa receptora de uma mensagem e que com esta possuem relações de precedência. Para estes casos, é também necessário actualizar os atributos de todas essas tarefas, pois estas podem gerar novas mensagens, indo afectar os tempos de resposta das tarefas suas sucessoras. É exemplo disso o presente conjunto de tarefas que se pretende analisar. Para descrever a aproximação seguida, designa-se por *subtransacção* uma sequência de tarefas de uma transacção executadas no mesmo processador e activadas por um evento externo ou por um evento interno gerado por outra *subtransacção*. Por exemplo, as tarefas $Conf\_N1\_4$ e $Comm\_N1\_5$ da Tabela 5.2 (que correspondem aos eventos $E6$, $E7$ e $E8$ do diagrama de eventos) definem uma *subtransacção* iniciada pela recepção do evento CONFIRM; cada mensagem transmitida também define uma *subtransacção*. É importante referir que tendo em consideração o modelo de execução do *RT-Appia*, os gestores que compõem uma *subtransacção* são executados em série por uma única tarefa.

5.4.1 Tempo de resposta para um único canal

É possível construir manualmente um diagrama temporal que descreva a execução do conjunto de tarefas de um único canal (e de uma única transacção) estabelecido entre os dois nós da arquitectura apresentada na Figura 5.3. Este diagrama vai facilitar a leitura dos resultados obtidos com os modelos de análise de escalonabilidade, apresentados nas subsecções seguintes e obter uma referência dos tempos de resposta para o melhor caso. Considerando que as tarefas não sofrem bloqueios em secções criticas nem a interferência de tarefas de outras transacções, é possível construir o diagrama representado na Figura 5.4.

A execução de tarefas e mensagens são representadas por rectângulos. Cada rectângulo inclui o identificador da tarefa de acordo com o que está definido na Tabela 5.2, com excepção da indicação do nó (N1, N2) e as mensagens são identificadas com o nome da tarefa receptora precedido de um m. Os algarismos indicam os $TCPC$.

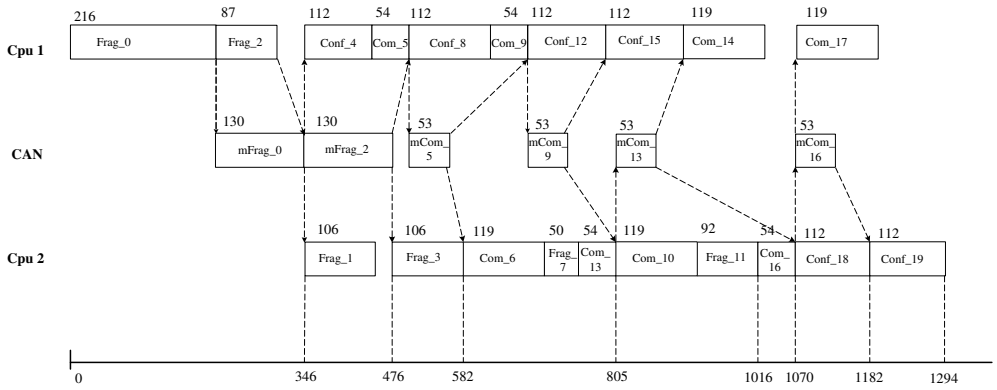

Figura 5.4: Diagrama de execução para as tarefas de uma transacção.

As setas indicam a relação entre as tarefas associadas à transmissão ou recepção de uma mensagem. Por exemplo a tarefa $Frag\_0$, representa a transmissão do primeiro fragmento e tem um $TCPC$ de $216\mu s$. A recepção da mensagem $mFrag\_0$ é executada pela tarefa $Frag\_1$ no $Cpu2$ e a respectiva confirmação dá origem à execução da tarefa $Conf\_4$ no $Cpu1$. Assume-se que o tempo para a indicação de uma confirmação é igual ao tempo de transmissão da mensagem.

Faz-se notar que a execução da tarefa Com_6 (que trata a recepção de $mCom\_6$) é atrasada devido à execução em curso do gestor $Frag\_3$. O mesmo acontece para a execução de Com_10, devido à execução em curso dos gestores Com_6, $Frag\_7$ e Com_13 associados à execução do evento $Commit$ assim como para a execução de $Conf\_18$ e de $Conf\_19$, no $Cpu2$ e de $Conf\_8$, $Conf\_12$ e $Conf\_15$ no $Cpu1$. Como se pode observar o tempo de resposta para a recepção da mensagem pela aplicação, no nó 2 é de $1016\mu s$ e a transacção tem um tempo de resposta de $1294\mu s$, dado pela tarefa $Conf\_19$. Como veremos adiante o cálculo dos tempos de resposta para o melhor caso recorrendo à Equação 5.4 é bastante mais optimista.

5.4.2 Modelo Holístico

O modelo holístico foi proposto por Tindell e Clark (Tindell *et al.*, 1992b) para determinar os tempos de resposta globais de tarefas e mensagens em sistemas distribuídos. Uma das dificuldades na utilização do modelo holístico é a de estabelecer as relações de precedência entre tarefas que são executadas no mesmo processador. Esta relação pode ser estabelecida considerando que as tarefas são libertadas em simultâneo e a ordem de execução é modelada através das prioridades atribuídas a cada tarefa.

Para calcular os tempos de resposta, é utilizada a ferramenta de análise desenvolvida por Tindell e Clark, substituindo o modelo computacional do protocolo TDMA, pelo modelo computacional para calcular a latência para o pior caso de uma mensagem de tempo-real estrito num barramento de campo CAN (Rodrigues *et al.*, 2002), apresentado na Subsecção 3.5.1 do Capítulo 3.

A aproximação seguida para calcular os piores tempos de resposta com o modelo holistíco foi a seguinte: considerou-se que uma tarefa destino de uma mensagem herda um *release jitter* dado pela Equação 3.27, definida no Capítulo 3; todas as tarefas da mesma *subtransacção* que a tarefa destino e localizadas no mesmo processador, herdam o mesmo *release jitter*; a ordem de execução destas tarefas (incluindo a da tarefa destino) é modelada através da prioridade atribuída a cada tarefa.

As Tabelas 5.3 e 5.4 apresentam os piores tempos de resposta (em μs) calculados para os dois canais, utilizando o modelo holístico e atribuindo uma tarefa à execução de uma sequência de gestores do diagrama. As tarefas do canal $RC0$ são modeladas pela transacção 1 e as do canal $RC1$ pela transacção 2. Não são considerados os efeitos do escalonador do sistema operativo, pelo que: $T_{clk} = 1$, $C_{clk} = 0$, $C_{QL} = 0$, $C_{QS} = 0$ e $J_i = 0$. As tarefas extraídas do diagrama são identificadas com o nome do evento, seguido do identificador da tarefa. A coluna C indica o *TCPC*, a coluna $prio$ a prioridade atribuída à tarefa e B o tempo de bloqueio calculado. Os valores herdados do *release jitter* são mostrados na coluna J.

Transacção 1 (Canal RC0)

tarefa	*cpu*	*prio*	*C*	*B*	*J*	*R*
Frag_0	cpu1	20	216	72	0	288
Frag_2	cpu1	21	87	72	0	375
Conf_4	cpu1	22	112	72	548	1035
Comm_5	cpu1	23	54	72	548	1089
Conf_8	cpu1	24	112	72	765	1418
Comm_9	cpu1	25	54	72	765	1472
Conf_12	cpu1	26	112	72	1532	2351
Conf_15	cpu1	27	112	72	2021	2952
Comm_14	cpu1	28	119	72	2535	3704
Comm_17	cpu1	29	119	72	3395	4445
Frag_1	cpu2	20	106	72	548	726
Frag_3	cpu2	21	106	72	765	1049
Comm_6	cpu2	22	119	72	1532	1935
Frag_7	cpu2	23	50	72	1532	1985
Comm_13	cpu2	24	54	72	1532	2039
Comm_10	cpu2	25	119	72	2021	2647
Frag_11	cpu2	26	92	72	2021	2739
Comm_16	cpu2	27	54	72	2021	2793
Conf_18	cpu2	28	112	72	2535	3419
Conf_19	cpu2	29	112	72	3395	4391
mensagem	*rede*	*Prio*	*C*	*B*	*J*	*R*
Frag_N1_1	can	20	130	130	288	548
Frag_N1_3	can	21	130	130	375	765
Comm_N1_6	can	23	53	130	1089	1532
Comm_N1_10	can	25	53	130	1472	2021
Comm_N2_14	can	24	53	130	2039	2535
Comm_N2_17	can	27	53	130	2793	3395

Tabela 5.3: Transacção 1 (modelo holístico).

Transacção 2 (Canal RC1)

tarefa	*cpu*	*prio*	*C*	*B*	*J*	*R*
Frag_0	cpu2	60	216	0	0	1140
Frag_2	cpu2	61	87	0	0	1227
Conf_4	cpu2	62	112	0	1742	3081
Comm_5	cpu2	63	54	0	1742	3135
Conf_8	cpu2	64	112	0	1959	3464
Comm_9	cpu2	65	54	0	1959	3518
Conf_12	cpu2	66	112	0	3920	5591
Conf_15	cpu2	67	112	0	4409	6192
Comm_14	cpu2	68	119	0	6290	8311
Comm_17	cpu2	69	119	0	7150	9052
Frag_1	cpu1	60	106	0	1742	2945
Frag_3	cpu1	61	106	0	1959	3268
Comm_6	cpu1	62	119	0	3920	5348
Frag_7	cpu1	63	50	0	3920	5398
Comm_13	cpu1	64	54	0	3920	5452
Comm_10	cpu1	65	119	0	4409	6060
Frag_11	cpu1	66	92	0	4409	6152
Comm_16	cpu1	67	54	0	4409	6206
Conf_18	cpu1	68	112	0	6290	8199
Conf_19	cpu1	69	112	0	7150	9171
mensagem	*rede*	*Prio*	*C*	*B*	*J*	*R*
Frag_N2_1	can	60	130	0	1140	1742
Frag_N2_3	can	61	130	0	1227	1959
Comm_N2_6	can	63	53	0	3135	3920
Comm_N2_10	can	65	53	0	3518	4409
Comm_N1_14	can	64	53	0	5452	6290
Comm_N1_17	can	67	53	0	6206	7150

Tabela 5.4: Transacção 2 (modelo holístico).

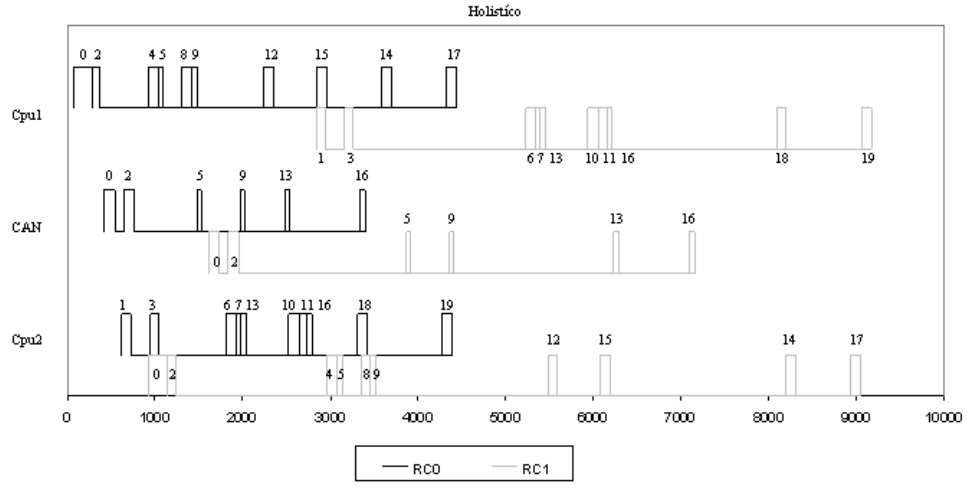

Figura 5.5: Diagrama de execução das tarefas com base nos tempos de resposta calculados pelo modelo holístico.

As mensagens são identificadas pelo nome do evento, o identificador do nó ($N1$ ou $N2$) e o identificador da tarefa que a transmite. A prioridade das mensagens é dada pela prioridade do canal a que pertencem pelo que se supõe que uma mensagem não sofre interferência das outras mensagens no mesmo canal.

A mensagem *Frag* (tarefa $Frag\_11$) é entregue à *Aplicação* com um tempo de resposta de $2739\mu s$, para o canal $RC0$, e a transacção tem um tempo de resposta de $4445\mu s$ (tarefa $Comm\_17$). No canal $RC1$ esses tempos são respectivamente de $6152\mu s$ e $9171\mu s$.

A Figura 5.5 apresenta um diagrama construído com os piores tempos de resposta calculados, para cada canal e em cada processador. O canal $RC0$ é representado com traço negro e o canal $RC1$ com traço cinzento. Cada tarefa está representada por um patamar cuja largura representa o $TCPC$ da tarefa, subtraído ao tempo de resposta calculado. As tarefas estão identificadas por um valor numérico que corresponde ao subscrito do respectivo identificador representado nas Tabelas 5.3 e 5.4.

5.4.3 Modelos dos Desfasamentos no Tempo

Conceptualmente, o modelo dos desfasamentos no tempo é o que melhor permite captar a resposta de um canal de tempo-real, descrito pelo diagrama de eventos. Todas as tarefas e mensagens de uma transacção são executadas com um desfasamento no tempo, relativo ao início de transacção.

É atribuído um desfasamento inicial nulo para cada tarefa que inicia uma *subtransacção*. Para as outras tarefas da *subtransacção* (se existirem) é atribuído um desfasamento inicial dado pelo tempo de resposta para o melhor caso da tarefa que a precede. O melhor tempo de resposta (R_i^b) de uma tarefa i é dado por (Palencia & Harbour, 1998):

$$R_i^b = C_i + \sum_{j=1}^{i-1} C_j \tag{5.4}$$

O melhor tempo de resposta é obtido considerando que uma tarefa i não sofre preempção de qualquer tarefa de maior prioridade e como tal o seu tempo de resposta é igual à soma do seu *TCPC* mais o *TCPC* de todas as tarefas que a precedem na transacção (Palencia & Harbour, 1998).

5.4.3.1 Modelo dos Desfasamentos Dinâmicos

Como se referiu, o *RT-Appia* suporta também o modelo de desfasamentos dinâmicos proposto por Palencia e Harbour (Palencia & Harbour, 1998) e descrito na subsecção 3.6.2. Para exemplificar os resultados obtidos com esta aproximação actualizou-se a ferramenta de análise desenvolvida por Tindell (Tindell, 1994), para incluir o calculo do *release jitter* equivalente.

Inicialmente são calculados os desfasamentos para o melhor caso das tarefas e mensagens de uma transacção, utilizando a Equação 5.4. A cada tarefa é atribuído um

Transacção 1 (Canal RC0)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*J*	*R*
Frag_0	cpu1	20	216	72	0	0	288
Frag_2	cpu1	21	87	72	216	0	375
Conf_4	cpu1	22	112	72	346	202	732
Comm_5	cpu1	23	54	72	458	202	786
Conf_8	cpu1	24	112	72	433	245	898
Comm_9	cpu1	25	54	72	545	245	952
Conf_12	cpu1	26	112	72	565	404	1153
Conf_15	cpu1	27	112	72	652	536	1372
Comm_14	cpu1	28	119	72	841	606	1757
Comm_17	cpu1	29	119	72	970	742	1903
Frag_1	cpu2	20	106	72	346	202	726
Frag_3	cpu2	21	106	72	433	245	856
Comm_6	cpu2	22	119	72	565	404	1160
Frag_7	cpu2	23	50	72	684	404	1210
Comm_13	cpu2	24	54	72	734	404	1264
Comm_10	cpu2	25	119	72	652	536	1383
Frag_11	cpu2	26	92	72	771	536	1475
Comm_16	cpu2	27	54	72	863	536	1529
Conf_18	cpu2	28	112	72	841	606	1641
Conf_19	cpu2	29	112	72	970	742	1896
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*J*	*R*
Frag_N1_0	can	20	130	130	216	72	548
Frag_N1_2	can	21	130	130	303	72	678
Comm_N1_5	can	23	53	130	512	274	969
Comm_N1_9	can	25	53	130	599	353	1188
Comm_N2_13	can	24	53	130	788	476	1447
Comm_N2_16	can	27	53	130	917	612	1712

Tabela 5.5: Transacção 1 (desfasamentos dinâmicos/*release jitter* equivalente)

desfasamento equivalente $O_i^{'} = R_{i-1}^{b}$ e um *release jitter* equivalente nulo ($J_i^{'} = 0$). Em cada ciclo do algoritmo, são calculados os piores tempos de resposta de todas as tarefas. De seguida são actualizados os valores do *release jitter* equivalente, dados pela Equação 3.28, para todas as tarefas de uma *subtransacção*.

As Tabelas 5.5 e 5.6 apresentam os piores tempos de resposta calculados para os dois canais, utilizando os desfasamentos dinâmicos onde todas as tarefas de uma *subtransacção* herdam o mesmo valor de *release jitter* equivalente e igual ao da primeira tarefa da *subtransacção*. A coluna O apresenta o desfasamento para cada tarefa e men-

Transacção 2 (Canal RC1)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*J*	*R*
Frag_0	cpu2	60	216	0	0	0	1140
Frag_2	cpu2	61	87	0	216	0	1227
Conf_4	cpu2	62	112	0	346	1290	2672
Comm_5	cpu2	63	54	0	458	1290	2726
Conf_8	cpu2	64	112	0	433	1439	2908
Comm_9	cpu2	65	54	0	545	1439	2962
Conf_12	cpu2	66	112	0	565	2580	4181
Conf_15	cpu2	67	112	0	652	2782	4470
Comm_14	cpu2	68	119	0	841	4043	6046
Comm_17	cpu2	69	119	0	970	4245	6258
Frag_1	cpu1	60	106	0	346	1290	2839
Frag_3	cpu1	61	106	0	433	1439	3075
Comm_6	cpu1	62	119	0	565	2580	4361
Frag_7	cpu1	63	50	0	684	2580	4411
Comm_13	cpu1	64	54	0	734	2580	4465
Comm_10	cpu1	65	119	0	652	2782	4650
Frag_11	cpu1	66	92	0	771	2782	4742
Comm_16	cpu1	67	54	0	863	2782	4796
Conf_18	cpu1	68	112	0	841	4043	6093
Conf_19	cpu1	69	112	0	970	4245	6424
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*J*	*R*
Frag_N2_0	can	60	130	0	216	924	1636
Frag_N2_2	can	61	130	0	303	924	1872
Comm_N2_5	can	63	53	0	512	2214	3145
Comm_N2_9	can	65	53	0	599	2363	3434
Comm_N1_13	can	64	53	0	788	3677	4884
Comm_N1_16	can	67	53	0	917	3879	5215

Tabela 5.6: Transacção 2 (desfasamentos dinâmicos/*release jitter* equivalente).

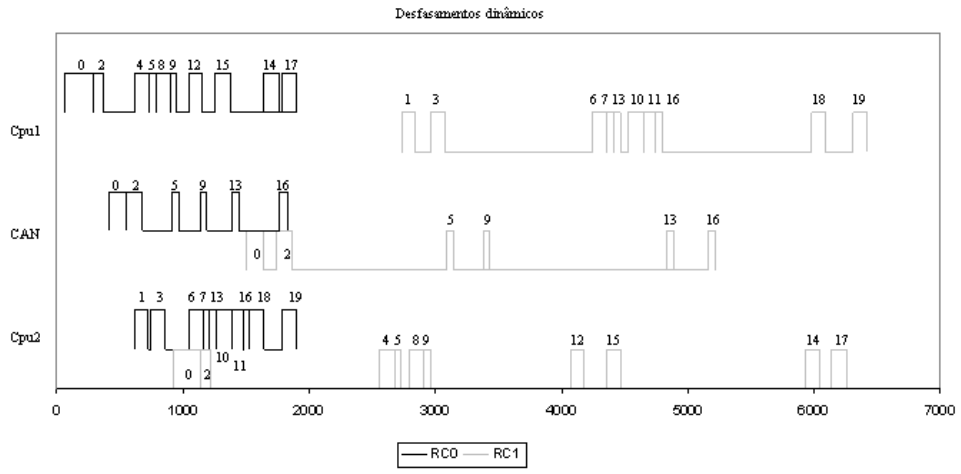

Figura 5.6: Diagrama de execução das tarefas com base nos tempos de resposta calculados pelo modelo dos desfasamentos dinâmicos.

sagem, calculados com base no melhor tempo de resposta, utilizando a Equação 5.4 e a coluna J mostra os valores do *release jitter* equivalente, calculados para cada *sub-transacção*. A recepção da mensagem *Frag* pela *Aplicação* no canal no canal $RC0$ (tarefa $Frag\_11$) tem um tempo de resposta de $1475\mu s$ e a transacção 1 um tempo de resposta de $1903\mu s$, dado pela tarefa $Comm\_17$. Para o canal $RC1$ estes tempos são respectivamente de $4742\mu s$ e $6424\mu s$, dado pela tarefa $Conf\_19$.

A Figura 5.6 apresenta o diagrama construído com os piores tempos de resposta calculados, para cada canal e em cada processador. O canal $RC0$ é representado com traço negro e o canal $RC1$ com traço cinzento. Cada tarefa está representada por um patamar cuja largura representa o $TCPC$ da tarefa, subtraído ao tempo de resposta calculado. As tarefas estão identificadas por um valor numérico que corresponde ao subscrito do respectivo identificador representado nas Tabelas 5.5 e 5.6.

Como se pode observar nas Tabelas 5.5 e 5.6, os valores obtidos para o melhor tempo de resposta utilizando a Equação 5.4 são mais optimistas do que os valores apresentados no diagrama da Figura 5.4. Isto leva a que os valores calculados para o *release jitter* equivalente, sejam maiores, introduzindo maior pessimismo no cálculo dos tempos de resposta para os canais de menor prioridade. Isto pode ser comprovado

Transacção 1 (Canal RC0)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*J*	*R*
Frag_0	cpu1	20	216	72	0	0	288
Frag_2	cpu1	21	87	72	216	0	375
Conf_4	cpu1	22	112	72	346	202	732
Comm_5	cpu1	23	54	72	458	202	786
Conf_8	cpu1	24	112	72	512	166	898
Comm_9	cpu1	25	54	72	624	166	952
Conf_12	cpu1	26	112	72	678	291	1153
Conf_15	cpu1	27	112	72	790	398	1372
Comm_14	cpu1	28	119	72	902	545	1757
Comm_17	cpu1	29	119	72	1123	589	1903
Frag_1	cpu2	20	106	72	346	202	726
Frag_3	cpu2	21	106	72	476	202	856
Comm_6	cpu2	22	119	72	582	387	1160
Frag_7	cpu2	23	50	72	701	387	1210
Comm_13	cpu2	24	54	72	751	387	1264
Comm_10	cpu2	25	119	72	805	383	1383
Frag_11	cpu2	26	92	72	924	383	1475
Comm_16	cpu2	27	54	72	1016	383	1529
Conf_18	cpu2	28	112	72	1070	377	1641
Conf_19	cpu2	29	112	72	1182	530	1896
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*J*	*R*
Frag_N1_0	can	20	130	130	216	72	548
Frag_N1_2	can	21	130	130	346	29	678
Comm_N1_5	can	23	53	130	512	274	969
Comm_N1_9	can	25	53	130	678	274	1188
Comm_N2_13	can	24	53	130	805	459	1447
Comm_N2_16	can	27	53	130	1070	459	1712

Tabela 5.7: Transacção 1 (desfasamentos dinâmicos/*release jitter* equivalente) com o novo cálculo dos melhores tempos de resposta

pelos resultados apresentados nas Tabelas 5.7 e 5.8, onde, foi utilizado um algoritmo mais preciso para o cálculo dos melhores tempos de resposta. Como se pode verificar o tempo de resposta para o canal de menor prioridade foi reduzido para $6199\mu s$. Esta redução será tanto mais visível quanto maior for o número de canais a analisar.

Transacção 2 (Canal RC1)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*J*	*R*
Frag_0	cpu2	60	216	0	0	0	1140
Frag_2	cpu2	61	87	0	216	0	1227
Conf_4	cpu2	62	112	0	346	1290	2672
Comm_5	cpu2	63	54	0	458	1290	2726
Conf_8	cpu2	64	112	0	512	1307	2855
Comm_9	cpu2	65	54	0	624	1307	2909
Conf_12	cpu2	66	112	0	678	2414	4128
Conf_15	cpu2	67	112	0	790	2591	4417
Comm_17	cpu2	68	119	0	1123	3986	6152
Comm_14	cpu2	69	119	0	902	3876	5940
Frag_1	cpu1	60	106	0	346	1290	2720
Frag_3	cpu1	61	106	0	476	1343	2945
Comm_6	cpu1	62	119	0	582	2510	4189
Frag_7	cpu1	63	50	0	701	2510	4358
Comm_13	cpu1	64	54	0	751	2510	4412
Comm_10	cpu1	65	119	0	805	2576	4531
Frag_11	cpu1	66	92	0	924	2576	4689
Comm_16	cpu1	67	54	0	1016	2576	4743
Conf_18	cpu1	68	112	0	1070	3708	5868
Conf_19	cpu1	69	112	0	1182	3927	6199
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*J*	*R*
Frag_N2_0	can	60	130	0	216	924	1636
Frag_N2_2	can	61	130	0	346	881	1819
Comm_N2_5	can	63	53	0	512	2214	3092
Comm_N2_9	can	65	53	0	678	2231	3381
Comm_N1_13	can	64	53	0	805	3607	4778
Comm_N1_16	can	67	53	0	1070	3673	5109

Tabela 5.8: Transacção 2 (desfasamentos dinâmicos/*release jitter* equivalente) com o novo cálculo dos melhores tempos de resposta.

5.4.3.2 Modelo dos Desfasamentos Estáticos

A ferramenta de análise de escalonabilidade para este modelo foi desenvolvida por João Ventura (Ventura, 2001) a partir de uma extensão de uma ferramenta originalmente desenvolvida por Tindell (Tindell, 1994). A extensão realizada consistiu na integração de um módulo para cálculo dos piores tempos de resposta das mensagens num barramento de campo CAN no programa original desenvolvido por Tindell. Seguindo esta estratégia de extensão, fica facilitada a integração na ferramenta de modelos de análise de escalonabilidade para diferentes redes de comunicação.

Descreve-se de seguida o modo como a análise é feita. Inicialmente são calculados os piores tempos de resposta das tarefas nos vários processadores. De seguida actualizam-se os valores dos desfasamentos das mensagens de forma a que estes correspondam ao pior tempo de resposta das tarefas que as geram e aplica-se a análise ao conjunto de mensagens. Obtêm-se assim os tempos de resposta das mensagens que são usados para actualizar os desfasamentos das tarefas receptoras, repetindo-se o ciclo de calcular os piores tempos de resposta de todas as tarefas. Como estes valores crescem de forma estritamente monotónica, os valores acabam por convergir ou então o sistema não é escalonável.

Foram introduzidas na ferramenta de análise desenvolvida por Ventura (Ventura, 2001) as alterações necessárias que permitem optimizar os desfasamentos herdados pelas tarefas de uma *subtransacção*. Inicialmente, são calculados os desfasamentos para o melhor caso das tarefas de cada *subtransacção*, utilizando a Equação 5.4. A primeira tarefa da *subtransacção* que é activada por um evento externo possui um desfasamento nulo. Todas as tarefas que iniciam as outras *subtransacções* herdam um desfasamento dado pelo tempo de resposta da *subtransacção* precedente e que corresponde ao pior tempo de resposta calculado para a tarefa que possui o maior desfasamento dentro da *subtransacção*. Como os desfasamentos das tarefas que iniciam uma *subtransacção* tendem a variar em cada ciclo do algoritmo, cada tarefa i sua sucessora herda um

Transacção 1 (Canal RC0)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*R*	R^b_{i-1}
Frag_0	cpu1	20	216	72	0	288	0
Frag_2	cpu1	21	87	72	216	375	216
Conf_4	cpu1	22	112	72	548	732	0
Comm_5	cpu1	23	54	72	660	786	112
Conf_8	cpu1	24	112	72	678	898	0
Comm_9	cpu1	25	54	72	790	952	112
Conf_12	cpu1	26	112	72	969	1153	0
Conf_15	cpu1	27	112	72	1135	1319	0
Comm_14	cpu1	28	119	72	1447	1638	0
Comm_17	cpu1	29	119	72	1712	1903	0
Frag_1	cpu2	20	106	72	548	726	0
Frag_3	cpu2	21	106	72	678	856	0
Comm_6	cpu2	22	119	72	969	1160	0
Frag_7	cpu2	23	50	72	1088	1210	119
Comm_13	cpu2	24	54	72	1138	1264	169
Comm_10	cpu2	25	119	72	1135	1383	0
Frag_11	cpu2	26	92	72	1254	1475	119
Comm_16	cpu2	27	54	72	1346	1529	211
Conf_18	cpu2	28	112	72	1447	1641	0
Conf_19	cpu2	29	112	72	1712	1896	0
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*R*	R^b_{i-1}
Frag_N1_0	can	20	130	130	288	548	0
Frag_N1_2	can	21	130	130	375	678	0
Comm_N1_5	can	23	53	130	786	969	0
Comm_N1_9	can	25	53	130	952	1135	0
Comm_N2_13	can	24	53	130	1264	1447	0
Comm_N2_16	can	27	53	130	1529	1712	0

Tabela 5.9: Transacção 1 (desfasamentos no tempo).

desfasamento equivalente dado por:

$$O_i^{'} = O_0 + R^b_{i-1} \qquad \forall i \geq 1 \tag{5.5}$$

onde O_0 é o desfasamento herdado pela tarefa que inicia a *subtransacção*.

As Tabelas 5.9 e 5.10 apresentam os piores tempos de resposta (em μs) calculados para os dois canais, utilizando o modelo dos desfasamentos no tempo com a

Transacção 2 (Canal RC1)

tarefa	*cpu*	*prio*	*C*	*B*	*O*	*R*	R^b_{i-1}
Frag_0	cpu2	60	216	0	0	1028	0
Frag_2	cpu2	61	87	0	216	1121	216
Conf_4	cpu2	62	112	0	1418	2242	0
Comm_5	cpu2	63	54	0	1530	2296	112
Conf_8	cpu2	64	112	0	1601	2514	0
Comm_9	cpu2	65	54	0	1713	2568	112
Conf_12	cpu2	66	112	0	2609	3433	0
Conf_15	cpu2	67	112	0	2881	3705	0
Comm_14	cpu2	68	119	0	3780	4836	0
Comm_17	cpu2	69	119	0	4269	5100	0
Frag_1	cpu1	60	106	0	1418	1968	0
Frag_3	cpu1	61	106	0	1601	2265	0
Comm_6	cpu1	62	119	0	2609	3172	0
Frag_7	cpu1	63	50	0	2728	3359	119
Comm_13	cpu1	64	54	0	2778	3467	169
Comm_10	cpu1	65	119	0	2881	3698	0
Frag_11	cpu1	66	92	0	3000	3902	119
Comm_16	cpu1	67	54	0	3092	3956	211
Conf_18	cpu1	68	112	0	3780	4336	0
Conf_19	cpu1	69	112	0	4269	4825	0
mensagem	*rede*	*Prio*	*C*	*B*	*O*	*R*	R^b_{i-1}
Frag_N2_0	can	60	130	0	1028	1418	0
Frag_N2_2	can	61	130	0	1121	1601	0
Comm_N2_5	can	63	53	0	2296	2609	0
Comm_N2_9	can	65	53	0	2568	2881	0
Comm_N1_13	can	64	53	0	3467	3780	0
Comm_N1_16	can	67	53	0	3956	4269	0

Tabela 5.10: Transacção 2 (desfasamentos no tempo).

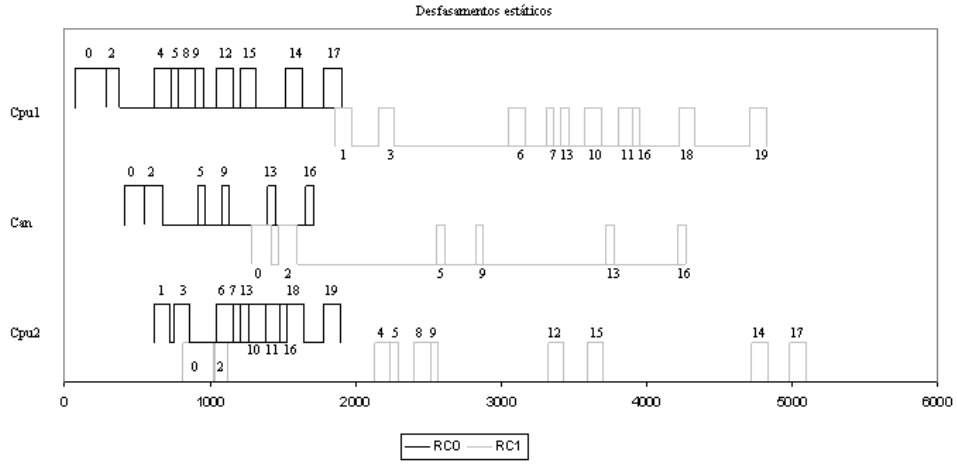

Figura 5.7: Diagrama de execução das tarefas com base nos tempos de resposta calculados pelo modelo dos desfasamentos estáticos.

aproximação descrita acima, e atribuindo uma tarefa à execução a cada gestor do diagrama. A coluna O apresenta o desfasamento calculado para cada tarefa utilizando a Equação 5.5 e a coluna R^b_{i-1} o melhor tempo de resposta da tarefa precedente. Um R^b_{i-1} nulo indica uma tarefa que inicia uma *subtransacção*.

A recepção da mensagem *Frag* pela *Aplicação* no canal $RC0$ (tarefa $Frag\_11$), tem um tempo de resposta calculado de $1475\mu s$. No entanto, devido à mensagem de *Commit* enviada pelo nó 2, a transacção 1 tem um tempo de resposta de $1903\mu s$, dado pela tarefa $Comm\_17$. Estes valores são idênticos aos obtidos com os desfasamentos dinâmicos. Para o canal $RC1$ estes tempos são respectivamente de $3902\mu s$ e $5100\mu s$.

A Figura 5.7 apresenta o diagrama construído com os piores tempos de resposta calculados, para cada canal e em cada processador. O canal $RC0$ é representado com traço negro e o canal $RC1$ com traço cinzento. Cada tarefa está representada por um patamar cuja largura representa o $TCPC$ da tarefa, subtraído ao tempo de resposta calculado. As tarefas estão identificadas por um valor numérico que corresponde ao subscrito do respectivo identificador representado nas Tabelas 5.9 e 5.10.

5.4.4 Análise Comparativa

O modelo holístico (Tabelas 5.3 e 5.4) é o que apresenta resultados mais pessimistas. Este pessimismo vem de três fontes: *i)* a interferência sofrida por uma tarefa devido a todas as tarefas de maior prioridade na mesma transacção e no mesmo processador; *ii)* a interferência sofrida por uma tarefa devido às tarefas de uma transacção de maior prioridade; *iii)* a influência dos valores herdados de *release jitter* no cálculo do período ocupado de tarefas de menor prioridade. A interferência referida em *ii)*, está relacionada com o cálculo do período ocupado, que inclui todas as tarefas das transacções de maior prioridade a serem executadas no mesmo processador, considerando que todas as tarefas são libertadas em simultâneo. Os valores herdados de *release jitter* para uma tarefa, referidos em *iii)*, vão também afectar a interferência causada pela tarefa nas tarefas de menor prioridade. No entanto a maior fonte de pessimismo introduzido pelo modelo holístico deve-se ao facto do calculo da interferência referida em *i)* incluir todas as tarefas de maior prioridade, mesmo as que possam já ter sido executadas e que não podem por isso causar qualquer interferência.

Os resultados obtidos com os modelos dos desfasamentos (Tabelas 5.5 a 5.10) são semelhantes. Os valores um pouco mais pessimistas obtidos com os desfasamentos dinâmicos, nomeadamente nas tarefas do canal $RC1$, de menor prioridade (Tabela 5.6), deve-se ao efeito que os valores de *release jitter* das tarefas de uma transacção têm no cálculo do período ocupado das tarefas de transacções com menor prioridade. O pessimismo introduzido pelos efeitos de *release jitter* tende a aumentar com o aumento do número de canais a analisar. No entanto utilizando um método mais preciso para o cálculo dos melhores tempos de resposta, é possível reduzir esse pessimismo. No modelo dos desfasamentos estáticos este efeito não se verifica. Utilizando os dois modelos dos desfasamentos no tempo apresentados nesta secção e confrontado-os com uma medição dos tempos na arquitectura final, em situações que possam reflectir o pior caso, será possível verificar se o pessimismo introduzido pelos desfasamentos dinâmicos é justificável.

No entanto, é possível concluir que o modelo dos desfasamentos no tempo é talvez mais apropriado para analisar conjuntos de tarefas com relações de precedência, como as que resultam de uma composição de protocolos. Pensamos também que a optimização introduzida com as *subtransacções* constitui uma aproximação válida na utilização dos modelos dos desfasamentos apresentados, nomeadamente quando o conjunto de gestores que compõem uma *subtransacção* são executados por uma única tarefa.

5.5 Síntese e Discussão

Neste capítulo apresentaram-se os mecanismos do *RT-Appia* para calcular o pior tempo de resposta de composições de protocolos a partir do grafo de eventos. Estes mecanismos baseiam-se num conjunto de ferramentas de análise que foram obtidas a partir da extensão de ferramentas inicialmente desenvolvidas por Tindell (Tindell *et al.*, 1992b; Tindell, 1994). Neste momento, a moldura *RT-Appia* suporta a análise usando diferentes modelos, nomeadamente o modelo holístico (Tindell *et al.*, 1992b), o modelo de desfasamentos dinâmicos (Palencia & Harbour, 1998), e o modelo de desfasamentos estáticos (Ventura, 2001). A experiência com a ferramenta confirma a intuição de que o modelo dos desfasamentos de tempo é o mais adequado à analise de composições de protocolos, uma vez que permite explorar a informação de causalidade na execução dos gestores de eventos capturada no grafo de eventos, para reduzir o pessimismo da análise.

6

Alocação e Optimização de Prioridades

Uma das dificuldades no projecto de sistemas distribuídos de tempo-real consiste em encontrar a alocação para um conjunto de tarefas a um conjunto de processadores e o conjunto de prioridades a atribuir a essas tarefas que minimize os tempos de resposta do sistema. Nesta secção será descrita uma metodologia simples para: *i)* a alocação de um conjunto de canais de tempo-real a um conjunto de processadores e; *ii)* um algoritmo baseado no arrefecimento simulado (*simulated annealing*) para a atribuição de prioridades a esse conjunto de canais, que minimize os seus tempos de resposta para o pior caso.

6.1 Alocação de Canais de Tempo-Real

Como já foi referido no Capítulo 3, Tindell (Tindell *et al.*, 1992a), utiliza o arrefecimento simulado para a resolução do problema da alocação, com o objectivo é minimizar a comunicação entre as tarefas de aplicação. No caso da alocação de canais de comunicação existem outros critérios a ter em conta, como: a distribuição da carga associada a cada canal, constrangimentos de memória volátil impostas pela arquitectura ou mesmo constrangimentos impostos por requisitos de tolerância a faltas.

O conceito de agregado de tarefas (Abdelzaher & Shin, 2000) pode ser directamente aplicado ao conjunto de tarefas que compõem uma transacção de um canal de tempo-real, repartindo o conjunto de tarefas da transacção num conjunto de *agregados locais*.

Um agregado local é composto pelo conjunto de tarefas da transacção que são executadas no mesmo processador e que, como já foi referido, corresponde ao conjunto de gestores executados pela mesma tarefa de um canal. Deste modo, o número de agregados locais de uma transacção é imposto pelo número de processadores utilizados pelo canal. De seguida é necessário determinar o factor de utilização $U(t,p)$ para cada agregado local. O factor de utilização imposto em cada processador p por uma transacção t com N tarefas é calculado por:

$$U(t,p) = \sum_{\forall i \subset p, i=1}^{N} \frac{C_i}{T_i} \tag{6.1}$$

Determinados os agregados locais de todas as transacções do sistema, é necessário identificar os subconjuntos de processadores onde cada transacção pode ser alocada. Utilizando o número de agregados locais impostos por cada transacção, são criados agregados de processadores que, identificam um subconjunto de processadores no sistema onde a transacção pode ser alocada. O conjunto de agregados de processadores necessários para alocar uma transacção com m agregados locais num sistema com um número p de processadores é dado por ${}_pC_m$, o número total de m combinações entre p elementos, com $p \geq m$. De notar que para n transacções com diferentes agregados locais é necessário compor n grupos de agregado de processadores diferentes. A *utilização* U_T total disponível em cada agregado de processadores é a soma da *utilização* U disponível para cada processador e a carga é o *factor de utilização* total de cada transacção, que é dado pela soma dos factores de utilização dos seus agregados locais.

É utilizado um algoritmo *worst-fit* para a alocação de agregado de tarefas a agregado de processadores, escolhendo o agregado de processadores com a maior utilização total disponível. Dentro de cada agregado de processadores é então alocada a carga a cada um dos processadores que o compõem, sendo actualizada a utilização disponível de cada processador e a utilização total do agregado. Este método tenta distribuir a carga igualmente entre os processadores, desde que os agregados de tarefas

do sistema estejam ordenados por ordem decrescente do seu factor de utilização. O algoritmo suporta diversos parâmetros de entrada que podem ser definidos pelo utilizador, tais como, a utilização total e a capacidade de memória disponíveis em cada processador, a pré-alocação de um conjunto de canais a um conjunto de processadores e a associação de replicas de canais, impondo que estes sejam alocados em processadores diferentes. A alocação de agregados de tarefas a agregados de processadores são realizadas tendo em conta todos estes parâmetros.

Quando é alocado mais do que um canal num determinado processador pode ser dada prioridade ao agrupamento de transacções com períodos harmónicos, que segundo (Abdelzaher & Shin, 2000), pode aumentar a utilização num processador.

Após ser realizada a alocação de um conjunto de canais, é optimizada a atribuição de prioridades aos canais utilizando um algoritmo baseado no arrefecimento simulado, como se descreve de seguida.

6.2 Optimização de Prioridades

O módulo de optimização de prioridades no *RT-Appia* tem por objectivo encontrar um conjunto de prioridades a atribuir a cada tarefa de cada canal em cada processador e a cada mensagem enviada por cada canal para a rede de comunicações, de modo a a tornar o conjunto escalonável e a minimizar o seu pior tempo de resposta. Na análise de escalonabilidade apresentada no Capítulo 5, foi atribuída a mesma prioridade base a todas as tarefas e mensagens de uma transacção, independentemente do processador onde a transacção é executada. No entanto, é possível obter outros tempos de resposta atribuindo a cada canal de tempo-real uma prioridade diferente em cada recurso (processadores e rede de comunicações) onde o canal é executado.

Assumindo que toda a actividade de um canal é descrita por uma única transacção, cada transacção tem um tempo de resposta global R_i e uma meta global D_i. O tempo

de resposta de uma transacção é dado pela tarefa com o maior tempo de resposta entre as tarefas da mesma transacção:

$$R_t = \max_{\forall k \in tasks(t)}(r_{tk}) \tag{6.2}$$

onde r_{tk} é o pior tempo de resposta de cada tarefa dentro da transacção. O teste de escalonabilidade é realizado comparando o valor de R_t calculado com a meta global da transacção, D_t.

A aplicação do arrefecimento simulado a qualquer problema de optimização passa pela concretização de duas funções fundamentais: a escolha de uma nova configuração dentro do espaço de soluções do problema e a avaliação da energia dessa nova configuração. O espaço de soluções do problema consiste no conjunto de todas as possíveis prioridades que podem ser atribuídas a um conjunto de canais. Este funcionamento, é ilustrado pelo algoritmo da Figura 6.1. De seguida descrevemos as funções *take_step()*, *energy()* e *accepted()* que especializam o comportamento do algoritmo para o problema da atribuição de prioridades.

A função *take_step()* permite obter uma nova configuração no espaço de soluções. O método seguido é simples e consiste em seleccionar aleatoriamente um processador (rede de comunicações incluída) e duas transacções. De seguida, trocam-se as prioridades entre as duas transacções no processador seleccionado, actualizando-se as prioridades de todas as tarefas (ou mensagens) de ambas as transacções. O algoritmo considera que inicialmente é atribuída uma prioridade a cada um dos canais. Com base nos resultados experimentais obtidos, este método representa um bom compromisso entre complexidade e o *grau de liberdade* para a escolha de uma nova configuração.

A função *energy()* determina o *custo* ou a *energia* da nova solução. Esta energia é dada pelo somatório das diferenças entre o tempo de resposta calculado para cada transacção e a sua meta, utilizando a equação seguinte:

```
proc siman(T_init, T_final, T_cool, T_accept, T_tries)
  begin
    T = T_init;
    E := energy();
    n := 0;
    do
       for j := 1 to T_tries step 1 do
          take_step();
          E_new := energy();
          if accepted(E, E_new) then
                                   E := E_new;
                                   n := n + 1;
                               else
                                   undo_step();
          fi
          if stop_criterion() then exit fi
          if n = T_accept then  endrep fi
    od
    T := T * T_cool;
  od
end
```

Figura 6.1: Algoritmo de arrefecimento simulado

$$E = \sum_{\forall t \in trans} max[0, R_t - D_t] \tag{6.3}$$

Onde R_t e D_t são respectivamente o pior tempo de resposta calculado e a meta de transacção t.

A função *accepted()* avalia a energia da nova solução utilizando a distribuição de probabilidades de Boltzmann definida pela Equação 3.29 e descrita no Capítulo 3. Se a nova solução não for aceite, a configuração anterior é reposta invocando a função *undo_step()*; caso contrário, a nova solução é adoptada como o novo ponto de partida e o ciclo é repetido um número de vezes para cada valor de temperatura T, até se obter um *equilíbrio térmico*. A condição de equilíbrio térmico utilizada estabelece um limite para o número de novas soluções aceites (T_{accept}) ou, em alternativa, um número máximo de tentativas (T_{tries}) para cada valor de temperatura, que depende da dimensão do *espaço de vizinhança*. O *espaço de vizinhança* consiste no conjunto de soluções possíveis que pode ser obtido a partir de uma solução corrente. Tindell (Tindell *et al.*,

1992b) considera um número máximo de tentativas de cerca de quatro vezes a dimensão do espaço de vizinhança.

Atingido o equilíbrio térmico, a temperatura é reduzida através da multiplicação por um factor de *arrefecimento* predefinido (T_{cool}) até que a temperatura atinja um valor mínimo, o qual determina a paragem do algoritmo. Poderá ser introduzida uma outra condição de paragem, como por exemplo a obtenção de uma configuração de prioridades que torne o sistema escalonável. No entanto, optou-se por executar o algoritmo até ser atingido o valor mínimo de temperatura, de modo a obter o número máximo de possíveis soluções.

O valor dos parâmetros a fornecer ao arrefecimento simulado varia em função da aplicação. A escolha do *factor de temperatura* inicial deve garantir que o número de novas configurações aceites seja elevada (ver Equação 3.29 no Capítulo 3). A possibilidade de novas configurações serem aceites vai sendo reduzida com a diminuição do *factor de temperatura*.

6.3 Apresentação de Resultados

Para exemplificar a utilização dos algoritmos de alocação e de atribuição de prioridades, vamos supor que se pretendiam utilizar seis canais distribuídos por uma arquitectura com três nós interligados por um barramento CAN. Para diminuir os tempos de processamento dos algoritmos, utilizou-se a pilha de protocolos CommitCan descrita no Capítulo 4. Assume-se também um cenário ideal onde não existem falhas e em que não é necessária a retransmissão das mensagens de *Commit*.

Convém realçar que o tempo de processamento do algoritmo depende essencialmente do cálculo dos tempos de resposta, que é realizado em cada ciclo. Utilizando seis canais e três nós, teríamos de calcular em cada ciclo os tempos de resposta para um total de 132 tarefas para o protocolo CommitCan e para 264 tarefas para o pro-

canal	*periodo*	*meta*	*cpu*	*prio*	*TRPC*
RC0	5000	2000	cpu1	30	2108
RC1	5000	1500	cpu2	20	1080
RC2	7500	5000	cpu3	50	4848
RC3	7500	4000	cpu3	40	3387
RC4	10000	5500	cpu2	60	6162
RC5	10000	6000	cpu1	70	7096

Tabela 6.1: Alocação dos seis canais nos três processadores.

cpu1	cpu2	cpu3	can
59.2%	59.2%	58.0%	15.9%

Tabela 6.2: Factores de utilização total nos três processadores e barramento CAN.

tocolo FragCan, o que tornaria o algoritmo de arrefecimento simulado extremamente demorado. Deste modo, o número de tarefas é reduzido para 72.

A Tabela 6.1 apresenta os períodos e metas de cada canal e os resultados da alocação dos seis canais nos três processadores. A coluna cpu indica onde foi alocado o canal emissor. Os factores de utilização resultantes para cada processador estão representados na Tabela 6.2. Supôs-se que todos os processadores possuiam uma utilização inicial de 100% e não possuiam qualquer constrangimento de memória. Como se pode observar o algoritmo distribuiu a carga igualmente entre os processadores.

A coluna *TRPC* da Tabela 6.1 apresenta os tempos de resposta no pior caso, entretanto calculados para cada canal, utilizando a atribuição de prioridades apresentada. Cada canal utiliza a mesma prioridade em todos os processadores onde é executado. Como se pode verificar o canal $RC0$ e os canais de menor prioridade $RC4$ e $RC5$ não cumprem as suas metas.

Seguidamente, foi executado o algoritmo de arrefecimento simulado para optimizar a atribuição de prioridades de cada canal em cada processador que torne o sistema escalonável. A Tabela 6.3 mostra o resultado com menor energia dado pelo arrefecimento simulado, entre as duas soluções encontradas. Foram usados os seguintes parâmetros: $T_{init} = 5000$, $T_{final} = 5$, $T_{cool} = 0,95$, $T_{accept} = 24$ e $T_{tries} = 100$. O tempo gasto pelo algoritmo a encontrar a primeira solução foi de $12170s$ e o tempo gasto a

Canal	Período	Meta	TRPC	prioridades			
				cpu1	cpu2	cpu3	can
RC0	5000	2000	1953	20	30	40	30
RC1	5000	1500	1494	30	20	30	20
RC2	7500	5000	4896	60	60	50	40
RC3	7500	4000	3099	50	50	20	50
RC4	10000	5500	5361	40	70	70	60
RC5	10000	6000	5467	70	40	60	70

Tabela 6.3: Prioridades atribuídas pelo algoritmo de arrefecimento simulado.

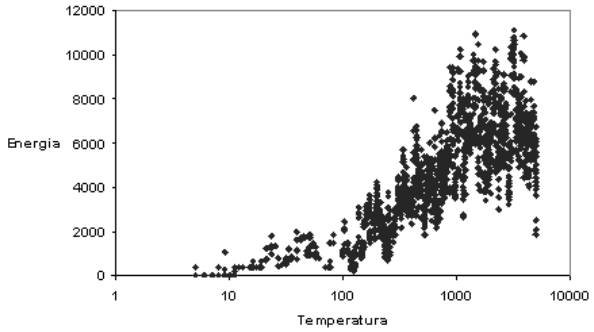

Figura 6.2: Variação da energia com a temperatura.

encontrar a solução apresentada foi de $12406s$.

Na Figura 6.2 é apresentado um gráfico com os valores de energia obtidos para as diversas temperaturas. Apesar dos valores apresentados no eixo da temperatura serem decimais, optou-se por uma representação logarítmica para tornar mais visível a variação da energia. Como se pode observar, os valores de energia tendem a ser menos dispersos e convergem para zero com a diminuição da temperatura. Os valores de baixa energia representados para a temperatura de 5000 devem-se à configuração inicial de prioridades, fornecida ao conjunto de canais.

Outro gráfico que é interessante observar é o do número de configurações aceites e rejeitadas pelo algoritmo, para cada valor de temperatura, representado na Figura 6.3. As configurações de *menor energia* são sempre aceites. As configurações de maior energia estão divididas em *aceites* e *rejeitadas* e correspondem à avaliação da distribuição de probabilidades de Boltzmann. Como se pode observar, para valores de temperatura elevados o número de configurações rejeitadas é muito baixo. O que significa que o

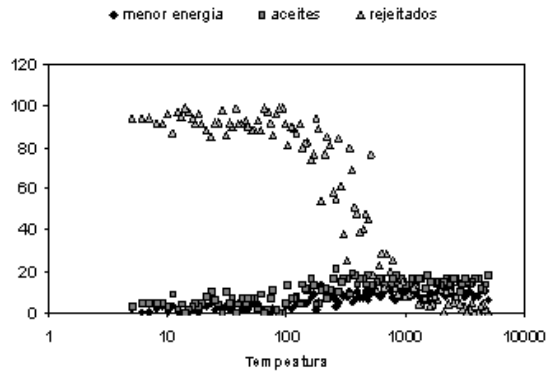

Figura 6.3: Variação com a temperatura das soluções com menor energia, aceites e rejeitadas.

valor de temperatura inicial escolhido não compromete a capacidade do algoritmo em aceitar a maioria das configurações propostas. Esta capacidade vai sendo reduzida com a diminuição de temperatura e, para valores de temperatura baixos, o número de configurações rejeitadas é elevado, enquanto o número de configurações de *menor energia* é muito baixo. Por esta razão, é necessário estabelecer um número máximo de ciclos (T_{tries}) porque, para valores de temperatura baixos, o número de configurações aceites pode nunca atingir o limite estabelecido por T_{accept}.

A Tabela 6.4 apresenta as prioridades atribuídas à pilha de protocolos FragCan, utilizando o arrefecimento simulado. Apesar de ser fácil determinar os tempos de resposta para todas as possíveis prioridades numa arquitectura com apenas dois nós e dois canais, este exemplo serve para complementar os resultados da análise de escalonabilidade apresentados na secção anterior. Com estas prioridades é possível reduzir o tempo de resposta do canal $RC1$ para 4622μs, utilizando o modelo dos desfasamentos dinâmicos. Foram utilizados os mesmos parâmetros do exemplo anterior com excepção de $T_{accept} = 2$ e $T_{tries} = 32$, visto que o espaço de soluções é muito menor.

Canal	Período	Meta	TRPC	prioridades		
				cpu1	cpu2	can
RC0	10000	4500	4202	20	60	20
RC1	20000	5000	4622	60	20	60

Tabela 6.4: Prioridades atribuídas aos dois canais com o arrefecimento simulado.

6.4 Avaliação e Discussão

Na utilização do arrefecimento simulado são possíveis outras aproximações na escolha de uma nova configuração, como por exemplo seleccionar aleatoriamente uma transacção e a prioridade a atribuir à transacção. No entanto existe maior possibilidade de transitar de pontos com baixa energia para um de alta energia. Supõe-se que, para este caso particular, seja preferível a troca de prioridades entre duas transacções. Alguns investigadores consideram também a introdução de sistemas periciais, com o objectivo de aumentar a possibilidade de uma nova solução ser aceite (Elmohamed *et al.*, 1998), eliminando um subconjunto de soluções em função de um conjunto de regras predefinidas. Para utilizar esta aproximação na atribuição de prioridades, seria necessário definir com exactidão o conjunto de regras a utilizar, sob pena de se restringir demasiado o *grau de liberdade* do sistema.

A metodologia apresentada para a alocação de um conjunto de canais permite organizar o espaço de soluções do problema facilitando a procura de novas soluções, principalmente quando a dimensão do espaço de soluções é elevada. Esta metodologia pode também ser incluída no algoritmo de atribuição de prioridades, não obstante um aumento significativo do tempo de processamento, uma vez que o espaço de soluções é muito maior. Em cada ciclo do algoritmo seriam afectados aleatoriamente agregados de tarefas a agregados de processadores, até se obter uma solução validada pelo algoritmo de alocação (sem que os constrangimentos de utilização, de memória dos processadores ou outros sejam violados). De seguida seria seleccionada aleatoriamente uma configuração de prioridades a atribuir ao conjunto de canais. De facto, uma das grandes vantagens da utilização do arrefecimento simulado é a grande flexibilidade

que oferece, podendo ser facilmente adaptado à resolução de novos problemas.

7

Modelo de Execução

Um dos principais objectivos deste trabalho consiste em obter uma moldura que permita fazer simultaneamente a análise temporal das composições de protocolos e obter um protótipo executável a partir do mesmo código fonte. Para isso, é necessário demonstrar que o modelo de composição anteriormente apresentado é exequível de concretizar em sistemas reais.

Neste capítulo, apresenta-se uma concretização do *RT-Appia* num sistema de execução concreto. Usa-se esta concretização para ilustrar o código resultante para a composição de protocolos que, nos capítulos anteriores, já foi utilizada para ilustrar a utilização das ferramentas de análise. Finalmente, apresenta-se uma análise de desempenho do protótipo resultante.

7.1 Sistema Alvo

O modelo de execução do *RT-Appia* consiste num processo composto por várias tarefas independentes cuja execução é suportada por um sistema operativo ou por um núcleo de multi-programação. As tarefas são escalonadas com base numa prioridade fixa e uma tarefa pode sofrer preempção de uma tarefa com maior prioridade. Este mecanismo de escalonamento é bastante flexível e é o mais utilizado na maioria dos sistemas operativos e núcleos de multi-programação comerciais.

Durante a preparação da dissertação, foi ponderada a utilização de diferentes sis-

temas alvo para concretizar o sistema *RT-Appia*:

- Inicialmente, considerou-se a hipótese de concretizar o modelo de composição do *RT-Appia* sobre a Especificação para Tempo-Real do Java (ETRJ) projectada pelo RTJEG (Real-Time for Java Experts Group) (Bollella *et al.*, 2000). O objectivo do RTJEG é o de fornecer uma plataforma de desenvolvimento de software adequada para uma larga extensão de aplicações, mantendo a compatibilidade com a linguagem Java. Algumas das características mais relevantes da ETRJ, incluem: a introdução de áreas de memória (*ScopedMemory*) que fornecem uma garantia temporal na reserva; a introdução entidades escalonáveis como tarefas de tempo-real (*RealTimeThread*) e classes de atendimento de eventos assíncronos (*AsyncEventHandler*), com uma elegibilidade de execução que depende da política de escalonamento utilizada; e a introdução de temporizadores periódicos e de disparo único. Infelizmente, na altura em que este trabalho teve início, não existiam concretizações de referência da ETRJ facilmente acessíveis, pelo que se abandonou esta ideia.

- Posteriormente, optou-se por concretizar o modelo de composição do *RT-Appia* na linguagem C++. Esta versão pode ser executado em sistemas operativos como o Windows NT/Windows 2000 ou em sistemas embebidos utilizando o núcleo ETS Kernel da Phar Lap Inc[1] ou o Windows CE. A plataforma de desenvolvimento utilizada para todos estes sistemas é o Visual C++ da Microsoft. No entanto, importa referir que só o Windows CE e o núcleo ETS suportam protocolos de herança de prioridade no acesso a recursos partilhados. A análise de desempenho apresentada neste capítulo usa uma versão que se executa sobre o Windows 2000, uma vez que só para este sistema foi possível obter gestores de periférico para as placas de acesso à rede CAN.

Nas secção seguintes iremos descrever a concretização dos diversos componentes

[1] ETS é um núcleo proprietário da Phar Lap Software, Inc.

do *RT-Appia* para esta última arquitectura. No entanto, é interessante salientar que muitas das opções de concretização podem ser facilmente adaptadas a outros sistemas operativos ou núcleos multi-tarefa de tempo real.

7.2 Canal de Tempo-Real

Um dos aspectos mais relevantes do modelo de execução do *RT-Appia* está relacionado com a execução de canais de tempo-real. Entre as actividades processadas por um canal, estão o escalonamento de eventos, a execução dos gestores das sessões, a gestão de temporizadores, o atendimento de eventos gerados na interface do canal, a gestão de memória relacionada com a reserva e libertação de eventos e mensagens e ainda o controlo de concorrência de sessões multi-canal. Todas as actividades do canal são executadas por uma tarefa de tempo-real fornecida pelo sistema operativo. Deste modo, um canal de tempo-real é um objecto escalonável pelo sistema operativo, com um atributo de prioridade que permite distinguir canais com diferentes prioridades de tráfego. Cada canal mantém as listas de eventos e as estruturas de dados necessárias para suportar a reserva de eventos e mensagens operadas pelas sessões e ainda a gestão de temporizadores e a activação de eventos na interface do canal, que serão descritas com maior detalhe nas secções seguintes.

Um canal de tempo-real é concretizado pelo objecto RTChannel e é composto por um conjunto ordenado de sessões. Cada sessão fornece o conjunto de gestores associados ao processamento dos eventos definidos na respectiva camada do protocolo. A posição da cada sessão no canal corresponde à posição ocupada pela camada na *configuração* definida para o canal. As instâncias de uma sessão são criadas através de pedidos realizados à respectiva camada do protocolo. A figura 7.1 apresenta um RTChannel. O *RT-Appia* define dois tipos de eventos, ChannelInit e ChannelClose, que são usados pelo canal para notificar as sessões da abertura e fecho de um canal. Ambos os eventos são propagados a todas as sessões do canal. O utilizador deve redefinir

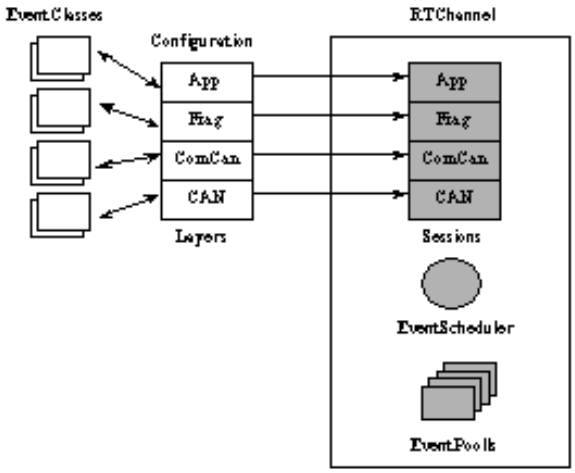

Figura 7.1: Exemplo de um objecto RTChannel

os gestores ChannelInitHandler e ChannelCloseHandler com as operações necessárias à iniciação e libertação de recursos para cada sessão.

O escalonamento de eventos e a execução dos gestores das sessões são realizados pelo objecto EventScheduler. Os eventos constituem o suporte para a comunicação entre sessões quer localmente (no mesmo processador) quer remotamente (noutro processador). Todos os eventos são concretizados com base na classe RTEvent. Cada evento mantém o estado sobre a próxima sessão a ser visitada num canal. Esta informação é gerida pelo objecto com base na tabela de encaminhamento do evento, definida na configuração do canal. Cada evento pode percorrer as sessões da sua tabela de encaminhamento na direcção descendente ou ascendente (DOWN/UP). Por omissão, é atribuída a direcção DOWN na criação de um evento. O utilizador pode também indicar qual das sessões incluídas na tabela de encaminhamento gera um evento. Neste caso, o evento só será entregue ao subconjunto de sessões que seguem a sessão geradora, numa dada direcção. Por omissão, a primeira sessão a ser visitada por um evento é a que se encontra num dos extremos da sua tabela. A iniciação de um evento é feita em função destes dois atributos.

Está associado a cada evento um atributo do tipo RTMessage para permitir às

sessões manipular as estruturas de dados associadas às mensagens dos protocolos. Esta classe fornece um conjunto de operações para adicionar e remover cabeçalhos, fragmentar e truncar mensagens, etc. No entanto, enquanto no *Appia* a dimensão dos tampões de dados associados a cada mensagem aumentam ou diminuem dinamicamente quando são adicionados ou removidos cabeçalhos, no *RT-Appia*, uma vez reservada, a dimensão dos tampões mantêm-se estática. Para activar um evento numa sessão remota, cada mensagem deve permitir identificar a classe do evento que a gerou.

As classes RTEvent e RTMessage são sub-classes da classe RTObject, que fornece as operações básicas relacionadas com a dinâmica dos objectos RTEvent e RTMessage, que permitem determinar em tempo de execução se um objecto é uma instância de uma determinada classe e a que lista de eventos disponíveis o objecto pertence.

7.3 Escalonamento de Eventos

O EventScheduler mantém todos os eventos que fluem no canal numa fila de eventos escalonáveis com uma disciplina FIFO. O escalonador selecciona o primeiro evento da lista e, de acordo com a sua tabela de encaminhamento, entrega-o à sessão que este deve visitar. Durante a execução do gestor do evento, a sessão pode inserir novos eventos na lista de eventos escalonáveis. No entanto, a execução do gestor não é interrompida com a inserção destes eventos. Em particular, se a sessão decide propagar o evento corrente, deve reinseri-lo na lista eventos do canal. Os eventos são processados pela ordem em que são inseridos na fila de eventos escalonáveis, dando prioridade aos eventos inseridos pela última sessão que foi visitada (a ordem de processamento de eventos é equivalente a aplicar um algoritmo de busca em profundidade ao diagrama de eventos extraído para o canal). Todos os eventos são inseridos na lista de eventos, invocando o método go do evento.

É apresentado em seguida o pseudo código que descreve o algoritmo utilizado no

escalonamento de eventos de um canal.

Algoritmo para o Escalonamento de Eventos:

```
begin
  while alive do
        waitForEvent();
        runnable = getRunnableEvent();
        while runnable do
              session = runnable.popSession();
              session.handler(runnable);
              runnable = getRunnableEvent();
        od
  od
end
```

O canal possui uma lista de *notificações*, na qual se bloqueia até receber uma notificação do sistema operativo (a activação de um temporizador ou de um evento na interface do canal). Quando ocorre uma notificação, o método waitForEvent insere o evento na lista de eventos escalonáveis. O conjunto de instruções seguinte é executado no âmbito do método consumeEvent do objecto EventScheduler, estando estas instruções relacionadas com o processamento dos eventos inseridos na lista de eventos do escalonador. A variável runnable contém o primeiro evento da lista de eventos (desde que a lista não esteja vazia). A próxima sessão a ser visitada pelo evento é obtida com o método popSession. Quando o evento runnable tiver visitado todas as sessões da sua tabela, é entregue ao gestor do canal e é seleccionado um novo evento. Quando forem processados todos os eventos e a lista de eventos estiver vazia, é novamente invocado o método waitForEvent, para verificar se existem notificações pendentes de eventos que entretanto possam ter sido activados.

O canal supervisiona o estado de execução de cada evento e gera uma excepção quando uma sessão tenta manipular um evento fora do contexto indicado pelo seu estado. Após a iniciação, um evento fica no estado READY. Quando um evento é inserido na lista de eventos escalonáveis, o canal coloca o evento no estado RUN-

NING. A transição para o estado RUNNING também ocorre quando uma sessão invoca o método go de um evento que se encontre no estado READY. Um evento que aguarda uma notificação do sistema operativo (a activação um temporizador ou de um evento na interface do canal) está no estado WAITING. Quando uma sessão cancela uma operação anteriormente invocada, como a paragem de um temporizador, o evento passa ao estado CANCELED.

Os eventos são inseridos na lista de eventos escalonáveis invocando o método insert do objecto EventScheduler. Opcionalmente, o método insert pode gerar uma excepção se verificar que o evento inserido não consta da tabela de *Triggers* da respectiva camada do protocolo. A utilização desta opção gera obviamente uma sobrecarga adicional no processamento de cada evento em cada sessão visitada pelo evento e deve ser usada apenas quando se justifique (nomeadamente no depuramento da concretização resultante).

7.4 Gestão de Temporizadores

No *RT-Appia* é possível associar um temporizador do sistema operativo a qualquer evento. Este mecanismo é suportado na classe RTEvent, que fornece um conjunto de métodos que permitem criar um temporizador (createTimer), iniciar um temporizador (startTimer), cancelar um temporizador (stopTimer) e libertar um temporizador (deleteTimer). Quando uma sessão cria um temporizador, o canal insere-o na lista de *notificações* do canal. O método startTimer inicia o evento em função dos atributos, direcção e sessão geradora, definidos para o evento. No caso dos temporizadores, a sessão geradora indica a sessão a que o evento é entregue quando o temporizador dispara e que normalmente é a própria sessão.

Quando o canal detecta o disparo do temporizador, insere o evento na lista de eventos escalonáveis. Enquanto uma sessão decidir propagar o evento, este será entregue a todas as outras sessões incluídas na sua tabela de encaminhamento. Este mecanismo

é bastante flexível pois permite construir eventos dedicados a funcionar unicamente como temporizadores mas também activar temporizações em qualquer evento, o que permite, por exemplo, que uma sessão guarde o estado de uma mensagem em protocolos com re-transmissões, evitando a introdução de mecanismos adicionais para esse efeito. Outra das vantagens em fazer propagar eventos com temporizadores por um conjunto de sessões, é na introdução de protocolos de sincronismo de relógio.

A gestão de temporizadores é baseada no estado do evento. Quando uma sessão cria um temporizador, o evento transita para o estado CANCELED. Se o temporizador for iniciado, o estado do evento passa para WAITING. Quando ocorre uma notificação de um temporizador, o método waitForEvent verifica se o evento se encontra no estado WAITING. Caso o estado do evento seja outro (i.e. CANCELED) isso significa que o temporizador foi cancelado e o evento não é inserido na lista de eventos mas é entregue ao gestor do canal.

Se o evento é inserido na lista de eventos escalonáveis, passa ao estado RUNNING. As operações invocadas pelas sessões durante o processamento do evento podem modificar o seu estado, que se mantém inalterado mesmo com a invocação do método go. Quando o evento é devolvido ao canal (método channel.handler, invocado directamente pela sessão ou indirectamente através do método go), o seu estado será mantido ou alterado em função do tipo de evento. Se for um temporizador periódico e o estado se mantiver RUNNING, o evento passa a WAITING. Se for um temporizador de disparo único e o estado se mantiver RUNNING, o evento passa a CANCELED. Caso não seja um temporizador (ou o temporizador tenha sido libertado), o evento transita para o estado READY.

A Figura 7.2 indica os estados de um evento e os métodos que provocam uma mudança de estado.

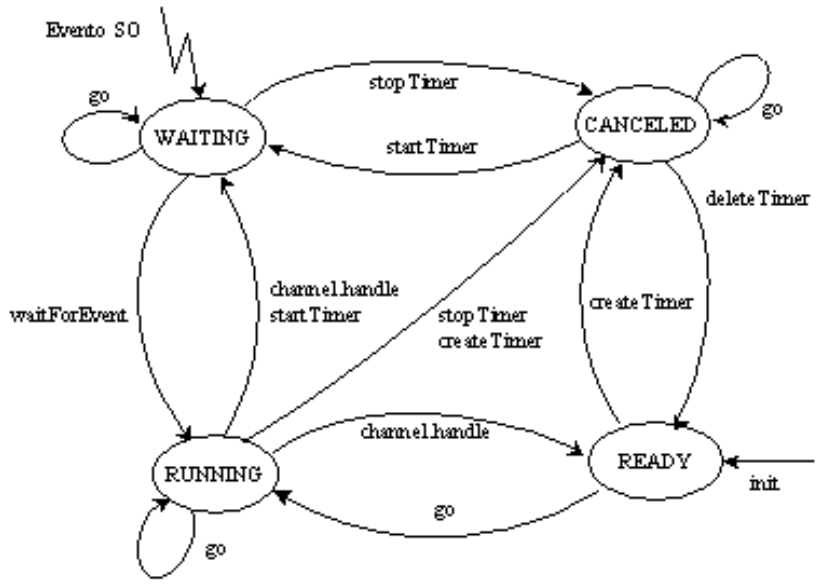

Figura 7.2: Estados de um evento.

7.5 Interface com a Rede e com as Aplicações

A interface de um canal de tempo-real com a rede de comunicações e com a aplicação é realizada em camadas bem definidas. Tipicamente, qualquer canal tem uma camada de aplicação e uma camada de rede. Estas camadas são responsáveis por inserir eventos no canal em resposta aos estímulos do utilizador ou de mensagens que chegam da rede de comunicação. No *RT-Appia* estas camadas são responsáveis por definir os eventos e os gestores associados a estes estímulos. Os eventos definidos por estas camadas são inseridos assincronamente na lista de eventos escalonáveis do canal, para evitar a introdução de mecanismos de controlo de concorrência na lista de eventos escalonáveis.

Para suportar a concretização destas camadas, o *RT-Appia* permite associar eventos do sistema operativo a eventos do canal definidos pelo utilizador. O método setOSEvent faz a associação entre esses eventos. Estas associações são inseridas na lista de notificações do canal. Quando o evento do sistema operativo é activado, o escalonador de eventos insere o respectivo evento na lista de eventos escalonáveis, que será então

entregue ao gestor da respectiva camada de interface. É possível criar associações com um único evento definido pelo utilizador ou com uma lista de eventos. O método setOSEvent permite criar automaticamente uma lista de eventos disponíveis para o evento do canal, com base numa dimensão definida pelo utilizador.

Em alternativa a este mecanismo, a interface com um canal pode também ser realizada utilizando gestores assíncronos, concretizados pela classe AsyncEventHandler. O método handler() desta classe é invocado directamente pelo canal quando ocorre a notificação associada a um objecto do tipo AsyncEventHandler. Este método é idêntico ao gestor de uma sessão e deverá ser definido pelo utilizador. No entanto, ao contrário de uma sessão, este método não aceita nenhum parâmetro. Na versão actual do *RT-Appia* é possível associar um só objecto do tipo AsyncEventHandler a um só evento do sistema operativo.

Este mecanismo deverá ser utilizado quando existir uma maior urgência no atendimento de um evento associado à interface de um canal. O método waitForEvent, descrito no escalonamento de eventos, também verifica se foi activada alguma notificação associada aos gestores assíncronos. Caso exista alguma notificação pendente, é invocado o respectivo gestor assíncrono.

Opcionalmente, o método waitForEvent pode ser invocado no modo não bloqueante. Neste modo, o método retorna de imediato caso não existam notificações pendentes. Este modo pode ser utilizado numa versão do escalonador de eventos que permita que um gestor assíncrono seja executado durante o processamento de um evento (i.e. entre a execução de duas sessões). No entanto, isso introduz um *jitter* variável na execução de cada evento, sendo difícil determinar quais os gestores assíncronos que podem ser activados durante a execução de cada evento. No pior caso poder-se-á considerar que todos os gestores podem estar activos durante a execução de cada evento, o que vai aumentar substancialmente o pessimismo da análise de escalonabilidade.

A classe AsyncEventHandler fornece o suporte base que permite realizar a

interligação entre um canal e uma sessão, mas não fornece nenhum mecanismo de sincronização no acesso a secções críticas. Faz-se notar que pode ser necessário utilizar mecanismos de sincronização, nos métodos de uma sessão partilhada, quando estes são invocados directamente de um objecto do tipo AsyncEventHandler. Para estes casos é da responsabilidade do utilizador invocar os mecanismos de sincronização necessários.

A classe AsyncEventHandler suporta ainda a possibilidade de que a execução do método handler esteja associada a um temporizador periódico, permitindo que a interface de um canal seja concretizada por um servidor periódico. Importa referir que o pior tempo de resposta de um canal que utiliza um servidor periódico é maior do que o período do servidor, uma vez que uma mensagem pode chegar ao periférico logo após o servidor ter sido executado. Deste modo o servidor só detectará a chegada da mensagem no período seguinte.

Estes mecanismos são particularmente úteis para concretizar uma interface assíncrona com um gestor do periférico da rede de comunicações e foram utilizados na concretização da camada CAN para a composição de protocolos descrita nos capítulos anteriores. Um aspecto importante que importa referir no *RT-Appia* é o de que todo o processamento adicional necessário à concretização da interface com um gestor de um dispositivo, pode ser executado dentro da prioridade do canal.

A interface de uma rede de comunicações com o gestor do periférico pode levantar alguns problemas. Um destes problemas está relacionado com a demultiplexagem das mensagens recebidas (identificar a que canal a mensagem é endereçada). Para evitar mudanças de contexto adicionais, a política seguida pelo CORDS realiza a demultiplexagem de um Path dentro da rotina de atendimento da interrupção associada ao dispositivo da rede de comunicações, invocando directamente um filtro de pacotes (Mogul *et al.*, 1987) integrado no gestor do dispositivo. Só então é activada a tarefa associada ao respectivo Path. No entanto, como já foi referido, o processamento adicional dentro de rotinas de atendimento de interrupções não é desejável em

sistemas de tempo-real porque agrava o efeito de inversão de prioridade e pode complicar a análise de previsibilidade do sistema. No *RT-Appia*, é da responsabilidade do programador utilizar o método mais apropriado.

7.6 Reserva de Memória

Como a maioria dos sistemas operativos comerciais não fornecem mecanismos de gestão de memória apropriados para o suporte de aplicações de tempo-real estrito, cada canal deve pré-reservar toda a memória necessária, de modo a garantir uma maior previsibilidade durante a sua execução. Assim, no *RT-Appia* todos os eventos (e mensagens) devem ser pré-reservados durante a iniciação do canal.

Cada canal fornece um mecanismo de gestão de listas de eventos para gerir a reserva e libertação de eventos criados pelas sessões. O objecto RtEventPool cria uma lista de eventos com base na classe do evento, na dimensão da lista e num identificador fornecido pelo canal. Este identificador é utilizado para assegurar que, quando o evento é libertado, este seja inserido na lista de pré-reserva correspondente. Cada evento pode ter associada a ele uma mensagem, que é reservada e libertada juntamente com o evento. O objecto RtEventPool cria automaticamente todas as instâncias dos eventos da lista (incluindo as mensagens associadas aos eventos) e coloca todos os eventos no estado READY.

Durante a iniciação de um canal (ao receber o evento ChannelInit), cada sessão deve reservar uma lista para cada tipo de evento que possa gerar durante a execução do canal. Os pedidos de reserva são dirigidos ao canal e podem ser realizados por qualquer sessão do canal, independentemente da sessão que criou a lista. Todos os eventos reservados são libertados automaticamente pelo canal, após executarem a última sessão da sua tabela de encaminhamento. Em todo o caso, uma sessão pode sempre requerer a libertação de um evento.

Uma sessão deve invocar o método getEventPool para reservar um novo evento ao canal. Este método aceita como parâmetro o identificador fornecido pelo canal durante a criação da lista de eventos. No entanto, podem ocorrer situações onde uma sessão necessita reservar um evento mas desconhece o identificador da lista de eventos disponíveis, como por exemplo quando a camada de rede necessita de reservar um evento da classe identificada pela mensagem recebida de um nó remoto. Para estes casos, é definido uma outra versão do método para identificar a lista de eventos disponíveis com base na classe do evento.

A dimensão de cada lista pode ser obtida automaticamente a partir da informação do conjunto de tarefas extraídas do diagrama de eventos, considerando que cada tarefa executa n gestores da sua tabela de encaminhamento. A dimensão obtida deste modo pode, no entanto, estar sobre-dimensionada, pelo que, dependendo dos requisitos impostos pela composição de protocolos, o utilizador deverá realizar as escolhas necessárias de modo a obter a previsibilidade necessária que evite a utilização excessiva de recursos.

7.7 Sessões Multi-Canal

A possibilidade de haver uma sessão a ser usada por mais do que um canal é uma característica importante do *Appia* porque permite manter os mecanismos para exprimir constrangimentos entre canais. No entanto, sessões partilhadas podem facilmente sofrer problemas de inversão de prioridade devido a: *i*) controlo de concorrência no acesso a estruturas de dados partilhadas; *ii*) aspectos de coordenação entre canais específicas a cada sessão.

Na moldura *Appia*, o controlo de concorrência é evitado utilizando uma estratégia de escalonamento de eventos que passa pela utilização de uma única tarefa para processar todos os eventos em todos os canais. Com a introdução de concorrência entre canais no *RT-Appia*, torna-se necessário utilizar mecanismos de sincronização

em sessões partilhadas. As sessões devem implementar um par de métodos de sincronização, preHandleSync e postHandleSync, que são invocados pelo *RT-Appia* em tempo de execução, quando uma sessão é partilhada por canais diferentes. O método preHandleSync é invocado antes da invocação do gestor da sessão e o método postHandleSync é invocado após o seu retorno. O *RT-Appia* assume que é utilizado um protocolo de herança ou de tecto de prioridade. É possível inferir automaticamente de uma instância de um canal se existem sessões partilhadas com outros canais.

As sessões mantêm a informação sobre os canais a que estão associadas, permitindo determinar automaticamente se a sessão é partilhada por mais do um canal e qual o canal de menor prioridade. Utilizando esta informação é possível determinar o TCPC de uma sessão partilhada.

7.8 Exemplo da Concretização das Sessões

Para complementar o exemplo prático da composição de protocolos apresentada no Capítulo 4, será apresentado nesta secção alguns exemplos da concretização das sessões, que permite exemplificar os mecanismos descritos nas secções anteriores e outros aspectos do modelo de execução do *RT-Appia*. O objectivo visado foi o de construir uma aplicação para a medição de tempos no *RT-Appia* e principalmente a comparação com os resultados obtidos na análise de escalonabilidade pelo que não visou uma concretização robusta da pilha de protocolos. Para comodidade do leitor repete-se na Figura 7.3 a composição da pilha de protocolos FragCan.

7.8.1 Camada de Aplicação

A camada de aplicação define a carga do sistema e é representada pela sessão AppCanSession. A carga é estabelecida por um temporizador periódico, criado pela própria sessão. Em cada período do temporizador, é invocado o gestor hFragTx-

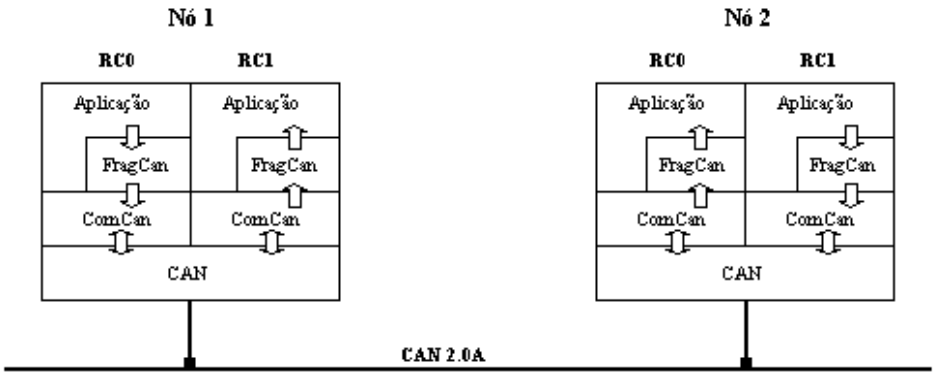

Figura 7.3: Pilha de protocolos do FragCan

Down que envia uma mensagem de dados com uma dimensão de dois octetos. A Listagem 7.1 apresenta os gestores ChannelInitHandler e hFragTxDown para a sessão AppCanSession.

Durante a execução do ChannelInitHandler, a camada de aplicação cria uma lista de eventos disponíveis para a classe dataTxFragClass, utilizando o método createEvent-Pool. A constante $FRAG\_POOL\_SZ$ indica a dimensão da lista e $FRAG\_MSG\_SZ$ a dimensão da mensagem associada ao evento. De seguida, é reservado um evento da lista, onde é criado um temporizador periódico, representado por dataTxFrag. O temporizador é iniciado invocando o método startTimer que aceita como único parâmetro o período do temporizador. Este método inicia o evento em função da direcção e da sessão geradora do evento, que no caso são as estabelecidas por omissão.

Em cada período do temporizador, o gestor hAppTxDown da sessão preenche a mensagem associada ao evento com os dados a transmitir e invoca o método go do evento. O método discard inicia o tampão do objecto RtMessage e push coloca os dados data16 no tampão da mensagem. A constante $RC0\_FRAG\_ID$ define o identificador da mensagem FRAG (associada ao canal RC0) a enviar para o barramento CAN. São atribuídos dois identificadores CAN às mensagens transmitidas por cada canal: um para as mensagens de dados associadas aos eventos FragEvent e outro para as mensagens RTR associadas aos eventos CommitEvent. No nó remoto, o canal tem que

Listagem 7.1: AppCanSession (ChannelInitHandler e hFragTxDown).

```
// ...
void AppCanSession::ChannelInitHandler( RtEvent* e ) {
    // ...
    // criar uma lista de eventos disponiveis para dataTxFragClass
    fragPoolId=channel->createEventPool( FRAG_POOL_SZ, dataTxFragClass, FRAG_MSG_SZ );
    // obter um evento da lista e inicia-lo
    dataTxFrag = (FragEvent *)channel->getPoolEvent( fragPoolId );
    fragTxMsg=dataTxFrag->getMessage( );
    fragTxMsg->setPriority( RC0_FRAG_ID ); // atribuir a prioridade a mensagem
    dataTxFrag->createTimer( PERIODIC ); // criar e iniciar o temporizador
    dataTxFrag->startTimer( RC0_FRAG_PERIOD );
    e->go( );
}

void AppCanSession::hAppTxDown( RtEvent* e ) {
    RtMessage* m=e->getMessage( );
    m->discard( ); // descartar os dados da mensagem
    m->push( data16, 0, sizeof(data16) ); // prencher a mensagem com os dados data16
    e->go( );
}
// ...
```

declarar ao gestor do periférico os identificadores das mensagens que está interessado em receber. Deste modo, a demultiplexagem de mensagens é realizada pelo gestor do dispositivo que sinaliza automaticamente um canal em função dos identificadores das mensagens recebidas.

7.8.2 Camada FragCan

A sessão FragCanSession representa a execução de camada FragCan e é apresentada na Listagem 7.2. O gestor hFragTxDown cria dois fragmentos da mensagem recebida e propaga os respectivos eventos. O segundo fragmento é propagado com um evento obtido da lista de eventos disponíveis para a classe dataTxFragClass e é representado pela variável txFragEvent. O método setDirection define a direcção do evento, que neste caso, se justifica porque os eventos dataTxFragClass também são reservados pela sessão CanSession. O método setGenerator define a própria sessão como a sessão geradora, que determina que evento será entregue às sessões seguintes (CommitCanSession e CanSession). O método init inicia o evento em função dos atributos anteriores. De seguida a mensagem é fragmentada e são propagados os dois fragmentos.

A reconstrução de uma mensagem é realizada no gestor hFragTxUp. É iniciado o temporizador de disparo único, fragTimer, criado num evento reservado ao canal durante a sua iniciação. Este temporizador é iniciado com a direcção UP e a própria sessão como a sessão geradora, para que, quando o temporizador disparar, este seja entregue à sessão FragCan. Quando é recebido o primeiro fragmento de uma mensagem, o temporizador é iniciado e o fragmento é guardado na mensagem do temporizador (fragTimerMsg) (a variável newFrag permite distinguir o primeiro do segundo fragmento recebidos). Caso não seja entretanto recebido o segundo fragmento, o temporizador dispara e a sessão considera que houve um erro e descarta o primeiro fragmento (ver método handleAlarm). Quando é recebido o segundo fragmento, o temporizador é cancelado e a mensagem original é reconstruída invocando o método join de fragTimerMsg. A mensagem é propagada à camada de aplicação utilizando o temporizador. Faz-se notar que invocando `channel->handler`, se evita que o evento recebido com o primeiro fragmento seja propagado à camada de aplicação.

7.8.3 Camada CommitCan

A Listagem 7.3 apresenta os gestores da sessão CommitCanSession. A iniciação da sessão CommitCanSession consiste essencialmente na iniciação do temporizador commitTimer usado no gestor hDataTxUp, que está associado à recepção de uma mensagem de dados. Este temporizador é usado para detectar se os nós remotos enviaram uma mensagem de COMMIT, caso o emissor falhe após transmitir uma mensagem de dados. Como esta sessão é a única que gera eventos COMMIT, é criada durante a iniciação uma lista de eventos disponíveis para a classe commitClass, onde é reservado o evento commitTimer.

Após propagar o evento recebido (gestor hDataTxDown), a sessão aguarda a recepção de uma confirmação. Ao receber a confirmação da mensagem transmitida, o gestor hConfirmUp propaga o respectivo COMMIT com um evento reservado da lista de eventos disponíveis do canal. O evento recebido é devolvido ao canal invocando o

Listagem 7.2: FragCanSession (hFragTxDown, hFragTxUp e handleAlarm).

```
// ...
void FragCanSession::hFragTxDown( RtEvent* e ) {
    RtMessage* m = e->getMessage( );

    // obter um novo evento da lista de eventos disponiveis e inicia-lo
    txFragEvent = channel->getPoolEvent( dataTxFragClass );
    txFragEvent->setDirection( DOWN );
    txFragEvent->setGenerator( this );
    txFragEvent->init( );
    txFragMsg = txFragEvent->getMessage( ); // colocar o segundo fragmento em txFragMsg
    m->frag( txFragMsg, m->getLength( ) - MAX_CAN_SZ );
    txFragMsg->setPriority( m->getPriority() ); // dar a mesma prioridade a nova mensagem
    e->go( ); // propagar ambos os fragmentos
    txFragEvent->go( );
}

void FragCanSession::hFragTxUp( RtEvent* e ) {
    RtMessage* m = e->getMessage( );

    if ( newFrag ) {
        fragTimerMsg = m;           // copiar m para fragTimerMsg
        fragTimer->startTimer( );   // iniciar temporizador
        newFrag = false;
    }
    else  {
        fragTimer->stopTimer( );    // parar temporizador
        fragTimerMsg->join( m );    // reconstruir a mensagem
        fragTimer->go();            // go
        newFrag = true;
    }
    channel->handler( e );          // devolver o evento recebido ao canal
}

void FragCanSession::handleAlarm( RtEvent* e ) {
    newFrag = true;
    // Erro ...
}
// ...
```

método go do evento.

A recepção de um COMMIT é tratada no gestor hCommitUp, que faz go do evento retirado do tampão rxData e retransmite o COMMIT recebido na direcção descendente (DOWN).

A recepção de mensagens é tratada no gestor hDataTxUp, que guarda o evento no tampão rxData e inicia o temporizador commitTimer. O método isCanceled( ) permite verificar o estado de um evento. Tratando-se de um temporizador, o evento deve estar no estado CANCELED para permitir ser re-utilizado, caso contrário, um erro grave ocorreu. Se entretanto não for recebido um COMMIT, o temporizador dispara e é invocado hCommitDown, que propaga o evento retirado do tampão rxData e transmite a mensagem de COMMIT.

7.8.4 Sessão CanSession

A sessão CanSession faz a interface com o gestor do dispositivo do barramento CAN. No modo assíncrono fornecido pelo gestor, a confirmação de uma transmissão e a recepção de uma mensagem são realizadas assincronamente, através da sinalização de eventos do sistema operativo. São definidos os gestores assíncronos AsyncRxNetHandler e AsyncTxNetHandler, derivados da classe AsyncEventHandler e associados, respectivamente, à recepção de uma mensagem e à recepção de uma confirmação. A Listagem 7.4 apresenta a definição destas classes assim como o gestor ChannelInitHandler e a Listagem 7.5 os gestores das sessões, hRxNet, hConfirm e hDataTx.

O método ChannelInitHandler cria as listas para os eventos confirmClass e dataTxClass (as listas criadas para os outros eventos, pelas outras sessões, podem ser usadas pela sessão CanSession). Se a sessão utiliza o modo de funcionamento assíncrono (NTCAN_MODE_OVERLAPPED), são criados os gestores AsyncRxNetHandler e AsyncTxNetHandler. Os eventos associados aos gestores, são guardados nas estruturas de

Listagem 7.3: CommitCanSession.

```
// ...
void CommitCanSession::hDataTxDown( RtEvent* e ) {
    e->go();
}

void CommitCanSession::hConfirmUp( RtEvent* e ) {
    CommitEvent* commit = (CommitEvent *)channel->getPoolEvent( commitPoolId );
    if ( commit != NULL ) {
        commit->setMessage( commitMsg );
        commit->setDirection( DOWN );
        commit->setGenerator( this );
        commit->init( );
        commit->go();
    }
    else
        // Error ...
    e->go();
}

void CommitCanSession::hDataTxUp( RtEvent* e ) {
    rxData->Put( e );                    // guardar o evento ate receber COMMIT
    if ( commitTimer->isCanceled( ) ) {
        commitTimer->startTimer( ); // iniciar temporizador
    }
    else
        // Erro: o evento nao esta disponivel ...
}

void CommitCanSession::hCommitUp( RtEvent* e ) {
    if ( !rxData->Empty() ) { // obter o evento da lista e propaga-lo
        DataTxEvent *event = (DataTxEvent *)rxData->Get( );
        event->go( );
        e->setDirection( DOWN ); // criar e iniciar o temporizador
        e->setGenerator( this );
        e->init( );
    }
    e->go( );
}

void CommitCanSession::hCommitDown( RtEvent* e ) { // obter o evento da lista e propaga-lo
    DataTxEvent *event = (DataTxEvent *)rxData->Get( );
    event->go( );
    e->go( );   // transmitir COMMIT
}
// ...
```

controlo para a recepção e transmissão assíncrona (rxOverlapped e txOverlapped, respectivamente). No modo síncrono, o gestor AsyncRxNetHandler é associado a um temporizador, que é iniciado com o período do servidor.

O método handler da classe AsyncRxNetHandler invoca o método hRxNet da sessão. O método hRxNet lê a mensagem do gestor do dispositivo, faz a reserva do respectivo evento e faz com que ele seja propagado às sessões acima. A classe do evento (dataTxClass ou dataTxFragClass) é identificada pelo primeiro octeto da mensagem utilizando o método da sessão getEventClassHandle. Como a sessão CanSession é partilhada pelos dois canais, o parâmetro channel identifica o canal onde ocorreu a notificação. É importante referir que o controlo de concorrência dentro deste método é garantido por não serem utilizadas na sessão variáveis comuns ao processamento de ambos os canais. A sessão reserva estruturas de controlo independentes, usadas para a transmissão e recepção de mensagens, para cada um dos canais criados.

O método handler da classe AsyncTxNetHandler invoca o método hConfirm da sessão, representado na Listagem 7.6, que lê a mensagem de confirmação do gestor do dispositivo e faz propagar o respectivo evento.

A transmissão de mensagens é tratada no gestor da sessão hDataTx, invocado pelo escalonador de eventos do canal, que preenche a estrutura de controlo para a transmissão de uma mensagem e invoca o método writeCan. O primeiro octeto da mensagem é reservado à identificação da classe do evento. Os métodos writeCan e readCan da sessão invocam os procedimentos do gestor do dispositivo para enviar e receber mensagens, respectivamente.

O modo assíncrono do gestor do dispositivo foi utilizado por o seu funcionamento ser semelhante à sinalização através de uma rotina de atendimento da interrupção associada ao dispositivo, visto que não foi possível obter o código fonte do gestor que permitiria concretizar uma interface directa com o *RT-Appia*.

Foi também concretizada uma interface síncrona com o gestor do dispositivo que

Listagem 7.4: CanSession (ChannelInitHandler).

```
class AsyncRxNetHandler : public AsyncEventHandler {
public:
    void handler( ) {
        CanSession* s = (CanSession*)session;
        s->hRxNet( channel ); // invocar hRxNet
    }
};

class AsyncTxNetHandler : public AsyncEventHandler {
public:
    void handler( ) {
        CanSession* s = (CanSession*)session;
        s->hConfirm( channel ); // invocar hConfirm
    }
};

void CanSession::ChannelInitHandler( RtEvent* e ) {
    initCan();
    // criar uma pool de eventos confirmClass
    channel->createEventPool( txqueuesize, confirmClass, CONF_MSG_SZ );
    // criar uma pool de eventos dataTxClass
    channel->createEventPool( rxqueuesize, dataTxClass, DATA_MSG_SZ );

    if ( mode == NTCAN_MODE_OVERLAPPED ) {
        // obter o identificador do canal
        int chId = getChannelId( e->getChannel( ) );
        // criar o gestor assincrono rxNetHandler
        AsyncRxNetHandler* rxNetHandler = new AsyncRxNetHandler( *channel, *this );
        // guardar o gestor do evento criado para a recepcao
        rxOverlapped[chId].hEvent = rxNetHandler->createAsyncEvent( );
        // criar o gestor assincrono txNetHandler
        AsyncTxNetHandler* txNetHandler = new AsyncTxNetHandler( *channel, *this );
        // guardar o gestor do evento criado para a transmissao
        txOverlapped[chId].hEvent = txNetHandler->createAsyncEvent( );
        // ...
    }
    else {
        // criar o servidor periodico rxNetHandler
        AsyncRxNetHandler* rxNetHandler = new AsyncRxNetHandler( *ch, *this );
        rxNetHandler->createAsyncTimer( );
        rxNetHandler->startAsyncTimer( RXNET_PERIOD );
    }
}
```

pode ser utilizada criando uma instância da sessão CanSession, no modo síncrono, através de um parâmetro fornecido ao seu construtor. Neste modo, só é utilizado o gestor AsyncRxNetHandler, que é executado como um servidor periódico. O servidor invoca a função não bloqueante, canTake do gestor do dispositivo, para verificar se existem novas mensagens para serem lidas. Para a transmissão de mensagens no modo síncrono, é utilizado o procedimento canWrite, fornecido pelo gestor. Como este procedimento só retorna quando recebe a confirmação da transmissão, a mensagem de confirmação é propagada às camadas superiores logo após a chamada a este procedimento e não é necessário utilizar o gestor AsyncTxNetHandler.

Foi também concretizada uma versão da sessão CanSession utilizando uma interface notificada por eventos. Nesta versão, hRxNet e hConfirm são definidos como gestores da sessão e são invocados pelo escalonador de eventos do canal. A única diferença é aceitarem como parâmetro o tipo RTEvent. São definidos os eventos RxNetEvent e TxNetEvent que durante a iniciação da sessão são associados a eventos do sistema operativo utilizando, o método setOSEvent. Este método aceita como parâmetros uma referência para o evento e uma referência para a sessão que pretende receber o evento. A Listagem 7.6 apresenta a declaração dos eventos e os métodos ChannelInitHandler, hRxNet e hConfirm para esta versão da sessão.

7.9 Desempenho do Protótipo

Nesta secção, fazemos uma análise de desempenho do protótipo resultante da concretização da arquitectura *RT-Appia* em C++ e para o sistema operativo Windows 2000. Com este exercício pretende-se aferir qual o custo de alguns dos mecanismos de composição do *RT-Appia*. Finalmente, é interessante comparar os valores obtidos com esta arquitectura com os valores para o pior caso obtidos com as ferramentas de análise de escalonabilidade descritas no capítulo anterior.

Para alcançar os objectivos atrás enunciados, foi concretizada e avaliada a ar-

Listagem 7.5: CanSession (hDataTx e hRxNet).

```
// ...
void CanSession::hDataTx( RtEvent* e )
{    // Obter a estrutura de controlo do canal
     CMSG *pCmsg = getCanTxMsg( e->getChannel() );
     m = e->getMessage();
     pCmsg->len = m->toByteArray( (unsigned char *)&pCmsg->data[1] );
     if ( pCmsg->len == 0 ) // mensagem de commit (RTR)
          pCmsg->len = NTCAN_RTR;
     else                          // mensagem de dados (DATA)
     {
          pCmsg->data[0] = ((RtEventClass *)(e->myClass()))->myClassId();
          pCmsg->len += 1;
     }
     pCmsg->id = m->getPriority();
     writeCan( pCmsg, len ); // invocar o metodo para transmitir a mensagem
     e->go();
}

void CanSession::hRxNet( RTChannel* channel )
{
     CMSG* pCmsg = readCan( channel ); // invocar o metodo para ler uma mensagem
     if ( pCmsg->len & NTCAN_RTR ) { // foi recebida uma mensagem de COMMIT
          RtEvent* e = channel->getPoolEvent( commitClass );
          e->setDirection( UP );
          e->init( );
          e->go();
     }
     else {    // foi recebida uma mensagem de dados
          cm->len &= 0x0f;
          int idClass = cm->data[0]; // obter a classe do evento com o identificador do primeiro octeto
          RtEventClass* eClass = getEventClassHandle( idClass );
          if ( eClass != NULL ) {
               RtEvent* e = channel->getPoolEvent( eClass );
               Message* m = e->getMessage();
               m->discard( );
               m->push((BYTE *)cm->data, 1, cm->len-1 );
               e->setDirection( UP );
               e->init( );
               e->go();
          }
          else
               //Erro: identificador desconhecido ...
     }
}
// ...
```

Listagem 7.6: CanSession (hConfirm, hRxNet e ChannelInitHandler).

```
// ...
class TxNetEvent : public RtEvent {
public:
    TxNetEvent( ) : RtEvent( )      {
        _myClass = (RtObjectClass *)txNetClass;
    }
};

class RxNetEvent : public RtEvent {
public:
    RxNetEvent( ) : RtEvent( ) {
        _myClass = (RtObjectClass *)rxNetClass;
    }
};

void CanSession::ChannelInitHandler( RtEvent* e ) {
    // ...
    RxNetEvent* rxNet = (RxNetEvent *)rxNetClass->make( );
    rxOverlapped[chId].hEvent = channel->setOSEvent( *rxNet, *this );

    TxNetEvent* txNet = (TxNetEvent *)txNetClass->make( );
    txOverlapped[chId].hEvent = channel->setOSEvent( *txNet, *this );
    // ...
}

void CanSession::hConfirm( RtEvent* e ) {
    CMSG *pCmsg;
    RtEvent* e;

    int chId = getChannelId( e->getChannel( ) ); // obter o identificador do canal
    pCmsg = readCan( channel );          // invocar o metodo para ler uma mensagem
    e = channel->getPollEvent( confirmClass );
    e->init( channel, UP );
    e->go();
}

void CanSession::hRxNet( RtEvent* e ) {
    // ...
    // obter o identificador do canal
    int chId = getChannelId( e->getChannel( ) );
    // ...
}
// ...
```

quitectura com dois canais apresentada na Figura 7.3. Note-se que a maioria das medições que se apresenta de seguida não pretendem calcular o tempo de computação ou de execução para o pior caso mas sim obter uma estimativa aproximada deste valor, através da execução de múltiplas instâncias das primitivas em causa. Para conhecer o valor para o pior caso seria necessário possuir informação precisa acerca da concretização do sistema operativo, que não existe disponível no caso do Windows 2000.

Cada nó do sistema é constituído por um computador pessoal com um processador Pentium III a $448Mhz$ da família x86, que executa o sistema operativo Windows 2000 da Microsoft. Os computadores estão interligados com um barramento $CAN2.0$ através de módulos $PCI/200$ da esd (electronic system design gmbh). A interface com o sistema operativo é realizada com um gestor de dispositivo fornecido pelo fabricante. Infelizmente o fabricante não possui um gestor de dispositivo para o núcleo ETS Kernel da Phar Lap Inc. A medição dos tempos de execução num sistema embebido facilitaria o isolamento do *RT-Appia* face a outras actividades concorrentes do sistema operativo, permitindo obter uma maior previsibilidade na medição dos tempos.

Isolaram-se ambos os nós da rede Ethernet e de todas as aplicações a serem executadas no sistema operativo e executou-se o processo *RT-Appia* com a máxima prioridade permitida pelo sistema operativo (REALTIME_PRIORITY_CLASS), atribuindo a cada canal uma prioridade diferente. Foram utilizados identificadores de $11bits$ ($CAN2.0A$) e uma taxa de transmissão de $1Mbps$ no barramento de campo.

Foi utilizada a versão da sessão CanSession, com uma interface com o gestor do dispositivo, notificada por eventos. Deste modo, a recepção e a confirmação da transmissão de mensagens, são sinalizadas pelo gestor de dispositivo utilizando o mecanismo de eventos do sistema operativo. É associado o evento RxNetEvent ao evento do sistema operativo definido para a recepção de mensagens e o evento TxNetEvent ao evento definido para a confirmação de uma transmissão que, são inseridos pelo canal na lista de eventos escalonáveis quando os respectivos eventos ocorrem. São então

invocados pelo escalonador de eventos do canal, os gestores hRxNet e hConfirm, que foram descritos na secção anterior.

Para medir os tempos de execução fizeram-se actuar (ON/OFF) os sinais DTR e ou RTS da porta série RS232 do computador, entre um conjunto de instruções de código que se pretende analisar. São invocados os procedimentos do sistema operativo para activar um sinal (ON) e desactivar (OFF) o mesmo sinal, respectivamente antes e depois do conjunto de instruções que se pretende analisar. Os sinais lógicos obtidos são visualizados utilizando os diversos canais disponíveis de um analisador de estados lógicos (Hewlett Packard $54620A$). Infelizmente não possuímos uma placa $GPIB$ que permita a aquisição dos sinais através de um computador, pelo que foi utilizado o modo $AutoStore$ do analisador.

Uma das limitações deste método de análise é o facto de não se conhecer com precisão o conjunto de instruções máquina a analisar, que resultam da compilação linguagem de alto nível (incluindo o das próprias instruções que actuam os sinais da porta série), nem a possível interferência de outras actividades, como por exemplo a execução de rotinas de interrupção, faltas de página em *caches* e execução especulativa em *pipelines*. Foram realizadas cerca de 3500 amostras em cada um dos tempos de execução apresentados e registados os tempos mínimo (min), máximo médio (max_{med}) e máximo esporádico (max_{esp}). O valor max_{med} corresponde a cerca $99,5\%$ das amostras, pelo que foi este o valor adoptado na análise de escalonabilidade.

Para ser um pouco mais preciso foram medidos os tempos de execução para o par de procedimentos que actuam os sinais da porta série (ON e OFF). Este tempo é depois subtraído ao tempo de execução obtido para o conjunto de instruções sob analise. Para o par de procedimentos, obteve-se o valor *min* de $40\mu s$ e o valor max_{esp} de $69\mu s$. No entanto o valor max_{med} foi de $56\mu s$. Deste modo o erro médio introduzido por este método é de $16\mu s$. Como se pretendem medir os tempos de execução para o pior caso, subtraiu-se sempre o valor mínimo obtido ($40\mu s$). Por exemplo, o pior tempo de execução médio, medido para a invocação do método go de uma sessão que insere

um novo evento na lista de eventos foi de $51\mu s$. A este tempo temos que subtrair o tempo mínimo gasto a executar o par de procedimentos que actua a porta série ($40\mu s$), o que resulta num tempo de $11\mu s$. Esta aproximação foi utilizada em todos os tempos medidos pelo que os tempos apresentados em seguida possuem esta correcção. A Figura 7.4 ilustra o método de medição.

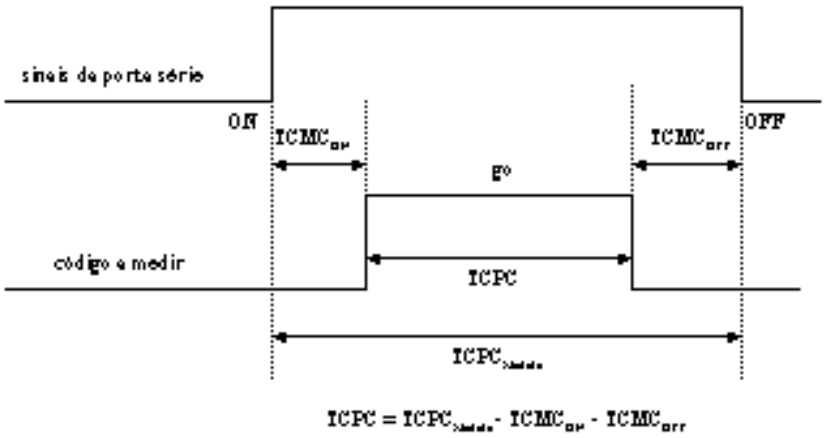

Figura 7.4: Representação do método de medição do TCPC. A sigla TCMC indica o tempo de computação para o melhor caso.

7.9.1 Medição dos piores tempos de execução

Os piores tempos de execução, mais relevantes, estão relacionados com a inserção e remoção de eventos na lista de eventos escalonáveis. O pior tempo de execução medido para inserir um evento na lista, C_{SchIn}, (invocação do método `go`) varia entre um min de $5\mu s$ e um max_{med} de $11\mu s$. Na remoção de eventos, a sobrecarga introduzida pelo escalonador de eventos depende do evento a remover. O custo, max_{med}, de remover um evento da lista de eventos escalonáveis (C_{SchRem} na equação 5.2) é de $3\mu s$. No entanto quando é seleccionado um novo evento (diferente do que estava em execução) é necessário adicionar o tempo gasto a executar o gestor do canal (o evento corrente é devolvido ao canal), assim como o custo da execução do método

waitForEvent (considerando que existe uma notificação activa e a lista de eventos está vazia). Esta situação está representada na Figura 7.5.

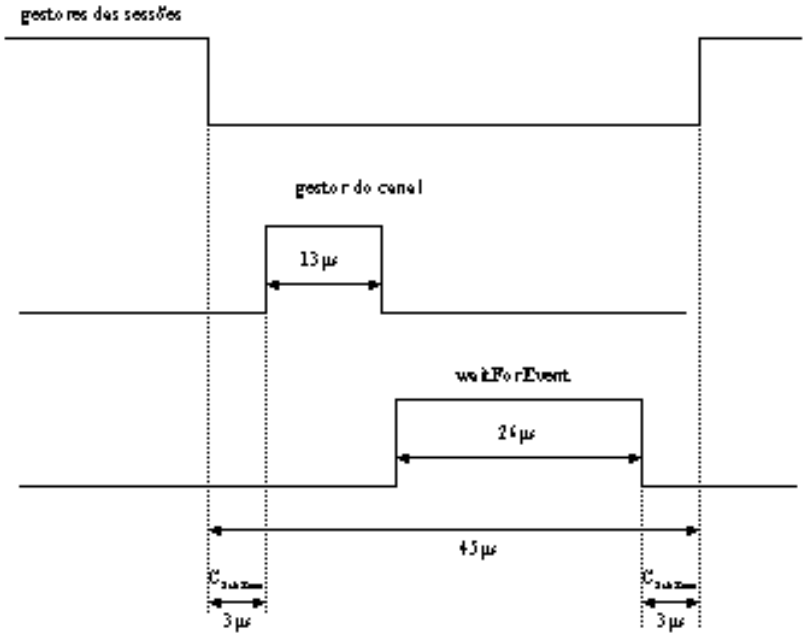

Figura 7.5: Representação dos piores tempos medidos no escalonador de eventos.

O sinal de topo representa a execução dos gestores das sessões e o sinal do meio a execução do gestor do canal. Ao remover um evento da lista de eventos escalonáveis que tenha executado a sua última sessão, o escalonador executa o gestor do canal e, como a lista de eventos está vazia, o canal invoca o método waitForEvent, representado pelo terceiro sinal. O tempo gasto a escalonar um evento activado na interface de um canal para este cenário será de $45\mu s$. Este é o cenário para o pior caso mas existem outros cenários possíveis que são casos particulares deste. Por exemplo, um evento pode ser devolvido ao canal quando existem outros eventos na lista de eventos escalonáveis e neste caso não é invocado o método waitForEvent; ou quando uma sessão decide não propagar um evento (e não é invocado o gestor do canal), mas a lista de eventos está vazia.

A Tabela 7.1 apresenta os valores observados para os procedimentos associados ao

escalonamento de eventos.

Tarefa	*TCPC*		
	min	max_{med}	max_{esp}
$waitForEvent$	12	26	40
C_{SchIn}	5	11	24
C_{SchRem}	1	3	7
$C_{H_{channel}}$	5	13	24

Tabela 7.1: Piores tempos de execução observados para a inserção e remoção de eventos, em μs.

O tempo gasto pelo escalonador de eventos a escalonar um evento activado na interface de um canal (C^i_{Sch} na equação 5.1) é de $3 + 26 = 29\mu s$ e o tempo gasto a escalonar um evento da lista de eventos escalonáveis e que é de $C_{Sch} = 3 + 11 = 14\mu s$. Deste modo o escalonamento de um novo evento sofre um *jitter* que pode variar entre um valor mínimo de $14\mu s$ e que, no pior caso é de $45\mu s$. Os valores utilizados na equação 5.1 do Capítulo 5 correspondem aos valores max_{med} medidos e apresentados nesta secção, nomeadamente, $C_{H_{channel}} = 16\mu s$, $C_{Sch} = 14\mu s$ e $C^i_{Sch} = 29\mu s$.

Os tempos medidos para os restantes métodos mais utilizados por uma sessão estão representados na Tabela 7.2. Os tempos de execução apresentados para os métodos createEventPool e setOSEvent foram obtidos para uma lista com oito eventos.

Objecto	*metodo*	*TCPC*		
		min	max_{med}	max_{esp}
RtEvent	init	5	12	25
RtEvent	createTimer	48	76	107
RtEvent	startTimer	11	22	40
RtEvent	stopTimer	8	16	24
RTChannel	createEventPool	83	108	166
RTChannel	getPoolEvent	5	22	41
RTChannel	putPoolEvent	5	12	26
RTChannel	setOSEvent	136	178	246
RTChannel	preHandleSync	4	12	21
RTChannel	postHandleSync	4	10	28
RTChannel	startChannel	41	70	85
RTChannel	stopChannel	12	22	30

Tabela 7.2: Piores tempos de execução observados medidos em μs.

A Tabela 7.3 apresenta os tempos de execução registados para a execução dos gestores. Na análise de escalonabilidade realizada no Capítulo 5 foram utilizados os valores máximos médios.

Tarefa	*TCPC*		
	min	max_{med}	max_{esp}
Frag_L3_D	11	20	32
Frag_L2_D	37	80	97
Frag_L1_D	14	33	41
Frag_L0_D	24	41	52
Frag_L0_U	25	43	56
Frag_L1_U	12	34	50
Frag_L2_U	22	37	54
Frag_L3_U	10	29	38
Comm_L0_D	20	38	49
Comm_L0_U	18	38	48
Comm_L1_U	15	34	44
Conf_L0_U	17	36	45
Conf_L1_U	20	34	42

Tabela 7.3: Piores tempos de execução observados para os gestores, em μs.

Os tempos de transmissão das mensagens num canal foram obtidos medindo a diferença entre a invocação do procedimento canWrite do gestor do dispositivo para a transmissão de uma mensagem e a activação do evento do sistema operativo que sinaliza a recepção da mensagem pelo gestor do dispositivo no método waitForEvent. Os tempos medidos incluem, assim, algum processamento adicional do gestor do dispositivo. Para uma mensagem de dados com oito octetos, obteve-se um tempo máximo de $204\mu s$ e um tempo mínimo de $160\mu s$. Para uma mensagem de COMMIT, obteve-se um tempo máximo de $108\mu s$ e um tempo mínimo de $64\mu s$.

7.9.2 Medição dos piores tempos de resposta

A Figura 7.6 representa a execução dos gestores das sessões para os dois canais em ambos os nós, obtidos com o analisador de estados lógicos. O canal $RC0$, de maior prioridade, é visualizado nos canais 8 e 0 do analisador. O canal 8 apresenta a execução dos gestores no nó 2 com a transmissão de uma mensagem pela aplicação e o canal

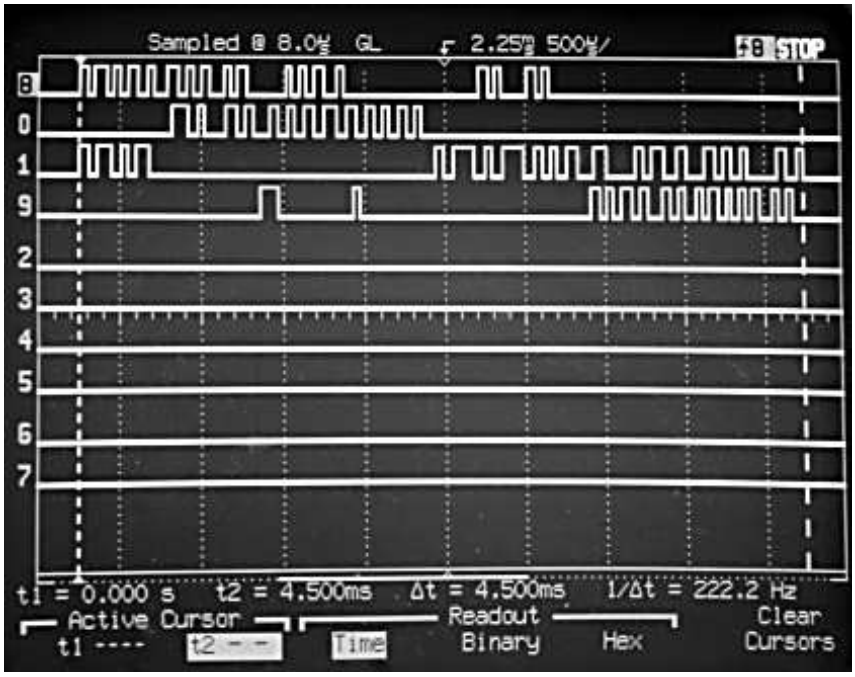

Figura 7.6: Imagem do analisador de estados lógicos captando a execução dos gestores de dois canais, com os relógios sincronizados.

0 visualiza a recepção da mensagem no nó 1. A transmissão de uma mensagem para o canal $RC1$, de menor prioridade, é visualizada no canal 1 do analisador e a sua recepção no canal 9. O cenário apresentado corresponde à situação em que os temporizadores de ambos os canais (evento FRAG) estão sincronizados. Neste cenário, o tempo de resposta para o canal $RC0$ é de $2200\mu s$[2] e o tempo de resposta para o canal $RC1$ é de $3680\mu s$. É possível verificar a interferência sofrida pelo canal $RC1$ quer no nó 1 (canal 1 do analisador) quer no nó 2 (canal 9 do analisador), devido ao canal $RC0$.

Este deveria ser o cenário ideal para a execução de ambos os canais, mas não reflecte o pior caso. O pior caso ocorre quando, devido a um desvio na sincronização dos relógios nos dois nós, as tarefas de ambos os canais são libertadas em simultâneo num dos processadores, provocando a máxima interferência na execução do canal de menor prioridade. Este cenário está representado na Figura 7.7 onde é possível verificar que

[2]Este valor é obtido subtraindo $920\mu s$ ao valor medido, correspondente aos tempos de actuação dos sinais da porta série.

o canal $RC1$ no nó emissor (canal 1 do analisador) só é executado quando todos os gestores do canal $RC0$ forem executados (canal 0 do analisador). Para este cenário, o tempo de resposta do canal $RC0$ é de $1710\mu s$ e o tempo de resposta para o canal $RC1$ é de $3920\mu s$.

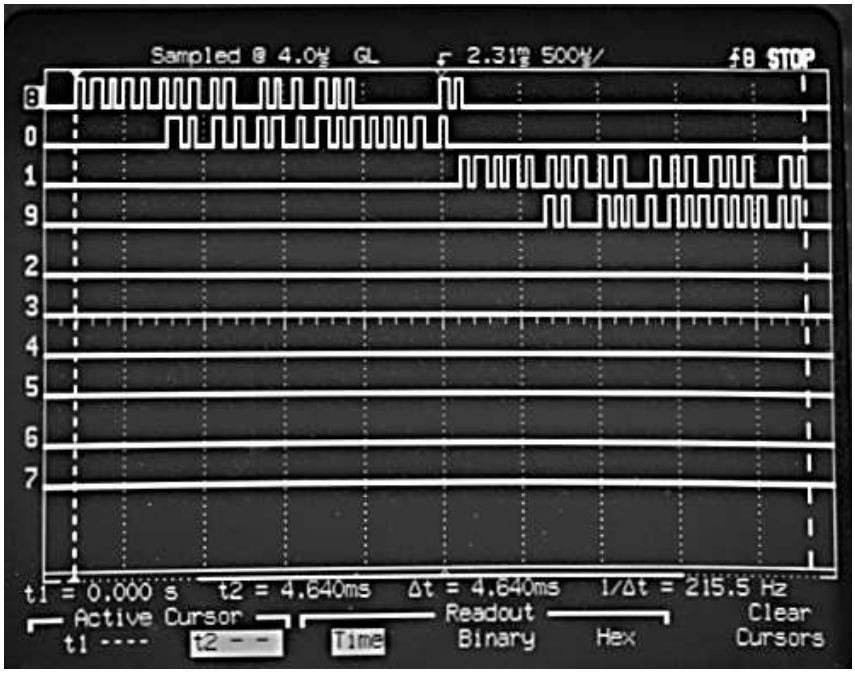

Figura 7.7: Imagem do analisador de estados lógicos captando a execução dos gestores de dois canais, com um desvio na sincronização dos relógios.

7.10 Avaliação e Discussão

Um dos principais objectivos *RT-Appia* foi o de garantir que se obtinha uma ambiente integrado que permite obter, a partir do mesmo código fonte, não só a análise de escalonabilidade mas também uma versão executável da composição de protocolos.

Neste capítulo apresentámos uma concretização da arquitectura *RT-Appia*, que foi utilizada para desenvolver um protótipo da composição de protocolos anteriormente utilizada para ilustar a análise de escalonabilidade. Infelizmente, por razões logísticas

inultrapassáveis, só foi possível obter um protótipo completamente funcional sobre o sistema operativo Windows 2000. Este facto impossibilitou uma análise rigorosa do tempo de computação no pior caso dos mecanismos do *RT-Appia*. No entanto, a estimativa deste valor, obtida através da múltipla execução das primitivas mais relevantes, permite aferir o custo destes mecanismos.

Comparando os resultados observados no protótipo com os valores obtidos pela ferramenta de análise de escalonabilidade é interessante observar o pessimismo introduzido pela análise de escalonabilidade, no cálculo dos piores tempos de resposta. Como seria de esperar, esse pessimismo é mais visível no canal de menor prioridade, onde o valor calculado utilizando a aproximação das sub-transacções aplicada ao modelo dos desfasamentos estáticos foi de $5100\mu s$ e o tempo de resposta medido de $3920\mu s$. Para o canal de maior prioridade essa diferença já não é tão significativa ($1903\mu s$ e $1710\mu s$ respectivamente).

8

Conclusões

Nesta dissertação foi apresentada uma moldura de suporte à concepção, composição e execução de sistemas de comunicação modulares para aplicações de tempo-real que designámos por *RT-Appia*. Um dos objectivos na construção desta moldura foi a obtenção de um ambiente de desenvolvimento integrado que permitisse realizar a análise do comportamento temporal das composições de protocolos, assim como a obtenção de código executável, a partir de um único código fonte.

Para atingir este objectivo, concretizaram-se os seguintes componentes da moldura:

- Um modelo de suporte à composição e desenvolvimento de protocolos de comunicação que facilita a posterior análise temporal da composição. Este modelo é uma extensão do modelo do sistema *Appia*. O modelo caracteriza-se por permitir que cada protocolo seja desenhado de forma independente dos restantes protocolos a incluir na composição final. Possui, no entanto, os mecanismos necessários para extrair informação sobre as inter-dependências entre os protocolos de forma a reduzir o pessimismo da análise temporal da composição resultante.

- Uma ferramenta de análise temporal de composição de protocolos, obtida através da adaptação e extensão das ferramentas desenvolvidas por Tindell (1994). A ferramenta suporta diferentes modelos de análise e inclui uma adaptação do mod-

elo dos desfasamentos de tempo às características dos sistemas de comunicação modulares.

- Uma ferramenta que automatiza o processo de atribuição e optimização de prioridades.

- Um ambiente de execução de composições de protocolos, integrado com as ferramentas anteriormente enumeradas e completamente funcional. O protótipo foi desenvolvido na linguagem C+ e pode ser executado em sistemas operativos como o Windows NT/Windows 2000 ou em sistemas embebidos utilizando o núcleo ETS Kernel da Phar Lap Inc. Acreditamos que as opções de concepção tomadas para este ambiente podem ser reaproveitadas para desenvolver o ambiente de execução do *RT-Appia* sobre outros sistemas operativos de tempo-real.

Consideramos que os objectivos definidos para este trabalho foram atingidos. Apesar dos mecanismos de composição propostos pelo *RT-Appia* possuírem limitações, como se referiu no Capítulo 4, acreditamos que a sua expressividade permite capturar os requisitos de um largo leque de protocolos de comunicação. A aproximação seguida no *RT-Appia* tem a vantagem de utilizar o mesmo compilador para gerar a informação de controlo necessária para a análise temporal e para gerar o código executável. Ferramentas mais sofisticadas, como as baseadas na análise de código, podem ser muito mais difíceis de desenvolver e manter. Para além disso, o programador não necessita de introduzir na implementação qualquer tipo de directivas adicionais. Deste modo, esta ferramenta é não só mais simples de usar como também mais eficiente relativamente ao esforço de desenvolvimento. Finalmente, a informação fornecida para suportar a extracção do diagrama de eventos é também útil em tempo de execução para optimizar a execução da composição de protocolos e para realizar diversas verificações em tempo de execução que validam a correspondência entre o diagrama de eventos construído e a execução da implementação.

8.0.1 Trabalho Futuro

Como trabalho futuro, seria interessante incluir na moldura uma ferramenta para extrair, de forma automática, o Tempo de Execução no Pior Caso dos gestores de eventos e também enriquecer os mecanismos de composição do *RT-Appia* de forma a aumentar a sua expressividade. A implementação de um conjunto de serviços de comunicação de grupo, por exemplo, permitiria explorar de modo mais exaustivo possíveis limitações tanto nos mecanismos de composição, como também na extracção do diagrama de eventos.

Seria também interessante estudar a utilização de servidores periódicos na execução dos gestores síncronos definidos para a camada de acesso ao meio (ver subsecção 7.8.4). Existem diversos algoritmos de *servidores periódicos* como por exemplo o *Sporadic Server* (Sprunt *et al.*, 1989), o *Deferrable Server* (Strosnider *et al.*, 1995) e o *Slack Stealing* (Lehoczky & Ramos-Thuel, 1992), entre outros. Outro ponto interessante seria o de expandir a ferramenta para analisar aplicações com canais heterogéneos, compostas por canais periódicos e não periódicos. Para além disso, dadas as recentes publicações sobre testes de escalonabilidade (Bini *et al.*, 2003) (Bini & Buttazzo, 2004) (Abdelzaher *et al.*, 2004), justificava-se a inclusão de um subsistema de controlo de admissão no *RT-Appia*.

No âmbito da atribuição de prioridades seria interessante explorar outras heurísticas para optimizar a atribuição de prioridades a um conjunto de canais. Dadas as características de escalonamento do *RT-Appia* poderá ser possível desenvolver uma heurística, com um tempo de execução mais reduzido que o do arrefecimento simulado. No entanto estamos em crer que o algoritmo do arrefecimento simulado constitui uma boa referência para comparação com outras heurísticas.

Finalmente seria também interessante aferir o desempenho do protótipo sobre um núcleo concebido para ambientes embebidos, como por exemplo o núcleo ETS Kernel, o que facilitaria a comparação dos resultados teóricos com as medições observadas.

Bibliografia

ABDELZAHER, T., DAWSON, S., FENG, W., JAHANIAN, F., JOHNSON, S., MEHRA, A., MITTON, T., SHAIKH, A., SHIN, K., WANG, Z., & ZOU, H. 1997. ARMADA Midleware Suite. *In: Proceedings of the Workshop on Middleware for Real-Time Systems and Services*. San Francisco, CA: IEEE.

ABDELZAHER, TAREK, SHAIKH, ANEES, JAHANIAN, FARNAM, & SHIN, KANG. 1996 (4). *RTCAST: Lightweight Multicast for Real-Time Process Groups*. Tech. rept. CSE-TR-291-96.

ABDELZAHER, TAREK F., & SHIN, KANG G. 2000. Period-Based Load Partitioning and Assignment for Large Real-Time Applications. *IEEE Transactions on Computers*, **49**(1).

ABDELZAHER, TAREK F., SHARMA, VIVEK, & LU, CHENYANG. 2004. A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling. *IEEE Trans. Computers*, **53**(3), 334–350.

ALMEIDA, L., PEDREIRAS, P., & FONSECA, J. A. 2002. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics - special issue on Factory Communication Systems*, **49**(6), 1189– 1201.

ALTENBERND, P. 1996. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. Ph.D. thesis.

ALTENBERND, PETER, & HANSSON, HANS. 1998. The Slack Method: A New Method for Static Allocation of Hard Real-Time Tasks. *Real-Time Systems*, **15**(2), 103–130.

AMNELL, T., BEHRMANN, G., BENGTSSON, J., DÍARGENIO, P.R., DAVID, A., FEHNKER, A., HUNE, T.S., JEANNET, B., LARSEN, K.G., MOLLER, M.O., PETTERSSON, P., WEISE, C., & YI, W. 2001. UPPAAL - Now, Next, and Future, Modeling and Verification of Parallel Processes. *Pages 100–125 of:* CASSEZ, F., JARD, C., ROZOY, B., & RYAN, M. (eds), *LNCS 2067*. Springer Verlag.

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

BALL, THOMAS, & RAJAMANI, SRIRAM K. 2001. Automatically Validating Temporal Safety Properties of Interfaces. *Pages 103–122 of: SPIN 2001, Workshop on Model Checking of Software*. LNCS 2057, Springer Verlag.

BHATTI, N., HILTUNEN, M., SCHLICHTING, R., & CHIU, W. 1998. Coyote: A System for Constructing Fine-grain Configurable Communication Services. *ACM Trans. on Computer Systems*, **16**(4), 321–366.

BINI, ENRICO, & BUTTAZZO, GIORGIO C. 2004. Schedulability Analysis of Periodic Fixed Priority Systems. *IEEE Trans. Computers*, **53**(11), 1462–1473.

BINI, ENRICO, BUTTAZZO, GIORGIO C., & BUTTAZZO, GIUSEPPE M. 2003. Rate Monotonic Analysis: The Hyperbolic Bound. *IEEE Transactions on Computers*, **52**(7), 933–942.

BOLLELLA, GREG, & GOSLING, JAMES. 2000. The Real-Time Specification for Java. *IEEE Computer*, **33**(6), 47–54.

BOLLELLA, GREG, GOSLING, JAMES, BROSGOL, BENJAMIN, DIBBLE, PETER, FURR, STEVE, & TURNBULL, MARK. 2000. *The Real-Time Specification for Java*. Java Series. Addison-Wesley. URL: `www.javaseries.com/rtj.pdf`.

BROSTER, I., BURNS, A., & GUEZ NAVAS, G. 2002. Probabilistic analysis of CAN with faults.

CHAPMAN, R., BURNS, A., & WELLINGS, A. J. 1994. Integrated program proof and worst-case timing analysis of SPARK Ada. *In: Workshop on language compiler and tool support for real-time systems*.

CHAPMAN, R., BURNS, A., & WELLINGS, A. J. 1996. Combining Static Worst-Case Timing Analysis and Program Proof. *Real-Time Systems*, **11**(2), 145–171.

CHEVOCHOT, PASCAL, & PUAUT, ISABELLE. 2000. Holistic schedulability analysis of a fault-tolerant real-time distributed run-time support. *Pages 355–362 of: RTCSA*.

CRISTIAN, FAND DANCEY, R. D., & DEHN, J. 1990 (Apr.). *Fault-Tolerance in the Advanced Automation System*. Research Report RJ 7424 (69595). IBM.

DAS, RAJSEKHAR, HILTUNEN, MATTI A., & SCHLICHTING, RICHARD D. 1999. *Supporting Configurability and Real-Time in RTD Channels*. Tech. rept. Department of Computer Science, The University of Arizona.

DERTOUZOS, M. L. 1974. Control robotics: the procedural control of physical processes. *Information Processing*, 74.

DIPIPPO, LISA CINGISER, WOLFE, VICTOR FAY, ESIBOV, LEVON, COOPER, GREGORY, BETHMANGALKAR, RAMACHANDRA, JOHNSTON, RUSSELL, THURAISINGHAM, BHAVANI, & MAUER, JOHN. 2001. Scheduling and Priority Mapping for Static Real-Time Middleware. *Real-Time Syst.*, **20**(2), 155 – 182.

DWYER, MATTHEW, HATCLIFF, JOHN, JOEHANES, ROBY, LAUBACH, SHAWN, PASAREANU, CORINA, WILLEM, ROBBY VISSER, & ZHENG, HONGJUN. 2001 (May). Tool-supported Program Abstraction for Finite-state Verification. *In: Proceedings of the 23rd International Conference on Software Engineering*.

ELMOHAMED, M., FOX, G., & CODDINGTON, P. 1998 (Apr.). *Guaranteeing Message Latencies on Controller Area Network (CAN)*. Technical Report DHPC045. Northeast Parallel Architectures Center, Syracuse University, NY, USA.

GAMMA, ERICH, HELM, RICHARD, JOHNSON, RALPH, & VLISSIDES, JOHN. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GARCIA, JOSÉ J., & HARBOUR, MICHAEL G. 1995. *Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems*. Technical Report. Departamento de Electrónica, Universidad de Cantabria, SPAIN.

GHEITH, AHMED, & SCHWAN, KARSTEN. 1993. CHAOSarc:Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications. *ACM Transactions on Computer Systems*, **11**(1), 33–72.

GUTIÉRREZ, J. C. PALENCIA, & HARBOUR, MICHAEL GONZÁLEZ. 2003. Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. *Pages 3–12 of: ECRTS*.

HARBOUR, MICHAEL GONZÁLEZ, GARCÍA, J. J. GUTIÉRREZ, GUTIÉRREZ, J. C. PALENCIA, & MOYANO, J. M. DRAKE. 2001. MAST: Modeling and Analysis Suite for Real Time Applications. *Pages 125–134 of: ECRTS*.

HARTER, P. K. 1984. *Response times in level structured systems*. Technical Report CU-CS-269-94. Department of Computer Science, University of Colorado, USA.

HARTER, P. K. 1987. Response times in level structured systems. *ACM Transactions on Computer Systems*, **5**(3), 232–248.

HAYDEN, M. 1998. *The Ensemble System*. Ph.D. thesis, Cornell University, Computer Science Department.

HILTUNEN, MATTI A., HAN, XIAONAN, & SCHLICHTING, RICHARD D. 1996. *Real-Time Issues in Cactus*. Technical Report AZ85721. Department of Computer Science, University of Arizona, Tucson.

HILTUNEN, MATTI A., SCHLICHTING, RICHARD D., HAN, XIAONAN, CARDOZO, MELVIN, & DAS, RAJSEKHAR. 1999. Real-Time Dependable Channels: Customizing QoS Attributes for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, **10**(6), 600–612.

HOLZMANN, G.J., & SMITH, M.H. 2002. An automated verification method for distributed systems software based on model extraction. *Trans. on Software Engineering*, **28**(4), 364–377.

HUTCHINSON, NORMAN C., & PETERSON, LARRY. 1991. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engeneering*, **17**(1), 64–76.

INGBER, L. 1996. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics*, **25**(1), 33–54.

ISHIKAWA, Y., TOKUDA, H., & MERCER, C. M. 1992. An object-oriented real-time programming language. *IEEE Computer*, **25**(10), 66–73.

ISO (ed). 1993. *ISO International Standard 11898 - Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for high-speed communication*. ISO.

JONSSON, JAN, & SHIN, KANG G. 1997. Deadline Assignment in Distributed Hard Real-Time Systems with Relaxed Locality Constraints. *Pages 0– of: ICDCS*.

JOSEPH, M., & PANDYA, P. 1986. Finding Response Times in a Real-Time System. *The Computer Journal (British Computer Society)*, **29**(5), 390–395.

KIRKPATRICK, S., GELATT, C. D., & VECCHI, M. P. 1983. Optimization by Simulated Annealing. *Science, Number 4598, 13 May 1983*, **220, 4598**, 671–680.

KLIGERMAN, E., & STOYENKO, A. 1986. Real-Time Euclid: A Language for Real-Time Systems. *IEEE Transactions on Software Engeneering*, **SE-12**(9), 941–949.

KODASE, SHARATH, WANG, SHIGE, GU, ZONGHUA, & SHIN, KANG G. 2003. Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-Time Systems Using Shared Buffers. *Pages 181–188 of: IEEE Real Time Technology and Applications Symposium*.

KOPETZ, H., & GRUNSTEIDL, G. 1994. TTP-A Ptotocol for Fault-Tolerant Real-Time Systems: a Multi-primitive Group Communication Service. *IEEE Transactions on Computers*, **27**(Jan.), 14–23.

KOPETZ, HERMANN, & AL. 1989. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, **9**(Feb.), 25–40.

LEHOCZKY, J. P., & RAMOS-THUEL, S. 1992 (Sep). An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. *Pages 110–123 of: Proceedings of the IEEE Real-Time Systems Symposium*.

LEHOCZKY, J. P., SHA, L., & DING, Y. 1989. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. *Pages 166–171 of: Proceedings of the 10h IEEE Real-Time Systems Symposium*. Santa Monica, California, USA: Computer Society Press, for IEEE.

LEHOCZKY, J. P., SHA, L., & DING, Y. 1990. Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines. *Pages 201–209 of: Proceedings of the 11h IEEE Real-Time Systems Symposium*. Lake Buena Vista, Florida, USA: Computer Society Press, for IEEE.

LEUNG, J. Y. T., & WHITEHEAD, J. 1982. On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, **2**(Dec), 237–250.

LIN, MAN, KARLSSON, LARS, & YANG, LAURENCE T. 2000 (Oct). Heuristic techniques: scheduling partially ordered tasks in amulti-processor environment with

tabu search and genetic algorithms. *Pages 515–523 of: Seventh International Conference on Parallel and Distributed Systems*.

LIU, C. L., & LAYLAND, JAMES W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, **20**(1), 46–61.

MENA, DANIEL C. BÜNZLI SERGIO, & NESTMANN, UWE. 2005. Protocol Composition Frameworks: A Header Driven Model. *In: Proceedings of the 4th IEEE International Symposium on Network Computing and Applications*. Cambridge, MA, USA: IEEE.

MIRANDA, H., PINTO, A., & RODRIGUES, L. 2001. Appia, a flexible protocol kernel supporting multiple coordinated channels. *Pages 707–710 of: Proceedings of the 21st International Conference on Distributed Computing Systems*. Phoenix, Arizona: IEEE.

MISHRA, S., PETERSON, L., & SCHLICHTING, R. 1993. Consul: A communication substrate for fault-tolerant distributed programs. *Distributes Systems Engineering*, **1**(2), 87–103.

MOGUL, J., RASHID, R., & ACCETTA, M. 1987. The Packet Filter: An Efficient Mechanism for User-level Network Code. *Pages 39–51 of: Proc. of the 11th Symposium on Operating Systems Principles, (ACM SIGOPS)*, vol. 21.

NATALE, MARCO DI, & STANKOVIC, JOHN A. 1995. Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control. *Pages 190–199 of: IEEE Real-Time Systems Symposium*.

O. GUDMUNDSSON, D. MOSSE, A. AGRAWALA, & TRIPATHY, S. 1990 (Oct). Maruti: A hard real-time operating system.

O'MALLEY, S., & PETERSON, L. 1992. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, **10**(2), 110–143.

PALENCIA, J. C., & HARBOUR, M. GONZALEZ. 1998 (Dec.). Schedulability Analysis for Tasks with Static and Dynamic Offsets. *Pages 26–37 of: Proc. IEEE Real-time Systems Symposium*.

PALENCIA, J. C., & HARBOUR, M. GONZALEZ. 1999. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems. *Pages 328–339 of: IEEE Real-Time Systems Symposium*.

PARK, C. Y., & SHAW, A. C. 1991. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, **24**(5), 48–57.

PASAJE, JULIO L. MEDINA, HARBOUR, MEDINA GONZÁLEZ, & DRAKE, JOSÉ M. 2001. MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time System. *Pages 245–256 of: IEEE Real-Time Systems Symposium*.

PENG, DAR-TZEN, SHIN, KANG G., & ABDELZAHER, TAREK F. 1997. Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems. *IEEE Trans. Software Eng.*, **23**(12), 745–758.

PINHO, LUÍS MIGUEL, & VASQUES, FRANCISCO. 2000 (Sep). Integrating Inaccessibility in Response Time Analysis of CAN Networks. *Pages 77–84 of: Proceedings of the 3rd IEEE International Workshop on Factory Communication Systems (WFCS'2000)*.

POP, TRAIAN, ELES, PETRU, & PENG, ZEBO. 2003. Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time Systems. *Pages 257–266 of: ECRTS*.

PORTO, S. C. S., & RIBEIRO, C. C. 1994. A Tabu Search Approach to Task Scheduling on Heterogeneous Processors Under Precedence Constraints. *International Journal of High Speed Computing (IJHSC)*, **7**(1), 45–??

POSPISCHIL, G., PUSCHNER, P., VRCHOTICKY, A., & ZAINLINGER, R. 1992. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, **9**(5), 35–44.

PUSCHNER, P., & SCHEDL, A. 1993. A Tool for the Computation of Worst-Case Task Execution Times. *Pages 224–229 of: Proc. Euromicro Workshop on Real-time Systems*.

PUSCHNER, P., & SCHEDL, A. 1997. Computing Maximum Task Execution Times - A Graph-Based Approach. *Real-Time Systems*, **13**(1), 67–91.

PUSCHNER, PETER, & BURNS, ALAN. 1999. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems, Kluwer Academic Publishers*, **18**(2/3), 115–128.

RAMAMRITHAM, KRITHI. 1995. Allocation and Scheduling of Precedence-Related Periodic Tasks. *IEEE Transactions on Parallel and Distributed Systems*, **6**(4), 412–420.

RICHARD, M., RICHARD, P., & COTTET, F. 2001. Task and message priority assignment in automotive systems. *Pages 105–112 of: 4th FeT IFAC Conference on Fieldbus Systems and their Applications*.

RICHARD, M., RICHARD, P., & COTTET, F. 2003 (Sep). Allocating and scheduling tasks in multiple fieldbus real-time systems. *Pages 137– 144 of: IEEE Conference on Emerging Technologies and Factory Automation. ETFA '03.*, vol. 1.

RODRIGUES, J., & RODRIGUES, L. 2004 (Sept.). From running code to event-graphs: a pragmatic approach to derive WCRT of protocol compositions. *Pages 265–274 of: Proceedings of the 5th IEEE International Workshop on Factory Communication Systems*.

RODRIGUES, J., MIRANDA, H., VENTURA, J., & RODRIGUES, L. 2001. The design of RTAppia. *Pages 275–282 of: Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-Time Dependable Systems*. Rome, Italy: IEEE.

RODRIGUES, J., VENTURA, J., & RODRIGUES, L. 2002. Schedulability Analysis of an Event-based Real-Time Protocol Framework. *In: Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-Time Dependable Systems*. San Diego, USA: IEEE.

RODRIGUES, L., & VERÍSSIMO, P. 1991. xAMp: a Multi-primitive Group Communication Service. *In: Proceedings of the Eleventh Symposium on Reliable Distributed Systems*. Houston, USA: IEEE.

RUFINO, JOSÉ, VERÍSSIMO, PAULO, ARROZ, GUILHERME, ALMEIDA, CARLOS, & RODRIGUES, LUÍS. 1998 (June). Fault-Tolerant Broadcasts in CAN. *Pages 150–159 of: Digest of Papers, The 28th IEEE International Symposium on Fault-Tolerant Computing*. IEEE, Munich, Germany.

SALLEH, SHAHARUDDIN, & ZOMAYA, ALBERT Y. 1998. Multiprocessor Scheduling Using Mean-Field Annealing. *Pages 288–296 of: IPPS/SPDP Workshops*.

SANDNES, F. E., & MEGSON, G. M. 1996 (June). A Hybrid Genetic Algorithm Applied to Automatic Parallel Controller Code Generation. *In: IEEE Proceedings of the Eigth Euromicro Workshop on Real-Time Systems*.

SANTOS, J., FERRO, E., OROZCO, J., & CAYSSIALS, R. 1997. A Heuristic Approach to the Multitask-Multiprocessor Assignment Problem using the Empty-Slots Method and Rate Monotonic Scheduling. *Real-Time Syst.*, **13**(2), 167–199.

SHA, LUI, RAJKUMAR, RAGUNATHAN, & LEHOCZKY, JOHN. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, **39**(9), 1175–1185.

SHIN, KANG G. 1991. HARTS: A distributed real-time architecture. *IEEE Computer*, May, 25–35.

SPRUNT, BRINKLEY, SHA, LUI, & LEHOCZKY, JOHN P. 1989. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, **1**(1), 27–60.

SPURI, MARCO. 1996 (Apr.). *Holistic Analysis for Deadline Scheduled Real-Time Distributed Systems*. Rapport de Recherche 2873. Institut Nacional de Recherche en Informatique et en Automatique (INRIA).

STANKOVIC, JOHN A., & RAMAMRITHAM, KRITHI. 1990. Editorial: What is Predictability for Real-Time Systems? *The Journal of Real-Time Systems*, **2**, 247–254.

STANKOVIC, JOHN A., & RAMAMRITHAM, KRITHI. 1991. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, **8**(3), 62–72.

STOYENKO, A. 1987. *A Real-Time Language With A Schedulability Analyser*. Ph.D. thesis.

STROSNIDER, JAY K., LEHOCZKY, JOHN P., & SHA, LUI. 1995. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Trans. Comput.*, **44**(1), 73–91.

TIA, TOO-SENG, LIU, JANE W.-S., & SHANKAR, MALLIKARJUN. 1996. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Syst.*, **10**(1), 23–43.

TINDELL, BURNS, & WELLINGS. 1992a. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *The Journal of Real-Time Systems*, **4**.

TINDELL, K., BURNS, A., & WELLINGS, A. 1991. *Guaranteeing Hard Real Time End-to-End Communications Deadlines*.

TINDELL, K., BURNS, A., & WELLINGS, A. 1992b. *An Extensible Approach for Analysing Fixed Priority Hard Real-Time Tasks*. Technical Report YCS189. Department of Computer Science, University of York.

TINDELL, KENNETH WILLIAM. 1994 (Jan.). *Adding Time-Offsets to Schedulability Analysis*. Technical Report YCS221. Department of Computer Science, University of York.

TINDELL, KENNETH WILLIAM, BURNS, ALAN, & WELLINGS, ANDY. 1994 (Sept.). Calculating Controller Area Network (CAN) Message Response Times. *In: Proceedings of the IFAC Workshop on Distributed Computer Control Systems*. IFAC, Toledo, Spain.

TOKUDA, H., & MERCER, C. 1989a. Arts: A distributed real-time kernel. *Operating Systems Review*, **23**(3), 29–53.

TOKUDA, HIDEYUKI, & MERCER, CLIFFORD W. 1989b. ARTS: A Distributed Real-Time Kernel. **23**(3).

TOKUDA, HIDEYUKI, NAKAJIMA, TATSUO, & RAO, PRITHVI. 1990. Real-Time Mach: Towards a Predictable Real-Time System. *Pages 73–82 of:* USENIX (ed), *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*. Berkeley, CA, USA: USENIX.

TRAVOSTINO, FRANCO, MENZE, ED, & REYNOLDS, FRANKLIN. 1996 (Feb.). Paths: Programming with System Resources in Support of Real-Time Distributed Applications. *In: Proceedings of the 2nd IEEE Workshop on Object-Oriented Real-Time Dependable Systems*.

VAN RENESSE, R., BIRMAN, K., & MAFFEIS, S. 1996. Horus: A flexible group communication system. *Communications of the ACM*, **39**(4), 76–83.

VENTURA, J. 2001 (July). *Análise do tempo de resposta da composição de micro-protocolos*. M.Phil. thesis, Faculdade de Ciências da Universidade de Lisboa.

VENTURA, J., RODRIGUES, J., & RODRIGUES, L. 2001. Timming Analysis of Object-Oriented Communication Protocols. *Pages 335–342 of: Proceedings of the 4rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*. Magdeburg, Germany: IEEE.

YOVINE, S. 1997. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*.